

Resolución del Tercer Parcial de EDA (2 de Junio de 2014)

1.- En la clase ABB, **se pide** implementar un método público contarEnRango que, con el menor coste posible, devuelva el número de elementos de un ABB comprendidos en el intervalo $[eI, eS]$, donde eI y eS son dos elementos dados de tipo genérico E tales que eI es menor o igual que eS ; dicho método **solo** puede hacer uso de los atributos de ABB (ver Anexo). **(2.5 puntos)**

```
public int contarEnRango(E eI, E eS) {
    return contarEnRango(eI, eS, this.raiz);
}
private int contarEnRango(E eI, E eS, NodoABB<E> actual) {
    if (actual == null) return 0;
    if (actual.dato.compareTo(eI) < 0) return contarEnRango(eI, eS, actual.der);
    if (actual.dato.compareTo(eS) > 0) return contarEnRango(eI, eS, actual.izq);
    return 1 + contarEnRango(eI, eS, actual.izq) + contarEnRango(eI, eS, actual.der);
}
```

Asumiendo que el ABB está equilibrado, indica y justifica brevemente el coste temporal asintótico del método diseñado. **(0.5 puntos)**

Talla = N , o número de nodos del ABB

Mejor caso: todos los elementos del ABB están fuera del rango (bien son menores que eI o bien son mayores que eS), por lo que solo se recorre una rama del ABB. Por tanto, $T(N) \in \Omega(\log N)$.

Peor caso: todos los elementos del ABB están dentro del rango, por lo que se realizan dos llamadas recursivas en secuencia en cada nodo del ABB. Por tanto, $T(N) \in O(N)$.

2.- En la clase GrafoDirigido, **se pide** implementar un método público aislados que, con el menor coste posible, devuelva el número de vértices aislados (de grado 0) que hay en un Grafo Dirigido; dicho método **solo** puede hacer uso de los atributos de GrafoDirigido (ver Anexo). **(2.5 puntos)**

```
public int aislados() {
    int[] grado = new int[numV];
    for (int v = 0; v < numV; v++) {
        ListaConPI<Adyacente> l = elArray[v];
        grado[v] += l.talla();
        for (l.inicio(); !l.esFin(); l.siguiente()) grado[l.recuperar().getDestino()]++;
    }
    int res = 0;
    for (int v = 0; v < numV; v++) if (grado[v] == 0) res++;
    return res;
}
```

3- 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. En base a esta definición, se pide implementar en la clase Grafo un método público getAristasHA, con el menor coste posible, devuelva el número de aristas Hacia Atrás de un Grafo Dirigido; dicho método solo puede hacer uso de los métodos definidos en la clase Grafo (ver Anexo). **(2.5 puntos)**

```
// SII es un Grafo Dirigido
public int getAristasHA() {
    int res = 0;
    visitados = new int[numVertices()];
    for (int v = 0; v < numVertices(); v++)
        if (visitados[v] == 0) res += getAristasHA(v);
    return res;
}
protected int getAristasHA(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] == 1) res++;
        else if (visitados[w] == 0) res += getAristasHA(w);
    }
    visitados[v] = 2;
    return res;
}
```

4- Dado el siguiente MF-Set:

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

(2 puntos)

- a) Dibujar en el siguiente recuadro cada uno de los subconjuntos disjuntos que contiene el MF-Set: **(0.4 puntos)**
- b) Mostrar cómo se van transformando los árboles después de ejecutar cada una de las siguientes operaciones, haciendo uso de la unión por rango y la compresión de caminos. **(1.6 puntos)**

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

merge(2,5)

0	1	2	3	4	5	6	7	8	9
-3	0	<u>5</u>	2	0	<u>-3</u>	1	0	7	5

find(3)

0	1	2	3	4	5	6	7	8	9
-3	0	5	<u>5</u>	0	-3	1	0	7	5

merge(0,9)

0	1	2	3	4	5	6	7	8	9
<u>5</u>	0	5	5	0	<u>-4</u>	1	0	7	5

find(8)

0	1	2	3	4	5	6	7	8	9
5	0	5	5	0	-4	1	<u>5</u>	<u>5</u>	5

ANEXO

Las clases **NodoABB** y **ABB** del paquete **jerarquicos**

```
class NodoABB<E> {
    E dato; NodoABB<E> izq, der;
    NodoABB(E dato) {...}
}

public class ABB<E> extends Comparable<E>> {
    protected NodoABB<E> raiz; protected int talla;
    public ABB() {...}
    public boolean esVacio() {...}
    public int tamaño() {...}
    ...
}
```

Las clases **Grafo**, **GrafoDirigido** y **Adyacente** del paquete **grafos**.

```
public abstract class Grafo {
    protected int visitados[]; // Para marcar los vértices visitados en un DFS o BFS
    protected int ordenVisita; // Orden de visita de un vértice en un DFS o BFS
    ...
    public abstract int numVertices();
    public abstract int numAristas();
    public abstract boolean existeArista(int i, int j);
    public abstract void insertarArista(int i, int j);
    public abstract void insertarArista(int i, int j, double p);
    public abstract double pesoArista(int i, int j);
    public abstract ListaConPI<Adyacente> adyacentesDe(int i);
    public String toString() {...}
    public int[] toArrayDFS() {...}
    // Recorrido DFS del vértice origen de un grafo
    protected int[] toArrayDFS(int i, int[] res) {...}
    public int[] toArrayBFS() {...}
    // Recorrido BFS del vértice origen de un grafo
    protected int[] toArrayBFS(int i, int[] res) {...}
    ...
}

public class GrafoDirigido extends Grafo {
    protected int numV, numA;
    protected ListaConPI<Adyacente>[] elArray;
    public GrafoDirigido(int numVertices) {...}
    public int numVertices() {...}
    public int numAristas() {...}
    ...
}

public class Adyacente {
    protected int destino; protected double peso;
    public Adyacente(int v, double peso) {...}
    public int getDestino() {...}
    public double getPeso() {...}
    public String toString() {...}
}
```