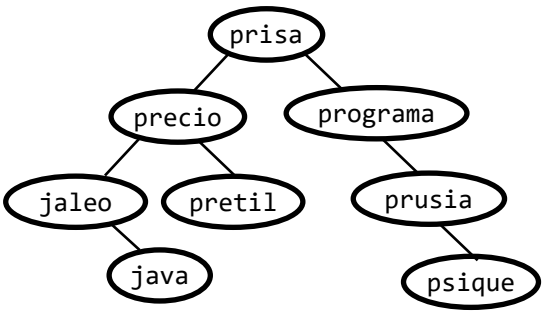
Recuperación Segundo Parcial de Estructuras de Datos y Algoritmos (EDA) – 17 de Junio de 2022

Duración: 1h. 30m.

APELLIDOS, NOMBRE	GRUPO

1.- ***(3 puntos)*** Para representar un ABB de palabras se ha definido la clase **`ABBDeString`**, una derivada de **`ABB`** cuyos elementos son de tipo **`String`**. Se quiere obtener el número de palabras de un **`ABBDeString`** que empiecen por un prefijo dado. Para ello, se dispone del método de instancia **`palabras(String)`**:

```
public class ABBDeString extends ABB<String> {
   public int palabras(String prefijo) {
      return palabras(raíz, prefijo);
   }
   ...
}
```



Se pide implementar eficientemente (con el menor coste posible y sin usar ninguna EDA auxiliar), el método de instancia, recursivo, invocado en la lanzadera, que devuelva el número de palabras que empiecen por el **`prefijo`** dado en el árbol de la **`raiz`** dada.

Ejemplos. Si **`this`** es el árbol de la figura y el prefijo es **`"pr"`**, el método devolverá **`5`**. Mientras que si **`this`** es el mismo árbol y el prefijo es **`"ja"`**, el método devolverá **`2`**.

Ayuda. Usa el método **`boolean startsWith(String prefijo)`** de la clase **`String`** para comprobar si una palabra del **`ABBDeString`** empieza por **`prefijo`**.

```
protected int palabras(NodoABB<String> n, String p) {
    if (n == null) { return 0; }
    if (n.dato.startsWith(p)) {
        return 1 + palabras(n.izq, p) + palabras(n.der, p);
    }
    if (n.dato.compareTo(p) > 0) { return palabras(n.izq, p); }
    return palabras(n.der, p);
}
```

2.- ***(1,5 puntos)*** Se pide completar el siguiente método, **`abbSortDesc`**, para que, usando como EDA auxiliar un **`ABB`**, **ordene descendentemente** los elementos del array **`v`** recibido, de la forma más eficiente posible. Precondición: todas las componentes de **`v`** son distintas (no hay valores repetidos).

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

IMPORTANTE. Cada respuesta **correcta**, **+0,30** puntos. Cada respuesta **incorrecta**, **–0,30** puntos.

```
public static <E extends Comparable <E>> void abbSortDesc(E[] v) {
   ABB<E> a = new ABB<E>();
   E e;
   for (int i = 0; i < v.length; i++) { [5] };
   [1] ;
   [7] ;
   for (int i = 1; i < v.length; i++) {
      [4] ;
      [8] ;
   }
}
```

[1] e = a.recuperarMax()

[2] e = a.recuperarMin()

[3] e = a.sucesor(e)

[4] e = a.predecesor(e)

[5] a.insertar(v[i])

[6] a.eliminar(v[i])

[7] v[0] = e

[8] v[i] = e

3.- ***(1,5 puntos)*** Implementa, en la clase **`GrafoDirigido`**, un método de instancia que reciba dos vértices, **`i`** y **`j`**, y que borre del grafo la arista con origen en **`i`** y destino en **`j`**, en el caso de que exista. Si no existe la arista, el grafo no debe modificarse.

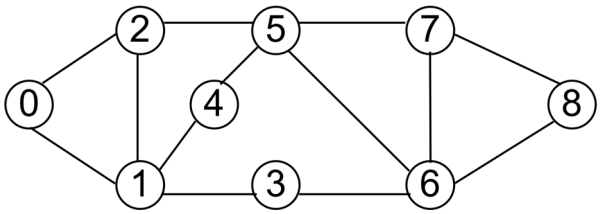
```
// Una solución:
public void eliminarArista(int i, int j) {
   if (existeArista(i, j)) {
      elArray[i].eliminar();
      numA--;
   }
}
// Otra solución:
public void eliminarArista(int i, int j) {
   for (elArray[i].inicio(); ! elArray[i].esFin(); elArray[i].siguiente()) {
      if (elArray[i].recuperar().getDestino() == j) {
         elArray[i].eliminar();
         numA--;
         return;
      }
   }
}
```

4.- ***(4 puntos)*** Sea una red de **N** ordenadores. Cada ordenador **A** puede comunicarse con cada uno de sus vecinos **B** al precio de 1 doblón por comunicación. A través de las conexiones de **B** y siguientes, **A** puede comunicarse con el resto de ordenadores de la red pagando a cada ordenador que utilice para la conexión el doblón correspondiente.

Considera que la red de ordenadores se modeliza mediante un **grafo no dirigido**, **conexo**, **no ponderado**, de **N** vértices.

Implementa, eficientemente, en la clase **`Grafo`** un método de instancia tal que dado un vértice **v** devuelva un array de enteros que contenga el número mínimo de doblones que le cuesta al ordenador (vértice) **v** establecer una conexión con los demás ordenadores (vértices).

Ejemplo. Dado el grafo de la figura, modelo de la red de ordenadores, si **v** = 0 el array devuelto debe ser: [0, 1, 1, 2, 2, 2, 3, 3, 4]. Para el mismo grafo, si **v** = 4, el resultado deber ser: [2, 1, 2, 2, 0, 1, 2, 2, 3].




```
// Una solución (usando una variable local):
public int[] doblonesMinimos(int v) {
   int[] precio = new int[numVertices()];
   int INF = (int) INFINITO;
   for (int i = 0; i < numVertices(); i++) { precio[i] = INF; }
   precio[v] = 0;
   q = new ArrayCola<Integer>(); q.encolar(v);
   while (!q.esVacia()) {
      int u = q.desencolar();
      ListaConPI<Adyacente> l = adyacentesDe(u);
      for (l.inicio(); !l.esFin(); l.siguiente()) {
         int w = l.recuperar().getDestino();
         if (precio[w] == INF) {
            precio[w] = precio[u] + 1;
            q.encolar(w);
         }
      }
   }
   return precio;
}
// Otra solución (usando atributo auxiliar distanciaMin):
public void doblonesMinimos(int v) {
   distanciaMin = new double[numVertices()];
   for (int i = 0; i < numVertices(); i++) { distanciaMin[i] = INFINITO; }
   distanciaMin[v] = 0;
   q = new ArrayCola<Integer>(); q.encolar(v);
   while (!q.esVacia()) {
      int u = q.desencolar();
      ListaConPI<Adyacente> l = adyacentesDe(u);
      for (l.inicio(); !l.esFin(); l.siguiente()) {
         int w = l.recuperar().getDestino();
         if (distanciaMin[w] == INFINITO) {
            distanciaMin[w] = distanciaMin[u] + 1;
            q.encolar(w);
         }
      }
   }
}
```

ANEXO

Las clases **ABB** y **NodoABB** del paquete **jerarquicos**. La interfaz **ListaConPI** del paquete **modelos**.

Las clases **Adyacente**, **Grafo** y **GrafoDirigido** 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 interface ListaConPI<E> {
  void insertar(E e);
  void eliminar();
  void inicio();
  void siguiente();
  void fin();
  E recuperar();
  boolean esFin();
  boolean esVacia();
  int talla();
}


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);
   ...
}

public class GrafoDirigido extends Grafo {
   protected int numV;
   protected int numA;
   protected ListaConPI<Adyacente>[] elArray;
   ...
}
```