Recuperación Primer Parcial de Estructuras de Datos y Algoritmos (EDA) – 17 de Junio de 2022

Duración: 1h. 30m.

APELLIDOS, NOMBRE	GRUPO

1.- ***(2,25 puntos)*** Implementa un método estático tal que:

- Reciba una **`Pila`** genérica de elementos **`Comparable`**, **p**, y un dato, **x**, del tipo de los datos en **p**.
- Borre de la pila **p** todos los elementos que sean menores estrictos que **x**.
- Los elementos de la pila **p** que sean mayores o iguales a **x** se mantendrán en la pila, conservando las posiciones relativas entre ellos.

La solución debe ser **eficiente**, con coste lineal con la talla de la pila, y sin utilizar ninguna EDA auxiliar.

Ejemplos.

- Si el método se invoca con **x** = 4 y **p** = [3,4,2,5,7,1,2,8], donde 3 es el tope de la pila y 8 es el fondo de la pila, entonces tras ejecutarse el método, **p** = [4,5,7,8].
- Si se invoca con **x** = "bd" y **p** = ["xa","aa","ab","ba","gk","bc","cx","ac","bb"], donde "xa" es el tope de la pila y "bb" es el fondo de la pila, entonces tras ejecutarse el método, **p** = ["xa","gk","cx"].

```
public static <E extends Comparable<E>> void metodo1(Pila<E> p, E x) {
   if (!p.esVacia()) {
      E e = p.desapilar();
      metodo1(p, x);
      if (e.compareTo(x) >= 0) p.apilar(e);
   }
}
```

2.- ***(2,25 puntos)*** Aplicación de la estrategia **Divide y Vencerás (DyV)**.

Sea **v**, un array de enteros positivos, ordenado descendentemente y que puede contener valores repetidos, que representa las citas a las publicaciones de un investigador (número de veces que cada publicación ha sido referenciada desde otras publicaciones).

Se quiere obtener **h**, el índice[1] del investigador cuyas citas están representadas en **v**. El índice **h** vale **x**, si **x** publicaciones han sido citadas más de **x** veces.

Ejemplos:

- Si **v** es [2,2,1,1,1,1,1,1,1,1], h es 2.
- Si **v** es [3,3,3,2,1], h es 3.
- Si **v** es [10,9,8,7,6,3,2,1], h es 5.
- Si **v** es [61,41,40,35,32,32,30,27,23,22,18,17,17,16,15,14,14,12,12], h es 15.

Se dispone del siguiente método lanzadera:

```
public static int metodo2(int[] v) {
   return metodo2 (v, 0, v.length – 1, 0);
}
```

[1] Según la definición de índice **h** en Wikipedia: *"Un investigador tiene un índice **h** si **h** de sus **N** trabajos tienen al menos **h** citas cada uno, y los otros **N-h** trabajos no tienen más de **h** citas cada uno".*

Aplicando **DyV** (y, por tanto, con **coste logarítmico**), se pide implementar el método recursivo, invocado en la anterior lanzadera, que devuelva el índice **h** del investigador cuyas citas están representadas en **v**. El cuarto argumento del método recursivo servirá para pasar el valor de **h** entre las llamadas recursivas.

```
public static int metodo2(int[] v, int ini, int fin, int h) {
   if (ini > fin) return h;
   int m = (ini + fin) / 2;
   if (v[m] >= m + 1) return metodo2(v, m + 1, fin, m + 1);
   else return metodo2(v, ini, m – 1, h);
}
```

3.- ***(2,75 puntos)*** Una empresa tiene en sus sistemas de información una ficha de cada empleado en la que aparece su nombre y su salario anual. Considera que cada empleado se modeliza mediante la clase **Empleado** mostrada a la derecha.

```
class Empleado {
   ...
   public Empleado(String n, double s) { ... }
   public String getNombre() { ... }
   public double getSalario() { ... }
}
```

A principio de cada año, se dispone de una **ListaConPI<Empleado> l1** que contiene dicha información. Y, a final del año, se genera una nueva **ListaConPI<Empleado> l2** con la información salarial actualizada. Sin embargo, **l1** y **l2** no contienen la información de los empleados en el mismo orden. Por ejemplo:

l1 = {(Pepe, 24000), (Ana, 32000), (Juan, 23000), (Paco, 50000), (Jorge, 36000), (Lola, 42000)}

l2 = {(Ana, 31500), (Lola, 43800), (Pepe, 25000), (Jorge, 36300), (Luis, 19000), (Paco, 50000)}

Se quiere obtener una nueva **ListaConPI<Empleado> l3** que contenga el listado de las personas que han modificado su salario, y el incremento que ha tenido (en esta lista **l3**, cada empleado tendrá su incremento como "salario"). Continuando el ejemplo anterior:

l3 = {(Jorge, 300), (Lola, 1800), (Ana, -500), (Pepe, 1000)}

Para obtener la lista **l3** de modo **eficiente** (coste lineal con la suma de las longitudes de las listas) se debe usar un **Map** como EDA auxiliar.

```
public static ListaConPI<Empleado> metodo3(ListaConPI<Empleado> l1,
ListaConPI<Empleado> l2) {
   ListaConPI<Empleado> l3 = new LEGListaConPI<>();
   Map<String, Double> m = new TablaHash<>(l2.talla());
   for (l2.inicio(); !l2.esFin(); l2.siguiente()) {
      Empleado e = l2.recuperar();
      m.insertar(e.getNombre(), e.getSalario());
   }
   for (l1.inicio(); !l1.esFin(); l1.siguiente()) {
      Empleado e = l1.recuperar();
      String s = e.getNombre();
      Double d1 = e.getSalario();
      Double d2 = m.recuperar(s);
      if (d2 != null && d1 != d2) { l3.insertar(new Empleado(s, d2 - d1)) }
   }
   return l3;
}
```

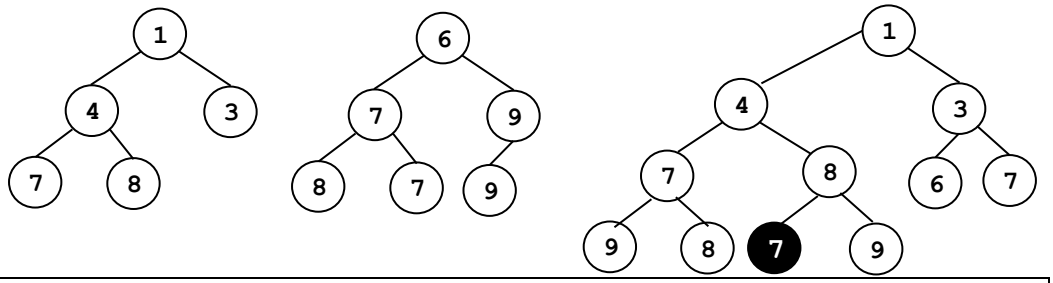
4.- ***(2,75 puntos)*** Implementa, en la clase **`MonticuloBinario`**, un método que reciba otro montículo binario, **`mb`**, y compruebe si los datos de **`mb`** (almacenados en su atributo **`mb.elArray`**) se podrían añadir a los datos de **`this`** (almacenados en **`this.elArray`**) en posiciones consecutivas de su array de forma que **`this`** siguiera siendo un montículo binario (es decir, se preservara la propiedad de orden).

La solución debe ser **eficiente** (coste lineal con tallas de los montículos), y sin utilizar ninguna EDA auxiliar.

Importante. El método debe limitarse a comprobar lo indicado, sin modificar ninguno de los montículos.

Precondición. Para simplificar la solución, suponer que la talla de **`this`** es impar y la talla de **`mb`** es par.

Ejemplo. Si, en la siguiente figura, **`this`** es el montículo en la izquierda y **`mb`** es el montículo en el centro, entonces el método devolverá **`false`** porque, como indica el montículo en la derecha, el quinto elemento de **`mb`** (7) no podría ser hijo del último elemento de **`this`** (8). Si el último elemento de **`this`** fuera 6 o fuera 7, entonces, para el mismo **`mb`**, el método devolvería **`true`**.




```
// Una solución:
public boolean metodo4(MonticuloBinario<E> mb) {
    for (int i = talla / 2 + 1, j = 1; i <= talla; i++, j = j + 2) {
        if (elArray[i].compareTo(mb.elArray[j]) > 0) return false;
        if (elArray[i].compareTo(mb.elArray[j+1]) > 0) return false;
    }
    return true;
}

// Otra solución:
public boolean metodo4(MonticuloBinario<E> mb) {
    for (int i = 1; i <= mb.talla; i++) {
        if (elArray[(talla + i) / 2].compareTo(mb.elArray[i]) > 0) return false;
    }
    return true;
}
```

ANEXO.

Las interfaces **Map, Pila** y **ListaConPI** del paquete **modelos**. La clase **MonticuloBinario** del paquete **jerarquicos**.

```
public interface Map<C, V> {
  V insertar(C c, V v);
  V eliminar(C c);
  V recuperar(C c);
  boolean esVacio();
  int talla();
  ListaConPI<C> claves();
}
public interface Pila<E> {
  void apilar(E e);
  E desapilar();
  E tope();
  boolean esVacia();
}
```

```
public interface ListaConPI<E> {
  void insertar(E e);
  void eliminar();
  void inicio();
  void siguiente();
  void fin();
  E recuperar();
  boolean esFin();
  boolean esVacia();
  int talla();
}
```

```
public class MonticuloBinario<E extends Comparable<E>> implements ColaPrioridad<E> {
  protected E[] elArray;
  protected int talla;
  ...
}
```