ESTRUCTURAS DE DATOS Y ALGORITMOS I

Grado en Ingeniería Informática

Convocatoria Ordinaria - Enero 2024

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. (0.5p) Sea G un grafo dirigido con 8 nodos, numerados desde el 0 hasta el 7. Tomando como punto de partida el vértice 0, ejecutamos el algoritmo de Dijkstra obteniendo como resultado la tabla que se muestra en la Figura 1. Obtener, de forma razonada, todos los caminos mínimos (y sus costes) que unen el vértice 0 con el resto.

2. (0.5p) Sea T el árbol binario de búsqueda bien equilibrado (tipo AVL) que se muestra en la Figura 2. Razone qué elemento hay que eliminar para que la restauración de la condición de equilibrio se logre tras una rotación doble tipo ID.

3. (0.5p) Sea T el heap de máximos (montículo) que se muestra en la Figura 3. Indique, razonadamente, cómo quedaría el árbol tras insertar el elemento 99 y, a continuación, realizar una operación extraer() (eliminando del árbol el elemento de máxima prioridad).

4. (0.5p) 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 y 25, utilizando como método de resolución de colisiones la técnica de doble hash, siendo la función de dispersión secundaria $H_2(x) = 7 - x \% 7$.

Vértice	Desde	Coste
0	0	0
1	0	3
2	0	5
3	0	3
4	2	12
5	1	5
6	1	6
7	5	10

Figura 1 - Dijkstra

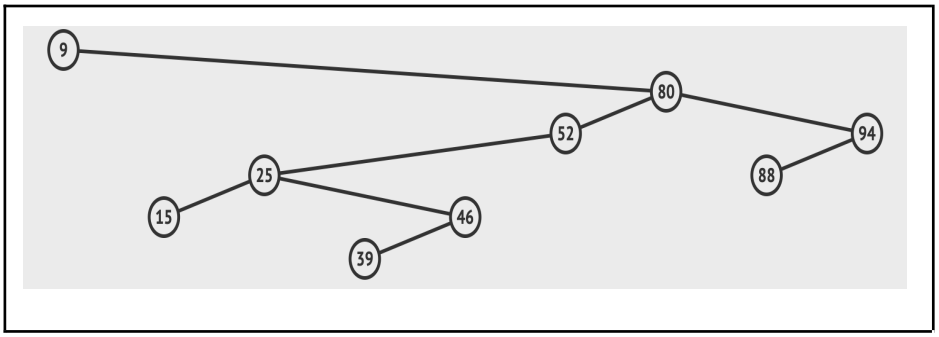

Figura 2 - Árbol Binario de Búsqueda

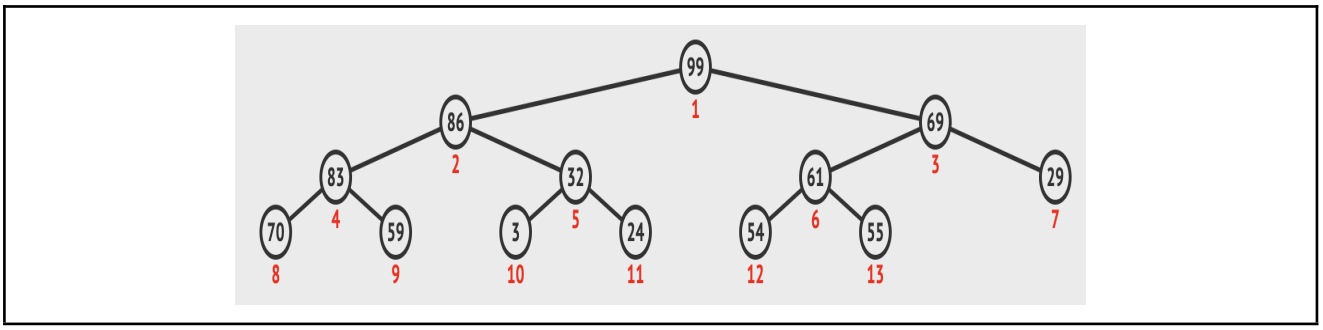

Figura 3 - Heap de máximos

PRÁCTICA (8p)

1. (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 se 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 ***reduce(String autorId)***. Este método devolverá un mapa formado por los pares (*frecuencia*, *conjunto_palabras*), donde *conjunto_palabras* contendrá todas las palabras, con el valor de *frecuencia* indicada por la clave, pertenecientes al conjunto global de artículos del autor especificado como parámetro de entrada (*autorID*).

   Por ejemplo. Supongamos que insertamos los siguientes datos:

```
GestionArticulos gestion = new GestionArticulos();
gestion.add("autor01", "articulo01", "w01", "w02", "w03");
gestion.add("autor01", "articulo02", "w03", "w02", "w01");
gestion.add("autor01", "articulo03", "w01", "w02", "w04", "w01");
gestion.add("autor02", "articulo04", "w01", "w02", "w03");
gestion.add("autor02", "articulo05", "w01", "w02", "w03", "w01");
gestion.add("autor03", "articulo06", "w03", "w03", "w03");
```

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

```
{autor01={articulo01={w01=1, w02=1, w03=1}, articulo02={w01=1, w02=1, w03=1}, articulo03={w01=2, w02=1, w04=1}},
 autor02={articulo04={w01=1, w02=1, w03=1}, articulo05={w01=2, w02=1, w03=1}},
 autor03={articulo06={w03=3}}}
```

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

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

   Sintetice, de forma razonada, la lógica de la implementación propuesta a partir del modelo gráfico de la estructura principal, **TreeMap<String, TreeMap<String, TreeMap<String, Integer>>> datos**, teniendo en cuenta los datos de ejemplo proporcionados en el enunciado. Hay que tener en cuenta las restricciones de uso de métodos especificada en la implementación propuesta.

2. (3p) Basándonos en la Parte03 de la Práctica 03, en el Anexo podemos ver una definición alternativa de las clases Node<T> y NodeCollection<T> que aquí llamamos, directamente, ***Nodo<T>*** y ***Grafo<T>***. Una vez analizadas las clases, se va a: (1) proponer una representación gráfica de la estructura de datos **TreeMap<Nodo<T>, TreeMap<Nodo<T>, Integer>> data** (modelo gráfico) teniendo en cuenta los datos de ejemplo especificados en el método ***main()***; (2) Tras la ejecución del método ***main()***, detalle, razonadamente, cuál sería la salida por Consola (***System.out.println(grafo.toString()***); y (3) Determine un contexto que dé significado a los datos del ejemplo (teniendo en cuenta los datos del ejemplo, hay que proponer un problema que se adecúe a la estructura).

3. (2p) Basándonos en Ejercicio 1 de la parte práctica de este examen, y teniendo en cuenta la implementación realizada en la Parte02 de la Práctica 02, en el Anexo podemos ver una definición alternativa de la clase ***GestionArticulos*** a la que llamamos ***GestionArticulosPar***. Vamos a implementar el método ***converter()*** (prestando especial atención al criterio de orden establecido) de manera que la ejecución del método *main()* muestre por Consola la cadena *"¡¡¡OK!!!"*.

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<Integer, TreeSet<String>> reduce(String autorID){
                TreeMap<Integer, TreeSet<String>> result = new TreeMap<>();
                //… Prohibido hacer uso del método containsKey() (solo V ← get(K) y V_old ← put(K,V))
                //… Prohibido hacer uso del método keySet() (solo entrySet() y values())
                return result;
        }
        @Override
        public String toString() {
                return this.datos.toString();
        }
}
```

```
public class Nodo<T extends Comparable<T>> implements Comparable<Nodo<T>>{
        private ArrayList<T> componentes;
        @SafeVarargs
        public Nodo(T...componentes) {
                this.componentes = new ArrayList<>(List.of(componentes));
        }
        @Override
        public String toString() {
                return this.componentes.toString();
        }
        @Override
        public int compareTo(Nodo<T> otra) {
                int lim1 = this.componentes.size();
                int lim2 = otra.componentes.size();
                for (int i=0; i<Math.min(lim1,  lim2); i++) {
                        int cmp = this.componentes.get(i).compareTo(otra.componentes.get(i));
                        if (cmp != 0) return cmp;
                }
                return lim1-lim2;
        }
}
```

```
public class Grafo<T extends Comparable<T>> {
        private TreeMap<Nodo<T>, TreeMap<Nodo<T>, Integer>> data = new TreeMap<>();
        public void add(Nodo<T> origen, Nodo<T> destino) {
                if (origen.equals(destino)) return;
                TreeMap<Nodo<T>, Integer> value = this.data.get(origen);
                if (value == null) {
                        this.data.put(origen, value = new TreeMap<>());
                }
                Integer cont = value.get(destino);
                value.put(destino, cont == null ? 1 : cont+1);
        }
        @Override
        public String toString() {
                String result = "";
                for (Entry<Nodo<T>, TreeMap<Nodo<T>, Integer>> par: this.data.entrySet()) {
                        result += par.getKey().toString() + " → " + par.getValue().toString() + "\n";
                }
                return result;
        }
    public static void main(String[]args) {
                Grafo<String> grafo = new Grafo<>();
                Nodo<String> nodo1 = new Nodo<>("españa", "italia", "suiza");
                Nodo<String> nodo2 = new Nodo<>("argentina");
```

```
                Nodo<String> nodo3 = new Nodo<>("ghana", "angola", "argelia");
                Nodo<String> nodo4 = new Nodo<>("india", "indonesia", "iran");

                grafo.add(nodo1, nodo2);
                grafo.add(nodo1, nodo2);
                grafo.add(nodo1, nodo3);
                grafo.add(nodo2, nodo3);
                grafo.add(nodo3, nodo2);
                grafo.add(nodo1, nodo2);
                grafo.add(nodo1, nodo4);
                grafo.add(nodo4, nodo3);
                grafo.add(nodo3, nodo2);

                System.out.println(grafo.toString()); //¿?
        }
}
```

```
public class GestionArticulosPar {
        private final AVLTree<Par<String, AVLTree<Par<String, AVLTree<String>>>>> data;
        public GestionArticulosPar() {
                this.data = new AVLTree<>();
        }
        public void add(String clave01, String clave02, String...palabras) {...}
        public ArrayList<Par<String, ArrayList<Par<String, ArrayList<String>>>>> converter() {
                ArrayList<Par<String, ArrayList<Par<String, ArrayList<String>>>>> result = new ArrayList<>();
                //…
                return result;
        }
        @Override
        public String toString() {
                return this.data.toString();
        }
        public static void main(String[] args) {
                GestionArticulosPar gestion = new GestionArticulosPar();
                gestion.add("autor01", "articulo01", "w01", "w02", "w03");
                gestion.add("autor01", "articulo02", "w03", "w02", "w01");
                gestion.add("autor01", "articulo03", "w01", "w02", "w04", "w01");
                gestion.add("autor02", "articulo04", "w01", "w02", "w03");
                gestion.add("autor02", "articulo05", "w01", "w02", "w03", "w01");
                gestion.add("autor03", "articulo06", "w03", "w03", "w03");
                System.out.println(gestion.toString().equals(gestion.converter().toString()) ? "¡¡¡OK!!!" : "¡¡¡Error!!!");
        }
}
```

```
public class Par <K extends Comparable<K>,V> implements Comparable<Par<K,V>>{
        private final K key;
        private V value;
        public Par(K key, V value) {
                this.key = key;
                this.value = value;
        }
        public K getKey() {...}
        public V getValue() {...}
        public V setValue(V value) {...}
        @Override public String toString() {...} // Formato: clave <valor>
        @Overridepublic boolean equals(Object o) {...}
        @Override public int compareTo(Par<K,V> other) {...}
}
```