Resolución de la Recuperación del Segundo Parcial de EDA (20 de Junio de 2016) - Puntuación: 3

1.- Se quiere sobrescribir el método `insertar` de la clase `MonticuloBinario` para impedir que se duplique `elArray` cuando se haga una nueva inserción con el array completamente lleno (excepto la posición 0 que siempre está libre). En el caso de que `elArray` esté lleno, el nuevo elemento a insertar deberá reemplazar al elemento máximo del Montículo.

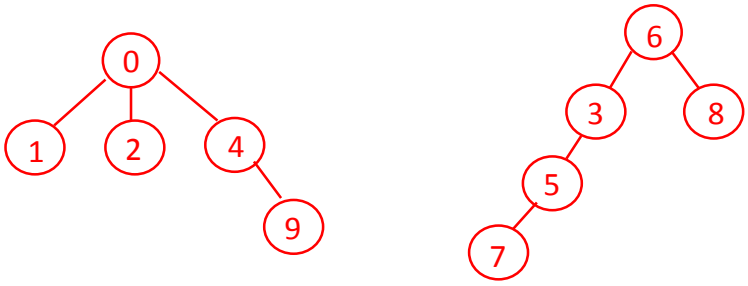
Implementa esta nueva versión del método `insertar` de la forma más eficiente posible. **(1 punto)**

```
public void insertar(E x) {
    // PASO 1: determinar hueco, la posición inicial de inserción de x en elArray:
    // si está lleno, hueco es la posición de un máximo de un Montículo Minimal
    // con Raíz en 1; si no lo está, hueco = ++talla, la posición de inserción
    // por Niveles de x en un AB Completo con Raíz en 1
    int hueco;
    if (talla == elArray.length - 1) {
        // Búsqueda de la posición de un máximo del Montículo a partir de su 1era Hoja
        hueco = talla / 2 + 1;
        for (int i = hueco + 1; i <= talla; i++) {
            if (elArray[hueco].compareTo(elArray[i]) < 0) { hueco = i; }
        }
    }
    else { hueco = ++talla; }
    // PASO 2: determinado hueco, en su caso, reflotar elArray[hueco] hasta
    // restablecer la propiedad de orden del Montículo
    while (hueco > 1 && x.compareTo(elArray[hueco / 2]) < 0) {
        elArray[hueco] = elArray[hueco / 2];
        hueco = hueco / 2;
    }
    elArray[hueco] = x;
}
```

2.- Sea el siguiente *MFSet*:

0	1	2	3	4	5	6	7	8	9
-3	0	0	6	0	3	-4	5	6	4

a) Dibuja el bosque de árboles representado en el *MFSet*: **(0.2 puntos)**



b) Teniendo en cuenta que el *MFSet* implementa la fusión por rango y la compresión de caminos indica como irá evolucionando el *MFSet* tras las siguientes instrucciones: **(0.3 puntos)**

	0	1	2	3	4	5	6	7	8	9
`find(9)`	-3	0	0	6	0	3	-4	5	6	***0***
`merge(2,5)`	***6***	0	0	6	0	***6***	-4	5	6	0
`find(7)`	6	0	0	6	0	6	-4	***6***	6	0

3.- Implementa un método en la clase `ForestMFSet` que devuelva una `ListaConPI<Integer>` con todos los elementos que pertenecen al mismo conjunto que un elemento `x` dado, `x` inclusive. **(0.5 puntos)**

```
public ListaConPI<Integer> mismoConjunto(int x) {
    ListaConPI<Integer> res = new LEGListaConPI<Integer>();
    int conjuntoDeX = find(x);
    for (int i = 0; i < talla; i++) {
        if (find(i) == conjuntoDeX) { res.insertar(i); }
    }
    return res;
}
```

4.- El grado de separación entre dos vértices de un *Grafo* se define como el número mínimo de aristas que los separan en él. Diseña en la clase `Grafo` un método que, dado un vértice `i` (0 <= `i` < `numVertices()`) y un grado de separación `g` >0, devuelva el número de vértices de un *Grafo* que están a un grado de separación `g` de `i`. **(1 punto)**

```
public int verticesSeparacion(int i, int g) {

    // Por definición de grado de separación, realizar un Recorrido BFS para
    // determinar el nº de vértices del Grafo que se encuentran a una distancia
    // mínima sin pesos (o nº de Aristas) g del vértice i
    int res = 0;
    int[] distanciaMinSinPesos = new int[numVertices()];

    // El valor entero -1 representa el INFINITO entero, la distancia o nº de
    // Aristas que separan inicialmente a i del resto de vértices del Grafo
    for (int v = 0; v < numVertices(); v++) { distanciaMinSinPesos[v] = -1; }
    // OJO: la distancia del vértice i a sí mismo es SIEMPRE cero
    distanciaMinSinPesos[i] = 0;
    q = new ArrayCola<Integer>();
    q.encolar(i);
    while (!q.esVacia()) {
        int u = q.desencolar();
        ListaConPI<Adyacente> l = adyacentesDe(u);
        for (l.inicio(); !l.esFin(); l.siguiente()) {
            int w = l.recuperar().getDestino();

            // condición, y garantía, de distancia mínima sin pesos:
            // que w no haya sido alcanzado aún desde i
            if (distanciaMinSinPesos[w] == -1) {
                distanciaMinSinPesos[w] = distanciaMinSinPesos[v] + 1;

                // condición exigida: que w esté a un grado de separación g de i
                if (distanciaMinSinPesos[w] == g) { res++; }
                else { q.encolar(w); }
            }
        }
    }
    return res;
}
```

```
public class MonticuloBinario<E extends Comparable<E>>
    implements ColaPrioridad<E> {

    protected static final int CAPACIDAD_INICIAL = 50;
    protected E[] elArray;
    protected int talla;
    public MonticuloBinario() { … }
    public void insertar(E x) { … }
    private void hundir(int hueco) { … }
    public E eliminarMin() { … }
    public E recuperarMin() { … }
    private void arreglar() { … }
}

public abstract class Grafo {
    protected static final double INFINITO = Double.POSITIVE_INFINITY;
    protected int[] visitados;
    protected int ordenVisita;
    protected Cola<Integer> q;
    …
    public abstract int numVertices();
    public abstract int numAristas();
    public abstract boolean existeArista(int i, int j);
    public abstract double pesoArista(int i, int j);
    public abstract ListaConPI<Adyacente> adyacentesDe(int i);
    public abstract void insertarArista(int i, int j);
    public abstract void insertarArista(int i, int j, double p);
    public String toString() { … }
    …
}
public class Adyacente {
    protected int destino;
    protected double peso;

    public Adyacente(int d, double p) { … }
    public int getDestino() { … }
    public double getPeso() { … }
    public String toString() { … }
}
public interface Cola<E> {
    void encolar(E e);
    E desencolar();
    E primero();
    boolean esVacia();
}

public interface MFSet {
    int find(int x);
    void merge(int x, int y);
}
public class ForestMFSet implements MFSet {
    protected int talla;
    protected int[] elArray;

    public ForestMFSet(int n) { … }
    public int find(int x) { … }
    public void merge(int x, int y) { … }
}
```