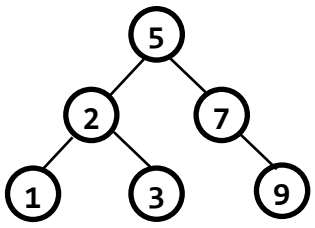
Segundo Parcial de EDA – 11 de Junio de 2021 – Duración: 1 h. 30 m.

APELLIDOS, NOMBRE	GRUPO

1.- ***(3 puntos)*** En la clase **ABB**, implementar un método que reciba una **`ListaConPI`** (genérica, no vacía, ordenada ascendentemente y sin elementos repetidos), y que compruebe si todos los elementos de la lista están en el árbol. Para que sea lo más eficiente posible, se debe explorar el ABB In-Orden.



Ejemplo. Dado el **ABB** de la figura, y una **`ListaConPI`** con los elementos [2, 3, 7, 9], el método devolverá **`true`**; pero si los elementos de la lista son [3, 4, 5, 7], el método devolverá **`false`**.

```
public boolean contiene(ListaConPI<E> lpi) {
   lpi.inicio();
   contiene(lpi, raiz);
   return lpi.esFin();
}

private void contiene(ListaConPI<E> lpi, NodoABB<E> actual) {
   if (actual != null && !lpi.esFin()) {
      contiene(lpi, actual.izq);
      if (!lpi.esFin()) {
         int resCmp = lpi.recuperar().compareTo(actual.dato);
         if (resCmp == 0) lpi.siguiente(); // Dato encontrado -> avanzamos
         if (resCmp >= 0) contiene(lpi, actual.der); // Solo si el elemento de la
      }                                              // lista es mayor que el del ABB
   }
}
```

2.- ***(3 puntos)*** Escribe un método estático, genérico y eficiente que, dada **`cP`**, una **`ColaPrioridad`** de tipo genérico, y dado **`e`**, un dato del mismo tipo, devuelva una **`ListaConPI`**, con los elementos de **`cP`** que sean mayores que **`e`**, **sin** repetidos. **`cP`** puede contener elementos repetidos, pero la **`ListaConPI`** a devolver no.

Además, este método tiene que usar una Pila como estructura de datos auxiliar, pues al terminar su ejecución **`cP`** debe contener los mismos elementos que tenía antes de invocarlo.

Ejemplos. Con una Cola de Prioridad de **`Integer`**:

*Cola de Prioridad, **`cP`***	*`e`*	***`ListaConPI`** resultado*
[2, 2, 3, 4, 4, 7, 7, 9]	1	[2, 3, 4, 7, 9]
[2, 3, 4, 7, 9]	1	[2, 3, 4, 7, 9]
[2, 2, 3, 4, 4, 7, 7, 9]	9	[]
[2, 3, 4, 7, 9]	9	[]
[2, 2, 3, 4, 4, 7, 7, 9]	3	[4, 7, 9]
[2, 3, 4, 7, 9]	3	[4, 7, 9]

Si se utiliza la interfaz `Pila` implementada mediante `ArrayPila`:

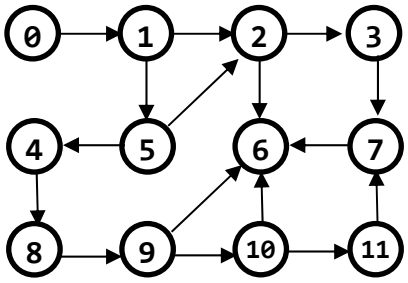
```
public static <E extends Comparable<E>> ListaConPI<E> metodo2(ColaPrioridad<E> cP,
E e) {
    ListaConPI<E> res = new LEGListaConPI<E>();
    Pila<E> aux = new ArrayPila<E>();
    E aComparar = e;
    while (!cP.esVacia()) {
        E min = cP.eliminarMin();
        if (min.compareTo(aComparar) > 0) {
            aComparar = min;
            res.insertar(min);
        }
        aux.apilar(min);
    }
    while (!aux.esVacia()) { cP.insertar(aux.desapilar()); }
    return res;
}
```

Si se utiliza la `Pila` de la Recursión:

```
public static <E extends Comparable<E>> ListaConPI<E> metodo2(ColaPrioridad<E> cP,
E e) {
    ListaConPI<E> res = new LEGListaConPI<E>();
    metodo2(cP, e, res);
    return res;
}

private static <E extends Comparable<E>> void metodo2(ColaPrioridad<E> cP,
E e, ListaConPI<E> res)
{
    if (!cP.esVacia()) {
        E aComparar = e;
        E min = cP.eliminarMin();
        if (min.compareTo(aComparar) > 0) {
            aComparar = min;
            res.insertar(min);
        }
        metodo2(cP, aComparar, res);
        cP.insertar(min);
    }
}
```

3.- ***(4 puntos)*** Un grafo puede representar un laberinto. En tal caso, el vértice **0** es el origen (entrada al laberinto), el vértice **numVertices()-1** es el destino (salida del laberinto), y existe un único camino de origen a destino.



Se pide implementar en la clase **Grafo** un método que obtenga, si existe, la solución del laberinto, y la devuelva como **String**, con los vértices separados por espacios. Para el grafo de la figura, sería: **0 1 5 4 8 9 10 11**. Si en el grafo no existe un camino del vértice **0** al vértice **numVertices()-1**, se devuelve el mensaje "No hay solución".

Ayuda. La exploración a implementar debe situar en el array **res** los vértices de la solución del laberinto en sentido "descendente": en la primera posición del array el vértice **numVertices()-1**, y en la última posición el vértice **0**.

Completa el método público, y escribe el método protegido (recursivo) en el siguiente recuadro:

```
public String laberinto() {
   String sol = "";
   int[] res = new int[numVertices()];
   visitados = new int[numVertices()]; ordenVisita = 0;
   if (laberinto(0, res)) {
      for (int i = ordenVisita - 1; i >= 0; i--) {
         sol += res[i] + " ";
      }
   } else { sol = "No hay solución"; }
   return sol;
}

protected boolean laberinto(int v, int[] res) {
   visitados[v] = 1;
   if (v == numVertices() - 1) {
      res[ordenVisita++] = v;
      return true;
   }
   ListaConPI<Adyacente> l = adyacentesDe(v);
   for (l.inicio(); !l.esFin(); l.siguiente()) {
      int w = l.recuperar().getDestino();
      if (visitados[w] == 0 && laberinto(w, res)) {
         res[ordenVisita++] = v;
         return true;
      }
   }
   return false;
}
```

ANEXO

Las interfaces `ListaConPI`, `ColaPrioridad` y `Pila` del paquete `modelos`.

```
public interface ListaConPI<E> {
   void insertar(E e);
   void eliminar();
   void inicio();
   void siguiente();
   void fin();
   E recuperar();
   boolean esFin();
   boolean esVacia();
   int talla();
}
```

```
public interface ColaPrioridad<E extends
Comparable<E>> {
   void insertar(E e);
   /** SII !esVacia() */ E  eliminarMin();
   /** SII !esVacia() */ E  recuperarMin();
   boolean esVacia();
}

public interface Pila<E> {
    void apilar(E e);
    E desapilar();
    E tope();
    boolean esVacia();
}
```

Las clases `NodoABB` y `ABB` del paquete `jerarquicos`.

```
class NodoABB<E> {
    E dato;
    NodoABB<E> izq, der;
    int talla;
    NodoABB(E dato) {…}
}
```

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

Las clases `Grafo` y `Adyacente` del paquete `grafos`.

```
public abstract class Grafo {
    protected int[] visitados;
    protected int ordenVisita;
    protected Cola<Integer> q;
    ...
    public abstract int numVertices();
    public abstract int numAristas();
    public abstract ListaConPI<Adyacente> adyacentesDe(int i);
    ...
}
public class Adyacente {
    protected int destino;
    protected double peso;
    public Adyacente(int d, double p) { … }
    public int getDestino() { … }
    public double getPeso() { … }
    ...
}
```