Algoritmos y Estructuras de Datos: Examen 2 (Solución)

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Grado en Ingeniería Informática, Grado en Matemáticas e Informática y
Doble Grado en Informática y Administración y Dirección de Empresas

- Este examen dura **100 minutos** y consta de **?? preguntas** que puntúan hasta **?? puntos**.
- Cada **pregunta** se debe entregar en un **fichero distinto**.
- La **hora límite** de entrega del examen serán las **18:40**. Reservad 5 minutos para hacer las fotos de las preguntas al final del examen.
- Todas las hojas entregadas deben indicar, en la parte superior de la hoja, **apellidos**, **nombre**, **DNI/NIE** y **número de matrícula** e incluir en la foto el carnet de la UPM.
- Las calificaciones provisionales de este examen se publicarán el **6 de Julio de 2020** en el Moodle de la asignatura junto con la fecha y lugar de la revisión.

(3 puntos) 1. **Se pide:** Implementar en Java el método:

```
static Map<String,Integer> joinNotas (Map<String,Integer> notas1,
                                      Map<String,Integer> notas2)
```

que recibe como parámetro dos `Map<String,Integer>` con las notas de alumnos identificados por su DNI (de tipo `String`) y devuelve un `Map<String,Integer>` con la unión de ambos listados sumando las notas correspondientes. En caso de que la nota sólo esté en uno de los *maps*, se pondrá la única nota disponible para ese DNI. Los valores devueltos por ambos maps nunca serán `null` y los maps `notas1` y `notas2` tampoco serán `null`.

Por ejemplo dados los maps `notas1 = [<"a",4>,<"b",3>,<"c",3>]` y el map `notas2 = [<"a",5>,<"b",4>,<"d",2>]`, el método debe devolver el map `[<"a",9>,<"b",7>,<"c",3>, <"d",2>]`. Se dispone de la clase `HashTableMap`, que implementa el interfaz `Map` y que dispone de un constructor sin parámetros para crear un `Map` vacío o bien un constructor de copia `Map(Map m)`.

Solución:

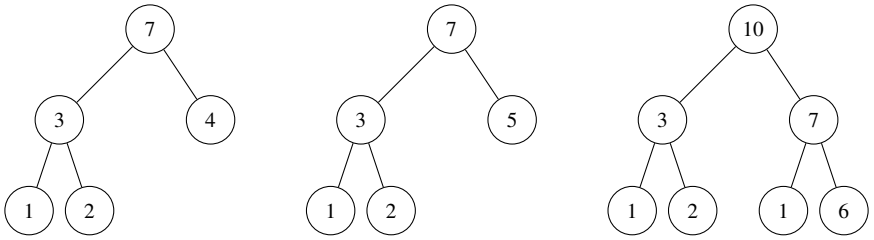
```
static Map<String,Integer> joinNotas (Map<String,Integer> notas1,
                                      Map<String,Integer> notas2) {
  Map<String,Integer> res = new HashTableMap<>(notas1);

  Iterator<String> it = notas2.keys().iterator();
  while (it.hasNext()) {
  String dni = it.next();
  Integer n1 = notas1.get(dni);
    if (n1 == null) {
      res.put(dni, notas2.get(dni));
    }
    else {
      res.put(dni, n1+notas2.get(dni));
    }
  }

  return res;
}
```

(3½ puntos) 2. **Se pide:** Implementar en Java el método:

```
static boolean esArbolSuma (BinaryTree<Integer> tree)
```

que recibe como parámetro un árbol binario propio `tree` que no contendrá elementos `null`. Un árbol binario es *propio* si todos los nodos del árbol, o bien son nodos *hoja*, o bien tienen dos hijos. El método `esArbolSuma()` debe devolver `true` si para todos los nodos internos del árbol, el valor del nodo es la suma de su hijo izquierdo y el valor de su hijo derecho, y `false` en otro caso. El árbol `tree` podría ser `null`, en cuyo caso el método debe lanzar la `IllegalArgumentException`. En caso de que el árbol `tree` esté vacío, el método debe devolver `true`.

Dados los siguientes árboles, el resultado sería:



`true` `false` `true`

Solución:

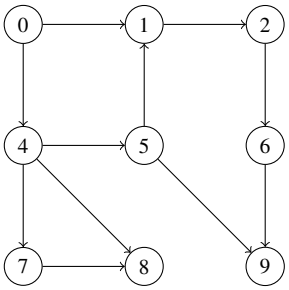
```
static boolean esArbolSuma (BinaryTree<Integer> tree) {
  if (tree == null){
    throw new IllegalArgumentException () ;
  }
  if (tree.isEmpty()) {
    return true;
  }
  return esArbolSuma(tree,tree.root());
}

private static boolean esArbolSuma (BinaryTree<Integer> tree,
                                    Position<Integer> cursor) {
  if (tree.isExternal(cursor)) {
    return true;
  }
  int sumaHijos = tree.left(cursor).element() + tree.right(cursor).element();
  if (cursor.element() != suma) {
    return false;
  }
  return esArbolSuma(tree,tree.left(cursor)) &&
         esArbolSuma(tree,tree.right(cursor));
}
```

(3½ puntos) 3. Se pretende implementar en Java el método `inReachableInSteps` que, dado un grafo *dirigido* `g` y un entero `n`, devuelve `true` si y solo desde el vértice `from` puede alcanzarse el vértice `to` mediante un camino de tamaño menor o igual que `n`.[1] Los vértices `from` y `to` siempre serán vértices del grafo `g` y `n` un entero mayor que 0.

```
public static <V,E> boolean isReachableInSteps (DirectedGraph<V, E> g,
                                                Vertex<V> from,
                                                Vertex<V> to,
                                                int n) {
  Set<Vertex<?>> visited = new HashTableMapSet<>();
  return isReachableInSteps(g, from, to, n, visited);
}
private static <V,E> boolean isReachableInSteps (DirectedGraph<V, E> g,
                                                Vertex<V> from,
                                                Vertex<V> to,
                                                int n,
                                                Set<Vertex<?>> visited) {
   /// COMPLETAR ESTE METODO
}
```

Se pide: Completar el código del método `isReachableInSteps` para que implemente la funcionalidad indicada. El grafo nunca será `null` ni contendrá vértices con elementos `null`.

NOTA: Para añadir elementos al conjunto `visited` podéis usar el método `visited.add(n)`, que añade el vértice `n` al conjunto `visited`.



Por ejemplo, dado el grafo anterior `g`, la invocación a `isReachableInSteps(g,v(0),v(5),1)` devolverá `false`, `isReachableInSteps(g,v(0),v(5),2)` devolverá `true`,
la llamada `isReachableInSteps(g,v(0),v(9),3)` devolverá `true`.
y la llamada `isReachableInSteps(g,v(0),v(9),2)` devolverá `false`.

NOTA: `v(x)` es el el `Vertex` en el que está el nodo con valor "`x`".

Solución:

```
public static <V,E> boolean isReachableInSteps (DirectedGraph<V, E> g,
                                                Vertex<V> from,
                                                Vertex<V> to,
                                                int n) {
  Set<Position<?>> visited = new HashTableMapSet<>();
  return isReachableInSteps(g, from, to, n, visited);
}
static <V,E> boolean isReachableInSteps (DirectedGraph<V, E> g,
                                                Vertex<V> from,
                                                Vertex<V> to,
                                                int n,
                                                Set<Vertex<?>> visited){

  if (from == to) {
    return true;
  }
```

[1] Recordad que el tamaño de un camino lo marca el número de aristas del camino.

```
  if  (n == 0 || visited.contains(from)) {
    return false;
  }

  visited.add(from);
  boolean reachable = false;
  Iterator<Edge<E>> it = g.outgoingEdges(from).iterator();
  while (it.hasNext() && !reachable) {
    reachable = isReachableInSteps(g, g.endVertex(it.next()), to, n-1, visited);
  }
  return reachable;
}
```