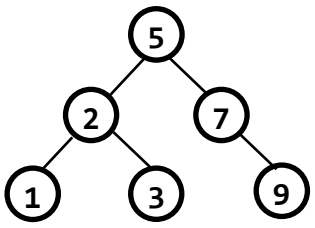
Segundo Parcial de Estructuras de Datos y Algoritmos (EDA) – 3 de Junio de 2022 – Duración: 1h. 30m.

APELLIDOS, NOMBRE	GRUPO

1.- ***(3 puntos)*** Considera el siguiente método de instancia **esPostOrden** de la clase **ABB**, que tiene que comprobar si la lista recibida, no vacía y de tipo **ListaConPI<E>**, contiene una secuencia de valores que se correspondan con el recorrido ***Post-Orden*** del **ABB** en el objeto **this** que lo invoque:

```
/** precondición: l.esVacia() == false */
public boolean esPostOrden(ListaConPI<E> l) {
   if (raiz == null) return false;
   l.inicio();
   esPostOrden(raiz, l);
   return l.esFin();
}
```



Se pide implementar, de la forma más eficiente, el método recursivo (cuya llamada inicial aparece en la anterior lanzadera) para que el método lanzadera funcione como se ha descrito.

Ayuda. En el recorrido Post-Orden de un ABB se visita, en primer lugar, el hijo izquierdo, en segundo lugar, el hijo derecho, y finalmente el nodo padre. Así, para el ABB ejemplo de la figura superior derecha, el método devolverá **true** si la lista recibida es **[1, 3, 2, 9, 7, 5]** y devolverá **false** para cualquier otra lista.

IMPORTANTE. No se permite usar otros métodos auxiliares que devuelvan el recorrido postOrden del árbol. En concreto, no se permite usar el método `toStringPostOrden()`.

```
protected void esPostOrden(NodoABB<E> n, ListaConPI<E> l) {
   if (n.izq != null && !l.esFin() && n.dato.compareTo(l.recuperar()) > 0) {
      esPostOrden(n.izq, l);
   }
   if (n.der != null && !l.esFin() && n.dato.compareTo(l.recuperar()) < 0) {
      esPostOrden(n.der, l);
   }
   if (!l.esFin() && n.dato.compareTo(l.recuperar()) == 0) {
      l.siguiente();
   }
}
```

2.- ***(1,5 puntos)*** Considera el siguiente método de la clase **ABB**, con el que se quiere obtener el número de nodos que no tienen hermanos:

```
public int sinHermanos() {
    if (talla() < 2) return 0;
    return sinHermanos(raiz);
}
```

Se quiere implementar el método cuya llamada inicial aparece en la anterior lanzadera, para que devuelva el número de nodos, descendientes de **n**, que no tengan hermanos (**n.izq** es nodo sin hermano si **n.der** es **null**; **n.der** es nodo sin hermano si **n.izq** es **null**).

Escribe en cada recuadro el número de la opción (ver listado a la derecha) que le corresponde. Una opción puede no aparecer en ningún recuadro, o puede usarse para rellenar uno o varios recuadros.

IMPORTANTE. Cada respuesta **correcta**, **+0,25** puntos. Cada respuesta **incorrecta**, **–0,25** puntos.

```
protected int
sinHermanos(NodoABB<E> n) {
    if ( [2] ) return [8] ;
    if ( [3] ) return [7] ;
    if ( [4] ) return [9] ;
    return 0;
}
```

1	`n.izq == null && n.der == null`
2	`n.izq != null && n.der == null`
3	`n.izq == null && n.der != null`
4	`n.izq != null && n.der != null`
5	`sinHermanos(n.der)`
6	`sinHermanos(n.izq)`
7	`1 + sinHermanos(n.der)`
8	`1 + sinHermanos(n.izq)`
9	`sinHermanos(n.izq) + sinHermanos(n.der)`

3.- ***(1,5 puntos)*** En un grafo, un vértice es un punto de ramificación si tiene grado de salida mayor que 1 (un vértice que sea origen de al menos dos aristas es un punto de ramificación). Se pide implementar, en la clase **Grafo**, un método que devuelva el número de vértices del grafo que son puntos de ramificación.

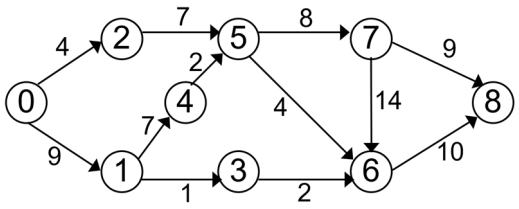
```
public int contarRamificaciones() {
    int cont = 0;
    for (int i = 0; i < numVertices(); i++)
        if (adyacentesDe(i).talla() > 1) cont++;
    return cont;
}
```

4.- ***(4 puntos)*** Se quiere comprobar si, en un **grafo dirigido y acíclico**, existe un camino entre dos vértices (diferentes) tal que el camino lo formen, de forma alternada, aristas con pesos par e impar.

Se quiere devolver **true** si existe al menos un camino desde un vértice, **origen**, a otro vértice, **destino**, tal que los pesos de sus aristas cumplan la alternancia par-impar indicada, y **false** si no existe ningún camino que cumpla la condición.

Ayuda. Un número natural **n** es par si **n%2** es 0, e impar si **n%2** es 1.

Sea el grafo de la figura:



- Si origen=0 y destino=8, entonces el método devolverá **tru e** pues hay un camino (vértices 0,2,5,7,8) donde se recorren 4 aristas de pesos 4,7,8,9 (par, impar, par, impar). Ten en cuenta que (según el contenido de las listas de adyacentes) podría explorarse, y descartarse, otro camino (vértices 0,2,5,6,8), antes de encontrarse el correcto.
- Si origen=2 y destino=6, entonces el método devolverá **true** pues hay un camino (vértices 2,5,6) donde se recorren 2 aristas de pesos 7,4 (impar, par).
- Si origen=1 y destino=8, entonces el método devolverá **false** pues no hay ningún camino que cumpla la condición de alternancia par-impar en los pesos de las aristas.
- Si origen y destino son adyacentes, entonces el camino tiene una sola arista, y el método devolverá **true**.

Se dispone del siguiente método, en la clase **Grafo**:

```
/** precondición: this es GrafoDirigido y acíclico */
public boolean buscar(int origen, int destino) {
   return buscar(origen, destino, true) || buscar(origen, destino, false);
}
```

En la misma clase **Grafo**, se pide implementar el método **buscar(int, int, boolean)**, de 3 argumentos, invocado en el anterior, para que funcione como se ha descrito.

```
protected boolean buscar(int origen, int destino, boolean ant) {
    if (origen == destino) return true;
    ListaConPI<Adyacente> l = adyacentesDe(origen);
    for (l.inicio(); !l.esFin(); l.siguiente()) {
        int w = l.recuperar().getDestino();
        int n = (int) l.recuperar().getPeso();
        boolean par = n % 2 == 0;
        if (par == !ant && buscar(w, destino, par)) {
            return true;
        }
    }
    return false;
}
```

ANEXO

Las clases ABB y NodoABB del paquete `jerarquicos`. Las clases `Adyacente` y `Grafo` del paquete `grafos`.

```
public class ABB<E extends
Comparable<E>> {
   protected NodoABB<E> raiz;
   public ABB() { ... }
   ...
}

class NodoABB<E> {
   protected E dato;
   protected NodoABB<E> izq, der;
   protected int talla;
   NodoABB(E e) { ... }
}

public class Adyacente {
   protected int destino;
   protected double peso;
   public Adyacente(int d,
                double p) { ... }
   public int getDestino() { ... }
   public double getPeso() { ... }
   ...
}
```

```
public abstract class Grafo {
   protected static final double INFINITO =
                          Double.POSITIVE_INFINITY;
   protected int[] visitados;
   protected int ordenVisita;
   protected Cola<Integer> q;
   protected double[] distanciaMin;
   protected int[] caminoMin;
   ...
   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 void insertarArista(int i, int j);
   public abstract ListaConPI<Adyacente>
                                adyacentesDe(int i);
   ...
}
```