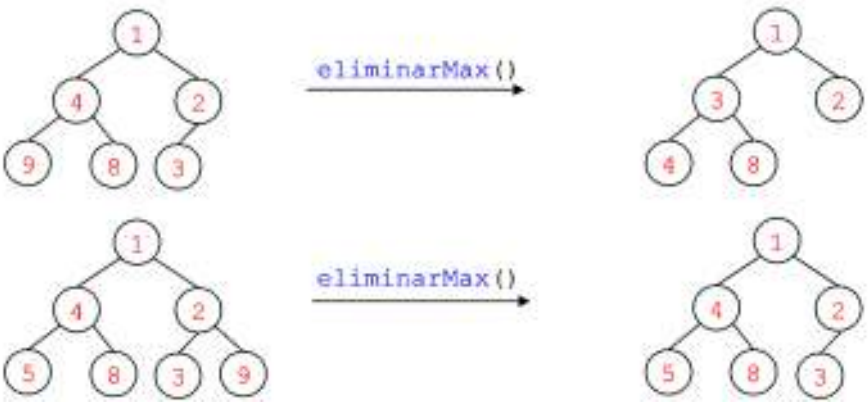
Resolución del Segundo Parcial de EDA (8 de Junio de 2017)

1.- Se quiere diseñar un nuevo método `eliminarMax` en la clase `MonticuloBinario` que elimine y devuelva el máximo de un Montículo Binario (Minimal) con Raíz en 1 y sin elementos repetidos. Para ello, **se pide:** **(4 puntos)**

a) Dibujar el estado de cada uno de los siguientes Montículos tras aplicarle este nuevo método. **(0,5 puntos)**



b) Implementar el método `eliminarMax` iterativamente y de la forma más eficiente posible. **(2.5 puntos)**

```
public E eliminarMax() {
    // PASO 1: obtener la posición del máximo, recorriendo SOLO las hojas del Heap
    int posMax = talla / 2 + 1;
    for (int i = posMax + 1; i <= talla; i++) {
        if (elArray[posMax].compareTo(elArray[i]) < 0) { posMax = i; }
    }
    E max = elArray[posMax];
    // PASO 2: borrar elArray[posMax] de un AB Completo
    elArray[posMax] = elArray[talla--];
    //PASO 3: reflotar elArray[posMax] hasta restablecer la prop. de orden del Heap
    E aux = elArray[posMax];
    while (posMax > 1 && aux.compareTo(elArray[posMax / 2]) < 0) {
        elArray[posMax] = elArray[posMax / 2];
        posMax = posMax / 2;
    }
    elArray[posMax] = aux;

    return max;
}
```

c) Para el método `eliminarMax` diseñado, indicar: **(1 punto)**

Talla del problema, en función de los atributos de la clase: x = `talla`.

¿Existen instancias significativas? Razonar la respuesta, indicando si ha lugar los casos Peor y Mejor y su coste

No: en el primer bucle se realizan, independientemente del caso, `talla`/2 – 1 `compareTo`, mientras que en el segundo se realiza una sola en el Mejor de los Casos, si NO hay que reflotar `elArray[posMax]`, y $\log_2$ `talla` en el Peor, cuando hay que reflotar elArray[posMax] hasta la Raíz; así, en cualquier caso, el coste total del método es proporcional a `talla`/2, i.e. el máximo del número de `compareTo` que se realizan en cada uno de estos dos bucles.

Coste Temporal Asíntótico, $T_{\texttt{eliminarMax}}(x)$, utilizando la notación asintótica (O y Ω o bien Θ):

$$T_{\texttt{eliminarMax}}(x) \in \Theta(x).$$

2.- Se dice que una arista (v, w) de un Grafo Dirigido es una arista Hacia Atrás si llega a un vértice w previamente alcanzado (visitado) durante el recorrido DFS del vértice v. Nótese también que si en un Grafo Dirigido hay algún ciclo es porque en él hay alguna arista Hacia Atrás.

En base a lo anterior, **se pide** implementar en la clase `Grafo` un método que, con el menor coste posible, devuelva el número de aristas Hacia Atrás de un Grafo. Ello supone, al igual que para cualquier otro recorrido DFS de un Grafo, implementar los siguientes dos métodos: **(4 puntos)**

a) Un método público `aristasHA`, que implementa el recorrido DFS de los vértices de un Grafo que calcula el número de aristas Hacia Atrás que este contiene; para ello, usa el método recursivo del siguiente apartado. **(1.5 puntos)**

```
public int aristasHA() {
    int res = 0; visitados = new int[numVertices()];
    for (int v = 0; v < numVertices(); v++) {
        if (visitados[v] == 0) { res += aristasHA(v); }
    }
    return res;
}
```

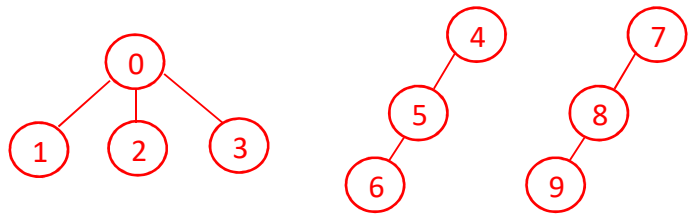
b) Un método recursivo y protegido (*protected*), que devuelve el número de aristas Hacia Atrás que se detectan en el recorrido DFS de un vértice de un Grafo. **(2.5 puntos)**

```
protected int aristasHA(int v) {
    int res = 0; visitados[v] = 1;
    ListaConPI<Adyacente> l = adyacentesDe(v);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        int w = l.recuperar().getDestino();
        if (visitados[w] == 0) { res += aristasHA(w); }
        else if (visitados[w] == 1) { res++; }
    }
    visitados[v] = 2;
    return res;
}
```

3.- Sea la siguiente representación de un MF-Set: **(2 puntos)**

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

a) Dibuja el bosque de árboles que contiene. **(0.5 puntos)**



b) Teniendo en cuenta que se implementa la fusión por rango y la compresión de caminos, indica cómo irá evolucionando dicha representación tras la ejecución de las instrucciones de la siguiente tabla. **(1.5 puntos)**

Nota: al unir dos árboles con la misma altura (o rango), el primer árbol deberá colgar del segundo.

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

```
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;
    …
    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 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) { … }
}
```