ESTRUCTURAS DE DATOS Y ALGORITMOS I

Grado en Ingeniería Informática

Convocatoria Extraordinaria - Febrero 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 5 nodos, numerados desde el 0 hasta el 4. Durante la ejecución del algoritmo de Floyd para la obtención de caminos mínimos, en un instante de tiempo (k=0) se obtienen, entre otros, los componentes de la matriz de costes mínimos $D_0[0,1] = 4$, $D_0[0,2] = 11$, $D_0[0,3] = \infty$, $D_0[0,4] = \infty$, $D_0[1,3] = 6$ y $D_0[1,4] = 2$. Teniendo en cuenta esta información, obtener (razonadamente) los componentes $D_1[0,3]$ y $D_1[0,4]$.

2. (0.5p) Sea T el árbol binario de búsqueda bien equilibrado (tipo AVL) que se muestra en la Figura 1. Obtener, justificadamente, el árbol AVL resultante tras eliminar el nodo con valor 75. ¿Qué tipo de rotación se ha tenido que aplicar para restaurar la condición de equilibrio?

3. (0.5p) Sea T el heap de máximos (montículo) que se muestra en la Figura 2. Indique, razonadamente, cómo quedaría el árbol tras insertar los elementos: (1) 26; y (2) 90.

4. (0,5p) En la Figura 3 podemos observar una tabla hash obtenida tras la inserción de la secuencia de elementos [25, 39, 7, 14, 2] haciendo uso de la técnica de gestión de colisiones conocida como prueba lineal. (1) A partir de esta tabla vamos a reubicar todos los elementos en una nueva tabla hash, esta vez con tamaño n = 11 (obviamente, haciendo uso de la misma técnica de gestión de colisiones). (2) ¿Cuál es el nombre que recibe esta operación? (3) ¿Cuál es su coste?

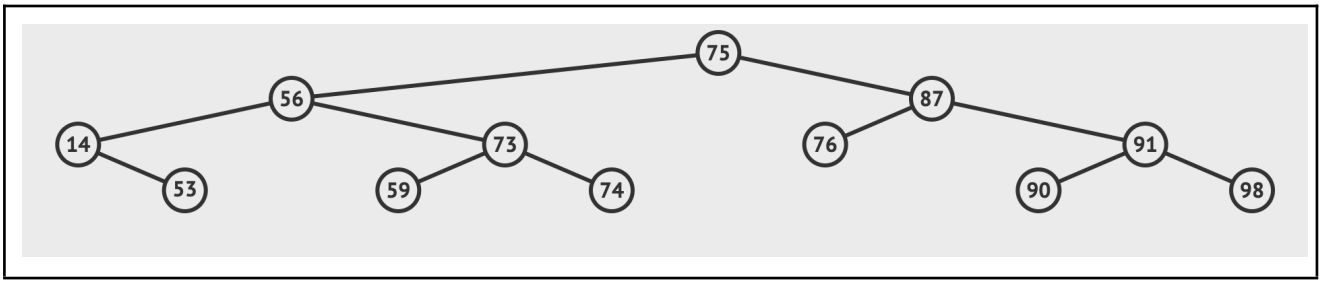

Figura 1 - Árbol AVL

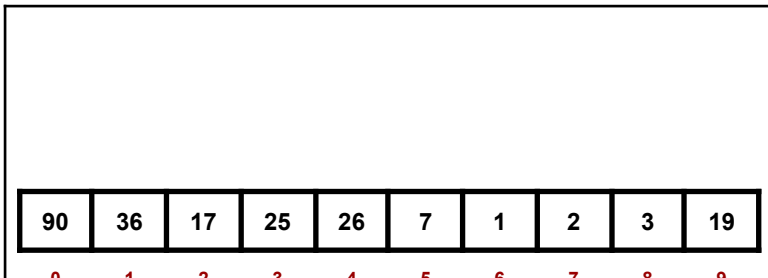

Figura 2 - Heap de máximos

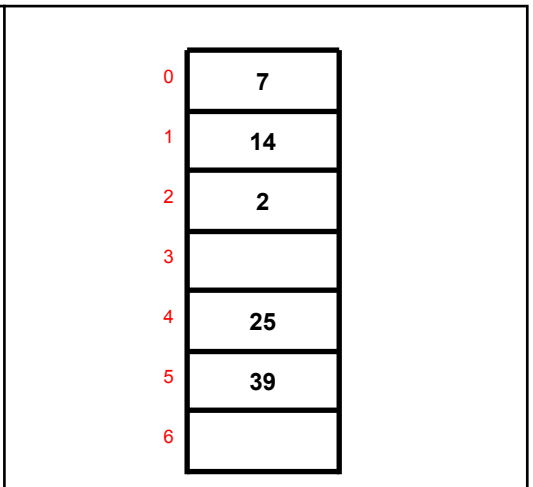

Figura 3 - Tabla hash

PRÁCTICA (8p)

1. (3p) Basándonos en la Parte 02 de la Práctica 03, y teniendo en cuenta la implementación realizada en las Partes 01 y 02 de la Práctica02, en el **Anexo** podemos ver una definición alternativa (y más simplificada) de la clase *GestionRepositorios* a la que llamaremos ***GestionArticulosReduce***. Como se puede observar, en esta versión hacemos uso de una estructura de datos asociativa anidada, llamada *data*, 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). Teniendo en cuenta lo anteriormente descrito, vamos a: (1) dibujar la estructura de datos teniendo en cuenta los datos de ejemplo que se detallan en el método *main()*; y (2) implementar (**razonadamente**) los métodos ***reduceAL()*** y ***reduceAVL()*** cuya lógica se puede deducir observando la salida por Consola que se muestra a continuación::

   gestion.toString() → {autor01={articulo01={w01=2, w02=4}, articulo02={w01=2, w02=1, w04=1}},
   autor02={articulo03={w01=7}},
   autor03={articulo04={w03=3}}}

   gestion.reduceAL() → [w01 <[articulo01, articulo02, articulo03]>, w02 <[articulo01, articulo02]>,
   w03 <[articulo04]>, w04 <[articulo02]>]
   gestion.reduceAVL() → [w01 <[articulo01, articulo02, articulo03]>, w02 <[articulo01, articulo02]>,
   w03 <[articulo04]>, w04 <[articulo02]>]

   Como se puede observar, la salida de ambos métodos es exactamente la misma. En ambos casos se devuelve una colección **ordenada** de pares cuya clave es una palabra y, como valor, un conjunto de identificadores de artículos en los que aparece dicha palabra (independientemente de la frecuencia de las palabras y de los autores de los artículos). La única diferencia entre ambos métodos es la colección principal utilizada, que el primer caso se trata de un ***ArrayList<T>***, mientras que en el segundo caso se trata de la estructura arborescente ***AVLTree<T>***.

2. (2p) 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** 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 indicando expresamente cuál sería la semántica del peso de las relaciones entre los distintos nodos que forman parte del ejemplo.

3. (3p) Basándonos en Ejercicio 1 de la parte práctica de este examen, en el Anexo podemos ver una definición alternativa de la clase ***GestionArticulosReduce*** a la que llamamos ***GestionArticulosConverter***. En este caso, como colección de pares se hace uso de estructuras arborescentes ***AVLTree<Par<K,V>>*** y, cuestión importante, los artículos no se representan con un conjunto de pares *(articuloId, palabrasFrecuencia)*, sino que las palabras que forman un artículo se van a almacenar en una estructura lineal tipo *ArrayList<T>*. A partir de esta nueva definición de estructura, llamada *data*, vamos a: (1) dibujar la estructura de datos teniendo en cuenta los datos de ejemplo que se detallan en el método *main()*; y (2) implementar (**razonadamente**) el método ***converter()*** que, simplemente, vuelca el contenido de *data* en la estructura asociativa **TreeMap<String, TreeMap<String, TreeMap<String, Integer>>>**. La ejecución del método *main()* debe mostrar por Consola:

   gestion.toString() →[autor01 <[articulo01 <[w01, w02, w01, w02, w02, w02]>, articulo02 <[w01, w02, w04, w01]>]>,
   autor02 <[articulo03 <[w01, w01, w01, w01, w01, w01, w01]>]>,
   autor03 <[articulo04 <[w03, w03, w03]>]>]

   gestion.converter() → {autor01={articulo01={w01=2, w02=4}, articulo02={w01=2, w02=1, w04=1}},
   autor02={articulo03={w01=7}},
   autor03={articulo04={w03=3}}}

ANEXO

```java
public class GestionArticulosReduce {
	private final TreeMap<String, TreeMap<String, TreeMap<String, Integer>>> data = new TreeMap<>();

	public void add(String autorId, String articuloId, String...palabras) {...}

	public ArrayList<Par<String, TreeSet<String>>> reduceAL() {
		ArrayList<Par<String, TreeSet<String>>> result = new ArrayList<>();
		//…
		return result;
	}

	public AVLTree<Par<String, TreeSet<String>>> reduceAVL() {
		AVLTree<Par<String, TreeSet<String>>> result = new AVLTree<>();
		//…
		return result;
	}


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

	public static void main(String[]args) {
		GestionArticulosReduce gestion = new GestionArticulosReduce();

		gestion.add("autor01", "articulo01", "w01", "w02", "w01");
		gestion.add("autor01", "articulo01", "w02", "w02", "w02");
		gestion.add("autor01", "articulo02", "w01", "w02", "w04", "w01");
		gestion.add("autor02", "articulo03", "w01", "w01", "w01");
		gestion.add("autor02", "articulo03", "w01", "w01", "w01", "w01");
		gestion.add("autor03", "articulo04", "w03", "w03", "w03");

		System.out.println(gestion);
		System.out.println(gestion.reduceAL());
		System.out.println(gestion.reduceAVL());
	}
}
```

```java
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;
	public Grafo() {
		this.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 ? 0 : cont) + 1);
	}

	@Override
	public String toString() {
		String result = "";
		for (Entry<Nodo<T>, TreeMap<Nodo<T>, Integer>> parMain: this.data.entrySet()) {
			result += parMain.getKey().toString() + ":\n";
			for(Entry<Nodo<T>, Integer> parSecond: parMain.getValue().entrySet()) {
				result += "\t" + parSecond.getKey().toString() + " <" +
					Format.formatInt(parSecond.getValue(), 3) + ">" + "\n";
			}
		}
		return result;
	}


	public static void main(String[]args) {
		Grafo<String> grafo = new Grafo<>();
		Nodo<String> nodo1 = new Nodo<>("carmen", "juan", "gregory");
		Nodo<String> nodo2 = new Nodo<>("carmen", "juan", "emilio");
		Nodo<String> nodo3 = new Nodo<>("andres", "antonio", "amelia", "jeremias");
		Nodo<String> nodo4 = new Nodo<>("eva", "adela", "antonio", "ivan");
		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 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) {...}
}
```

```
public class GestionArticulosConverter {
        AVLTree<Par<String, AVLTree<Par<String, ArrayList<String>>>>> data = new AVLTree<>();

        public void add(String clave01, String clave02, String...items) {...}

        public TreeMap<String, TreeMap<String, TreeMap<String, Integer>>> converter() {
                TreeMap<String, TreeMap<String, TreeMap<String, Integer>>> result = new TreeMap<>();
                TreeMap<String, TreeMap<String, Integer>> articulos;
                TreeMap<String, Integer> palabrasFreq;
                //…
                return result;
        }

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

        public static void main(String[]args) {
                GestionArticulosConverter gestion = new GestionArticulosConverter();
                gestion.add("autor01", "articulo01", "w01", "w02", "w01");
                gestion.add("autor01", "articulo01", "w02", "w02", "w02");
                gestion.add("autor01", "articulo02", "w01", "w02", "w04", "w01");
                gestion.add("autor02", "articulo03", "w01", "w01", "w01");
                gestion.add("autor02", "articulo03", "w01", "w01", "w01", "w01");
                gestion.add("autor03", "articulo04", "w03", "w03", "w03");

                System.out.println(gestion);
                System.out.println(gestion.converter());
        }
}
```