ESTRUCTURAS DE DATOS Y ALGORITMOS I
Grado en Ingeniería Informática
Convocatoria Extraordinaria - Febrero 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)

1. (1p) Clasificación de las estructuras de datos según los parámetros: (1) tipo; (2) organización; (3) gestión de memoria; y (4) soporte de almacenamiento.

2. (1p) Define: (1) Árbol binario lleno; (2) Árbol binario completo; (3) Árbol binario equilibrado; y (4) Árbol binario bien equilibrado. Indica un ejemplo de cada tipo de árbol.

PRÁCTICA (8p)

3. (3p) Basándonos en la Parte 02 de la Práctica 02, en el **Anexo** podemos ver una definición alternativa (y más simplificada) de la clase Biblioteca. Como puedes observar, en esta versión eliminamos las clases Usuario y Libro, ya que en ambos casos solo nos interesa su identificador (clave). Así pues, en nuestra clase Biblioteca, además de un identificador (*bibliotecaID*), tendremos dos colecciones: *prestamos* e *historicoPrestamos*. En la colección *prestamos*, árbol de **pares** *(libroID, usuarioID)*, vamos a tener un registro de todos los préstamos en curso. Por otra parte, en *historicoPrestamos* tendremos la colección de **pares** *(usuarioID, libros_prestados)*, donde *libros_prestados* es un contenedor de tipo *ArrayList<T>* en el que están almacenados todos los identificadores de libros que ha prestado el usuario con identificador *usuarioID*. Hay que tener en cuenta que un usuario puede tomar prestado un determinado libro tantas veces como desee (de ahí la necesidad de hacer uso de una colección que permita elementos repetidos). A partir de la estructura de datos *historicoPrestamos*, vamos a implementar el método *resume()* (ver **Anexo**), cuya funcionalidad consiste en devolver una colección formada por los pares *(usuarioID, libros_frecuencia)* donde, a su vez, *libros_frecuencia* es una colección de pares *(libroID, frecuencia)*, indicando el número de veces que *usuarioID* ha tomado prestado el libro con identificador *libroID*.

   Por ejemplo. Supongamos que realizamos las siguientes operaciones:

```
Biblioteca biblio = new Biblioteca("biblio");
biblio.prestarLibro("usuario01", "libro01");
biblio.prestarLibro("usuario01", "libro02");
biblio.prestarLibro("usuario02", "libro03");
biblio.prestarLibro("usuario03", "libro04");
biblio.devolverLibro("usuario01", "libro01");
biblio.devolverLibro("usuario01", "libro02");
biblio.devolverLibro("usuario03", "libro04");
biblio.prestarLibro("usuario01", "libro01");
biblio.prestarLibro("usuario01", "libro02");
biblio.devolverLibro("usuario01", "libro02");
biblio.devolverLibro("usuario01", "libro01");
biblio.prestarLibro("usuario02", "libro02");
biblio.prestarLibro("usuario03", "libro01");
biblio.devolverLibro("usuario03", "libro01");
biblio.prestarLibro("usuario03", "libro01");
```

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

```
biblio -> [usuario01 <[libro01, libro02, libro01, libro02]>,
           usuario02 <[libro03, libro02]>,
           usuario03 <[libro04, libro01, libro01]>]
```

   En este caso, la salida por Consola del método resume().toString() debe ser:

```
{usuario01={libro01=2, libro02=2}, usuario02={libro02=1, libro03=1}, usuario03={libro01=2, libro04=1}}
```

4. (3p) Vamos a observar con detenimiento los detalles de las clase MySimpleBSTree<T> especificada en el **Anexo**. En particular, vamos a analizar la implementación pública del método *toStringLevel(int* levelMin*)* que devuelve la secuencia ordenada (según criterio de orden natural **descendente**, de mayor a menor) de los elementos del árbol situados en un nivel **mayor o igual** que el valor del parámetro definido por el usuario levelMin. Partiendo de la implementación del método público, vamos a implementar la versión privada teniendo en cuenta la restricción especificada en el **Anexo**.

   Veamos un ejemplo sencillo:

```
MySimpleBSTree<Integer> arbol = new MySimpleBSTree<Integer>();

arbol.add(10);
arbol.add(6);
arbol.add(3);
arbol.add(15);
arbol.add(20);
arbol.add(17);
arbol.add(23);
```

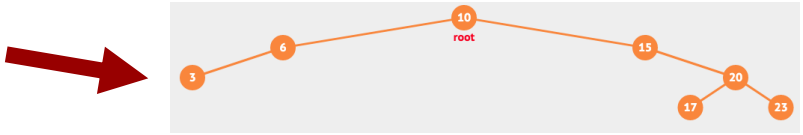

Figura 1. Árbol de ejemplo (asumimos que root está en el nivel **0**)

   A continuación se muestran las salida en Consola estableciendo diferentes valores de levelMin:

```
arbol.toStringLevel(0).toString() → [23, 20, 17, 15, 10, 6, 3]
arbol.toStringLevel(1).toString() → [23, 20, 17, 15, 6, 3]
arbol.toStringLevel(2).toString() → [23, 20, 17, 3]
arbol.toStringLevel(3).toString() → [23, 17]
arbol.toStringLevel(4).toString() → []
```

   Para comprobar la corrección del método vamos a hacer un **seguimiento exhaustivo** (tal y como se ha estudiado en clase) partiendo del árbol de la Figura 1 con levelMin = 2.

5. (2p) Basándonos en la Práctica 04, en el **Anexo** podemos ver la declaración simplificada de un grafo que denominamos MySimpleGraph. Vamos a estudiar con detalle el método add(v$_i$, v$_j$, w), el cual devuelve **true** o **false** dependiendo de si ya existía (o no) la relación que estamos insertando ($v_i \rightarrow^w v_j$). Haciendo uso del método add(), vamos a implementar el método *invierte()* (ver **Anexo**), el cual genera un nuevo grafo copiando todas y cada una de las relaciones existentes en el grafo original (*this*), pero invirtiendo el sentido. Es decir, cada relación del grafo original $v_i \rightarrow^w v_j$ dará lugar a la relación $v_j \rightarrow^w v_i$ en el grafo resultante.

   Por ejemplo, supongamos que insertamos los siguiente datos en un grafo simple:

```
MySimpleGraph<Integer> grafo = new MySimpleGraph<>();
grafo.add(1, 2, 1);
grafo.add(1, 3, 1);
grafo.add(1, 4, 1);
grafo.add(2, 5, 2);
grafo.add(2, 1, 2);
grafo.add(3, 4, 3);
grafo.add(4, 5, 4);
```

   Si mostramos el grafo por Consola, la salida sería:

```
grafo.toString() → {1={2=1, 3=1, 4=1}, 2={1=2, 5=2}, 3={4=3}, 4={5=4}}
```

   Por otra parte, la salida por Consola del grafo invertido debe ser:

```
grafo.invierte().toString() → {1={2=2}, 2={1=1}, 3={1=1}, 4={1=1, 3=3}, 5={2=2, 4=4}}
```

ANEXO

```
public class Biblioteca {
        private String bibliotecaID; //identificador de biblioteca
        protected MySimpleBSTree<Pair<String, String>> prestamos = new MySimpleBSTree<>();
        protected MySimpleBSTree<Pair<String, ArrayList<String>>> historicoPrestamos = new MySimpleBSTree<>();
        public Biblioteca(String bibliotecaID) {...}
        public void clear() {...}
        public boolean prestarLibro(String usuarioID, String libroID) {...}
        public boolean devolverLibro(String usuarioID, String libroID) {...}
        private void guardarHistorico(String libroID, String usuarioID) {...}
        public TreeMap<String, TreeMap<String, Integer>> resume() {
            //...
       }
        @Override
        public String toString() {
                return this.bibliotecaID + " -> " + this.historicoPrestamos.toString();
        }
}
```

```
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 = 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 boolean remove(T item) {...}
    private BSTNode<T> remove(BSTNode<T> current, T item) {...}
    public T find(T item) {...}
    private T find(BSTNode<T> current, T item) {...}
   @Override public String toString() {.. .}
    private String toString(BSTNode<T> current) {...}
```

```
    @Override public Iterator<T> iterator() {...}

    public ArrayList<String> toStringLevel(int levelMin) {
          ArrayList<String> result = new ArrayList<>();
            if (this.root != null) toStringLevel(this.root, result, levelMin, 0);
            return result;
    }
    private void toStringLevel(BSTNode<T> current, ArrayList<String> arr, int levelMin, int levelCurrent) {
            //3 líneas
          //…
    }
}
```

```
public class MySimpleGraph<V extends Comparable<V>> {

            private TreeMap<V, TreeMap<V, Integer>> datos = new TreeMap<>();

            public boolean add (V origen, V destino, Integer peso) {
                        if (origen.equals(destino)) return false;
                        TreeMap<V, Integer> vecinos = this.datos.get(origen);
                        if (vecinos == null) this.datos.put(origen, vecinos = new TreeMap<>());
                        return vecinos.put(destino, peso) == null;
            }

            public MySimpleGraph<V> invierte(){
                        MySimpleGraph<V> result = new MySimpleGraph<>();
                      //…
                        return result;
            }

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