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 **90 minutos** y consta de **3 preguntas** que puntúan hasta **10 puntos**.
- Todas las hojas entregadas deben indicar, en la parte superior de la hoja, **apellidos**, **nombre**, **DNI/NIE** y **número de matrícula**.
- Las calificaciones provisionales de este examen se publicarán el **6 de Julio de 2023** en el Moodle de la asignatura junto con la fecha de la revisión.
- **NOTA IMPORTANTE:** Recordad que el uso de métodos auxiliares SÍ está permitido y que NO está permitido modificar ninguna de las estructuras de datos recibidas como parámetro, salvo que se indique lo contrario en el enunciado.

(3 puntos) 1. **Se pide:** Implementar en Java, de forma **recursiva**, el método:

```
public static boolean estaOrdenada (PositionList<Integer> list)
```

que recibe como parámetro una lista de números y devuelve si está ordenada o no. La lista podría ser **null** en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. El uso de bucles (**while**, **for**, **do**–**while** o **for**-each), así como de iteradores, o la creación de estructuras de datos nuevas como pilas, colas, listas, Strings, etc, NO están permitidos. La implementación debe realizarse mediante **métodos auxiliares** que sean **recursivos**.

Por ejemplo, la llamada `estaOrdenada([1,3,2,4])` deberá devolver **false**, la llamada `estaOrdenada([1,2,2,3,4])` deberá devolver **true** y la llamada `estaOrdenada(null)` deberá lanzar la excepción `IllegalArgumentException`.

Solución:

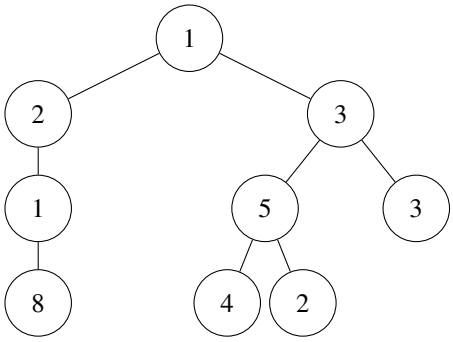
```
public static boolean estaOrdenada (PositionList<Integer> list) {
  if (list == null) {
    throw new IllegalArgumentException();
  }

  return estaOrdenada(list, list.first());
}

private static boolean estaOrdenada(PositionList<Integer> list,
                                    Position<Integer> cursor) {
  if (cursor == list.last()) {
    return true;
  }
  if (cursor.element() > list.next(cursor).element()) {
    return false;
  }
  return estaOrdenada(list,list.next(cursor));
}
```

(3½ puntos) 2. **Se pide:** implementar el método

```
public static Iterable<E> hojasOrdenadas (Tree<E> tree)
```

que recibe como parámetro un árbol `Tree<E>` y devuelve sus nodos hoja ordenados de forma ascendiente. El árbol `tree` nunca será **null** ni podrá contener elementos **null**.

Por ejemplo, dado el siguiente árbol `tree`:



la llamada `hojasOrdenadas(tree)` devolverá un iterable con los siguientes elementos `2,4,3,8`. Se dispone de la clase `HeapPriorityQueue<K,V>` que implementa el interfaz `PriorityQueue<K,V>` y dispone de un constructor sin parámetros permite crear una cola con prioridad vacía. También se dispone de la clase `NodePositionList<K,V>` que implementa el interfaz `PositionList<K,V>` y dispone de un constructor sin parámetros que permite crear una lista vacía.

Solución:

```
public static Iterable<Integer> hojasOrdenadas (Tree<Integer> tree) {
  PriorityQueue<Integer, Void> pq = new HeapPriorityQueue<>();

  hojasOrdenadas(tree, tree.root(), pq);

  PositionList<Integer> res = new NodePositionList<>();
  while (!pq.isEmpty()) {
    res.addLast(pq.dequeue().getKey());
  }
  return res;
}

private static void hojasOrdenadas(Tree<Integer> tree,
                                   Position<Integer> node,
                                   PriorityQueue<Integer, Void> pq) {

  if (tree.isExternal(node)) {
    pq.enqueue(node.element(), null);
  }
  for (Position<Integer> child: tree.children(node)) {
    hojasOrdenadas(tree,child,pq);
  }
}
```

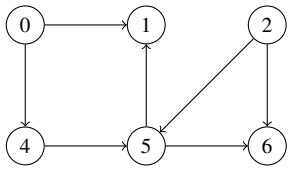
(3½ puntos) 3. **Se pide:** implementar en Java el siguiente método

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

  // COMPLETAR ESTE METODO

}
```

Se pide: dado un grafo *dirigido* `g` y dos vértices, `v1` y `v2`, devuelve true si se puede alcanzar (hay un camino) desde `v1` a `v2`. El grafo nunca será **`null`** ni contendrá vértices con elementos **`null`**. Los vértices `v1` y `v2` nunca serán **`null`** y siempre serán vértices contenidos en el grafo.

Nota: Para añadir elementos al conjunto `visited` podéis usar el método `visited.add(v)`, que añade el vértice `v` al conjunto `visited`. Para comprobar si un elemento está presente en `visited` se puede ejecutar `visited.contains(v)` que devuelve **`true`** si `v` está presente, y **`false`** si no.

Por ejemplo, dado el siguiente grafo `g`, el método pedido deberá devolver lo siguiente. Notad que `v(x)` hace referencia al nodo que contiene el valor `x`:



`isReachable(g,v(0),v(1))` -> **`true`**,
`isReachable(g,v(1),v(0))` -> **`false`**,
`isReachable(g,v(0),v(5))` -> **`true`**,
`isReachable(g,v(0),v(2))` -> **`false`**,
`isReachable(g,v(2),v(1))` -> **`true`**.

Solución:

```
public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
                                         Vertex<V> from,
                                         Vertex<V> to) {
  Set<Position<?>> visited = new HashTableMapSet<>();
  return isReachable(g, from, to, visited);
}
public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
                                         Vertex<V> from,
                                         Vertex<V> to,
                                         Set<Position<?>> visited ) {
  if (from == to) {
    return true;
  }
  if  (visited.contains(from)) {
    return false;
  }

  visited.add(from);
  boolean reachable = false;
  Iterator<Edge<E>> it = g.outgoingEdges(from).iterator();
```

```
    while (it.hasNext() && !reachable) {
      reachable = isReachable(g, g.endVertex(it.next()), to, visited);
    }
    return reachable;
  }
```