Resolución de la Recuperación del Tercer Parcial de EDA (19 de Junio de 2015 - 2,4 puntos)

1.- Se pide diseñar un método estático, genérico e iterativo `seleccionDel` que, con el menor coste posible, devuelva el k-ésimo menor elemento de un ABB dado `a`. Para ello, ténganse en cuenta las siguientes condiciones sobre los parámetros del problema: `a` es un ABB no vacío de N nodos, Equilibrado, sin repetidos y que no podrá ser modificado en ningún momento durante la ejecución del método; `k` es un valor entero válido ($1 \leq k \leq N$); solo se podrán usar los métodos públicos de la clase `ABB` para diseñar el método, pues no se tiene acceso a su código. **(0.6 puntos)**

```
public static <E extends Comparable<E>> E seleccionDel(int k, ABB<E> a) {
   E res = a.recuperarMin();
   for (int i = 1; i < k; i++) res = a.sucesor(res);
   return res;
}
```

2. Se denomina inverso (o simétrico) de un ABB aquel que tiene sus mismos elementos pero la propiedad de orden inversa, es decir, que el dato que contiene cada nodo del ABB inverso es menor que todos los de su Hijo Izquierdo y mayor que todos los de su Hijo Derecho. Por ejemplo, en la siguiente figura el ABB de la derecha es el inverso o simétrico del de la izquierda.

En la clase `ABB`, **se pide** implementar un método consultor público `getInverso` que, con el menor coste posible, devuelva el inverso (o simétrico) de un ABB. **(0.6 puntos)**

```
public ABB<E> getInverso() {
   ABB<E> inverso = new ABB<E>();
   inverso.talla = this.talla;
   inverso.raiz = getInverso(this.raiz);
   return inverso;
}
private NodoABB<E> getInverso(NodoABB<E> actual) {
   if (actual == null) return null;
   NodoABB<E> nuevo = new NodoABB<E>(actual.dato);
   nuevo.izq = getInverso(actual.der);
   nuevo.der = getInverso(actual.izq);
   return nuevo;
}
```

3.- En teoría de grafos, un vértice aislado es aquel que tiene grado 0; nótese entonces que, un vértice aislado de un Grafo Dirigido es aquel cuyos grados de entrada y salida son iguales y valen cero.

En la clase `GrafoDirigido`, **se pide** implementar un método consultor público `getAislados` tal que, de la manera más eficiente posible, devuelva una `ListaConPI` con los vértices aislados de un Grafo Dirigido; si el grafo no contuviera ningún vértice aislado, la lista resultado sería vacía. **(0.6 puntos)**

```
public ListaConPI<Integer> getAislados() {
    // PASO 1: calcular los grados de los vértices de this Grafo Dirigido
    int[] grado = new int[numV];
    for (int i = 0; i < numV; i++) {
        // actualizar del grado del vértice i y de sus adyacentes
        ListaConPI<Adyacente> l = elArray[i]; grado[i] += l.talla();
        for (l.inicio(); !l.esFin(); l.siguiente())
            grado[l.recuperar().getDestino()]++;
    }
    // PASO 2: construir la Lista resultado, i.e. recorrer el array grado
    // e insertar en la lista todas las posiciones de este cuyo contenido sea 0
    ListaConPI<Integer> res = new LEGListaConPI<Integer>();
    for (int i = 0; i < numV; i++)
        if (grado[i] == 0) res.insertar(i);
    return res;
}
```

4- En la clase `Grafo`, **se pide** implementar un método público `distanciaConPesos` que, con el menor coste posible, devuelva la distancia con pesos de un camino cualquiera que una un vértice `origen` dado con otro vértice `destino` dado, o -1 si el vértice `destino` no es alcanzable desde `origen`. Supóngase que en el Grafo no hay aristas con pesos negativos. **(0.6 puntos)**

```
public double distanciaConPesos(int origen, int destino) {
   visitados = new int[numVertices()];
   return distanciaConPesosDFS(origen, destino);
}
protected double distanciaConPesosDFS(int actual, int destino) {
   if (actual == destino) return 0;
   visitados[actual] = 1; ListaConPI<Adyacente> l = adyacentesDe(actual);
   for (l.inicio(); !l.esFin(); l.siguiente()) {
       int w = l.recuperar().getDestino(); double pesoVW = l.recuperar().getPeso();
       if (visitados[w] == 0) {
          double longitud = distanciaConPesosDFS(w, destino);
          if (longitud != -1) return pesoVW + longitud;
       }
   }
   return -1;
}
```

Alternativamente, el método `protected` podría escribirse como sigue:

```
protected double distanciaConPesosDFS(int actual, int destino) {
   visitados[actual] = 1; ListaConPI<Adyacente> l = adyacentesDe(actual); double res = -1;
   for (l.inicio(); !l.esFin() && res == -1; l.siguiente()) {
       int w = l.recuperar().getDestino(); double pesoVW = l.recuperar().getPeso();
       if (visitados[w] == 0) {
          if (w == destino) res = pesoVW;
          else {
               double longitud = distanciaConPesosDFS(w, destino);
               if (longitud != -1) res = pesoVW + longitud;
          }
       }
   }
   return res;
}
```

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 NodoAB<E> raiz; protected int talla;
	public ABB() {…}
	public boolean esVacio() {…}
	public int tamanyo() {…}
	public E recuperar(E e) {…}
	public void insertar(E e) {…}
	public void eliminar(E e) {…}
	public E eliminarMin() {…}
	public E recuperarMin() {…}
	public E eliminarMax() {…}
	public E recuperarMax() {…}
	public E sucesor(E e) {…}
	public E predecesor(E e) {…}
	public String toStringInOrden() {…}
	public String toStringPreOrden() {…}
	public String toStringPostOrden() {…}
	public String toStringPorNiveles() {…}
}
```

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
	protected Cola<Integer>;   // Cola auxiliar para poder realizar un 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() {…}
	public int[] toArrayBFS() {…}
	…
}

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() {…}
}
```