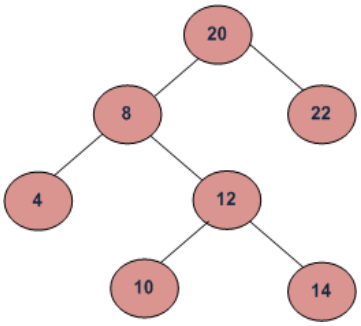
Resolución del Tercer Parcial de EDA (25 de Mayo de 2015) – Puntuación 2.4 puntos

1.- **Se pide** diseñar un método estático, genérico e iterativo `abbSortDesc` que, usando como estructura de datos auxiliar un ABB, ordene descendentemente los elementos (`Comparable`) de un array `v` de la forma más eficiente posible; para ello, asúmase que `v` contiene `v.length` componentes distintas y completamente desordenadas y que solo se pueden usar los métodos públicos de la clase `ABB` (ver Anexo), pues no se tiene acceso a su código. **(0.6 puntos)**

```
public static <E extends Comparable <E>> void abbSortDesc(E[] v) {
   ABB<E> a = new ABB<E>();
   for (int i = 0; i < v.length; i++) a.insertar(v[i]);

   E x = a.recuperarMax(); v[0] = x;
   for (int i = 1; i < v.length; i++) {
      x = a.predecesor(x); v[i] = x;
   }
   //Alternativamente: for (int i = 0; i < v.length; i++) v[i] = a.eliminarMax();
}
```

2.- El Ascendiente Común "Más Bajo", o *Lowest Common Ancestor* (*LCA*), de dos elementos `e1` y `e2` de un Árbol (con Raíz) se define como el elemento situado en el nodo "más bajo" (a mayor profundidad o distancia de la raíz) del que los nodos que contienen a `e1` y `e2` son descendientes, pudiendo ser un nodo descendiente de él mismo. Así, por ejemplo, en el ABB de la siguiente figura: el LCA de 10 y 14 es 12, el de 8 y 14 es 8, el de 10 y 22 es 20 y el de 8 y 22 es 20.



En la clase `ABB` (ver Anexo), **se pide** implementar un método público `lCA` que, en tiempo lineal con su altura, devuelva el LCA de `e1` y `e2` en un ABB Equilibrado, no vacío y sin elementos repetidos; asúmase también que tanto `e1` como `e2` son dos elementos del ABB, por lo que su LCA siempre existe, y que `e1` es menor que `e2`. **(0.6 puntos)**

```
/ ** SII el ABB no es vacío, e1 y e2 están en el árbol y e1 es menor que e2 **/
public E lCA(E e1, E e2) { return lCA(this.raiz, e1, e2).dato; }
//devuelve el Nodo que contiene el LCA de e1 y e2: búsqueda con garantía de éxito
protected NodoABB<E> lCA(NodoABB<E> actual, E e1, E e2) {
   // Como e1 menor que e2, si actual.dato es mayor que e2 también es mayor que e1
   if (actual.dato.compareTo(e2) > 0)
      return lCA(actual.izq, e1, e2);
   // sino, si actual.dato es menor que e1 también es menor que e2
   if (actual.dato.compareTo(e1) < 0)
      return lCA(actual.der, e1, e2);
   // sino, bien actual.dato es e1, bien es e2, bien es mayor que e1 y menor que e2
     return actual.dato;
}
```

3.- En teoría de grafos, el Complemento o Inverso de un grafo G = (V, E) es un grafo G' = (V, E'), con el mismo conjunto de vértices y tal que dos vértices de G' son adyacentes si y sólo si no son adyacentes en G.

En la clase `GrafoDirigido` (ver Anexo), **se pide** implementar un método consultor público `getComplemento` que, con el menor coste posible, devuelva la matriz de Adyacencias que representa al Complemento de un Grafo Simple Dirigido Sin Pesos. **(0.6 puntos)**

```
public boolean[][] getComplemento() {
    boolean[][] res = new boolean [numV][numV];
    //PASO 1: inicializar a true todas las aristas de res, excepto las de la diagonal
    for (int i = 0; i < numV; i++)
        for (int j = 0; j < numV; j++)
            if (i != j) res[i][j] = true;
    //PASO 2: poner a false todas las aristas de res que existen en this Grafo
    for (int i = 0; i < numV; i++) {
        ListaConPI<Adyacente> l = elArray[i];
        for (l.inicio(); !l.esFin(); l.siguiente())
            res[i][l.recuperar().getDestino()] = false;
    }
    return res;
}
```

4- En la clase `Grafo`, se pide implementar un método `tallasDeCC` que, con coste mínimo, devuelva una `ListaConPI` con el nº de vértices (talla) de cada componente conexa de un Grafo. Asume que el Grafo es No-Dirigido. **(0.6 puntos)**

```
public ListaConPI<Integer> tallasDeCC() {
    ListaConPI<Integer> res = new LEGListaConPI<Integer>();
    visitados = new int[numVertices()];
    for (int v = 0; v < numVertices(); v++)
        if (visitados[v] == 0) {
           ordenVisita = 0;
           tallaDeCC(v);
           res.insertar(new Integer(ordenVisita));
        }
    return res;
}
protected void tallaDeCC(int v) {
    visitados[v] = 1; ordenVisita++;
    ListaConPI<Adyacente> l = adyacentesDe(v);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        int w = l.recuperar().getDestino();
        if (visitados[w] == 0) tallaDeCC(w);
    }
}
Alternativamente, sin usar el atributo ordenVisita
public ListaConPI<Integer> tallasDeCC() {
    ListaConPI<Integer> res = new LEGListaConPI<Integer>();
    visitados = new int[numVertices()];
    for (int v = 0; v < numVertices(); v++)
        if (visitados[v] == 0) {
           int talla = tallaDeCC(v);
           res.insertar(new Integer(talla));
        }
    return res;
}
protected int tallaDeCC(int v) {
    visitados[v] = 1; int cont = 1;
    ListaConPI<Adyacente> l = adyacentesDe(v);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        int w = l.recuperar().getDestino();
        if (visitados[w] == 0) cont += tallaDeCC(w);
    }
    return cont;
}
```

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() {…}
	public E predecesor() {…}
	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
	…
	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() {…}
}
```