ESTRUCTURAS DE DATOS Y ALGORITMOS I
Grado en Ingeniería Informática
Convocatoria Extraordinaria (Ordinaria - incidencias)
Febrero 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) Sea H un heap de máximos cuyo estado inicial es el que se muestra en la Figura 01. Sea E = [3, 5, 8, 1, 7, 3] una secuencia de datos de entrada. Obtener, paso a paso e indicando en todo momento el proceso de razonamiento, cuál sería el heap resultante tras insertar en H los datos contenidos en E.

2. (1.5p) Sea H una tabla hash de tamaño N=5. Tras insertar la secuencia de datos de entrada E = [33, 55, 82, 16], aplicando una sistema de gestión de colisiones lineal, se decide aumentar el tamaño de la tabla estableciendo el valor de N igual a 11. A partir de la tabla con N=5, obtener, paso a paso e indicando en todo momento el proceso de razonamiento, la distribución de los datos en la nueva tabla. ¿Qué nombre recibe este proceso y cuál sería el coste asociado?

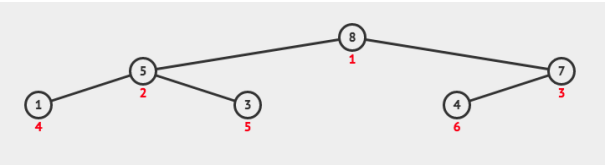

Figura 01. Heap de máximos (H)

PRÁCTICA (7p)

3. (2.5p) Basándonos en la Parte 02 de la Práctica 03, en el **Anexo** se puede ver una definición alternativa (y más simplificada) de la clase **GestionArticulos**. Como se 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. En este ejercicio se propone implementar el método *getAutoresPalabra(String)*. Este método, que hará uso únicamente de las variables locales *result* y *freq* (observad detenidamente la implementación parcial que se propone en el **Anexo**), devolverá un mapa formado por el conjunto de pares *(autor, frecuencia)* asociado con la *palabra* que se especifica como parámetro de entrada. De forma resumida, se puede decir que el valor *frecuencia* indica el número de veces que un determinado *autor* ha utilizado en sus artículos la *palabra* especificada como parámetro de entrada. En caso de necesitar declarar alguna variable local adicional, deberá justificarlo debidamente.

   Por ejemplo. Supongamos que se insertan los siguientes datos:

```
GestionArticulos gestion = new GestionArticulos();
gestion.add("autor01", "articulo01_01", "w01", "w02", "w04");
gestion.add("autor02", "articulo02_01", "w01", "w02", "w03");
gestion.add("autor03", "articulo03_01", "w03", "w03", "w03", “w05”);
gestion.add("autor01", "articulo01_02", "w01", "w02", "w04", "w01");
gestion.add("autor02", "articulo02_02", "w01", "w02", "w03", "w01");
gestion.add("autor01", "articulo01_03", "w04", "w02", "w01");
```

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

```
{autor01={articulo01_01=[w01, w02, w04], articulo01_02=[w01, w02, w04, w01], articulo01_03=[w04, w02, w01]},
 autor02={articulo02_01=[w01, w02, w03], articulo02_02=[w01, w02, w03, w01]},
 autor03={articulo03_01=[w03, w03, w03, w05]}}
```

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

```
gestion.getAutoresPalabra("w01") → {autor01=4, autor02=3, autor03=0}
        //autor01 ha utilizado la palabra w01 4 veces
        //autor02 ha utilizado la palabra w01 3 veces
        //autor03 no ha hecho uso de la palabra w01
gestion.getAutoresPalabra("w03") → {autor01=0, autor02=2, autor03=3}
gestion.getAutoresPalabra("w05") → {autor01=0, autor02=0, autor03=1}
gestion.getAutoresPalabra("w23") → {autor01=0, autor02=0, autor03=0}
```

4. (2.5p) Basándonos en la Parte 03 de la Práctica 03, en el **Anexo** se puede 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**). Hay que tener 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. En este ejercicio se propone implementar el método *toStringOrdered()*. Este método, que hará uso únicamente de las variables locales *valueAux* y *result* (observad detenidamente la implementación parcial que se propone en el **Anexo**), devolverá una cadena de caracteres mostrando un conjunto ordenado formado por todos los pares (*palabra*, *países*) donde, para cada *palabra* distinta existente en la estructura de datos principal, se muestran los *países* desde los que se han enviado mensajes incluyendo dicha *palabra* (independientemente del hashtag utilizado). Se puede observar en el siguiente ejemplo los criterios de orden establecidos (ascendente según *palabra* y *país*). En caso de necesitar declarar alguna variable local adicional, deberá justificarlo debidamente.

   Por ejemplo. Supongamos que se insertan los siguientes datos:

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

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

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

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

```
gestion.toStringOrdered() → {w01=[pais01, pais02], w02=[pais01, pais03], w03=[pais02, pais03], w04=[pais01]}
```

5. (2p) Basándonos en la Práctica 01, presentamos una implementación parcial de la clase **Device** especificada en el **Anexo**. Como se puede observar, esta clase encapsula una estructura de datos lineal tipo lista enlazada, llamada *words*, en la que se almacenan las palabras que un determinado usuario ha enviado haciendo uso de un determinado dispositivo. Si se observa detenidamente la propuesta de implementación del método *sendMessage(String)* se puede comprobar que **no se permite que existan palabras repetidas** dentro de la estructura: esta es una restricción fundamental que **siempre** debe cumplirse. Una vez analizado dicho método vamos a centrarnos en el método *substitute(***String, String***)*. Este método ha sido implementado con la idea de que si ejecutamos el método **main()** que se especifica a continuación, se debería obtener la salida que se especifica en la Tabla 01.

```
public static void main(String[]args) {
        Device dev = new Device("iPhone"); //Suponemos que se le asigna id=1
        dev.sendMessage("Hoy he estado en la playa");
        dev.sendMessage("Hoy he estado en la casa");

        System.out.println(dev.toString());  //Línea 01
        System.out.println(dev.substitute("casa", "playa")); //Línea 02
        System.out.println(dev.toString()); //Línea 03
        System.out.println(dev.substitute("casa", null)); //Línea 04
        System.out.println(dev.toString()); //Línea 05
        System.out.println(dev.substitute("playa", "casa")); //Línea 06
        System.out.println(dev.toString()); //Línea 07
}
```

1.- iphone --> [hoy, he, estado, en, la, playa, casa]	Línea 01 → método toString()
false	Línea 02 → Como la palabra *playa* ya existe, se ignora la acción y se devuelve false.
1.- iphone --> [hoy, he, estado, en, la, playa, casa]	Línea 03 → método toString()
true	Línea 04 → Se elimina la palabra *casa (true).*
1.- iphone --> [hoy, he, estado, en, la, playa]	Línea 05 → método toString()
true	Línea 06 → Se sustituye la palabra *playa* por la palabra *casa (true).*
1.- iphone --> [hoy, he, estado, en, la, casa]	Línea 07 → método toString()

Tabla 01 - Salida esperada.

Sin embargo, si partiendo de este ejemplo hacemos un seguimiento exhaustivo del método *substitute()* se puede comprobar que la salida real no es igual que la salida esperada. (1) ¿En qué consiste el error? (2) Como ejercicio se propone, además, una implementación alternativa del método *substitute()* tal que la salida real sea, exactamente, igual que la salida esperada (según Tabla 01).

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> getAutoresPalabra(String palabra){
                TreeMap<String, Integer> result = new TreeMap<>();

                int freq=0;
                 //…
                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>> result = new TreeMap<>();
             TreeSet<String> valueAux = null;
             //…
             return result.toString();
        }
```

```
        @Override
        public String toString() {
                return this.datos.toString();
        }
}
```

```
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 + ">";
   }
}
```

```
public class Device implements Iterable<String>{
        private static int numDevices=0;
        private String name;
        private int id;
        protected LinkedList<String> words;

        public Device() {...}
        public Device(String name) {...}
        public static void inicializaNumDevices() {...}
        public int getId() {...}
        public void clear() {...}
        public void sendMessage(String msg) {
                if (msg == null) return;
                for (String word : msg.toLowerCase().split(" ")) {
                        if (word.isEmpty()) continue;
                        if (words.contains(word)) continue; //no se permiten palabras repetidas
                        words.add(word);
                }
        }
        public boolean substitute(String word1, String word2) {
                int pos = this.words.indexOf(word1.toLowerCase());
                if (pos == -1) return false;
                if (word2 == null)
                        this.words.remove(pos);
                else
                        this.words.set(pos, word2);
                return true;
        }
        @Override
        public String toString() {
                return this.id + ".- " + this.name + " --> " + this.words.toString();
        }
        public boolean contains(String word) {...}
        @Override
        public boolean equals(Object o) {...}
        @Override
        public Iterator<String> iterator() {…}
}
```