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 **4 preguntas** que puntúan hasta **10 puntos**.
- **Las preguntas 1 y 2 deben contestarse en la misma hoja.**
- **Las preguntas 3 y 4 deben contestarse en la misma hoja (distinta de la hoja de las preguntas 1 y 2).**
- 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 **25 de Enero de 2024** 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 del ejercicio.

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

```
PositionList<Integer> getElementosValle (PositionList<Integer> list)
```

que recibe como parámetro una lista `list` y que devuelve una `PositionList` con los elementos contenidos en `list` que son *valle*, es decir, que son estrictamente menores que sus vecinos. Un elemento x_i, que ocupa la posición i, será valle si se cumple que $x_{i-1} < x_i < x_{i+1}$. Por simplicidad, el último y el primer elemento de la lista no se consideran elementos valle. Si la lista de entrada tiene menos de 3 elementos el método debe devolver una lista vacía. Los elementos de la lista resultado deben estar en el mismo orden que en `l`. Disponéis de la clase `NodePositionList`, que implementa la interfaz `PositionList`, y que dispone de un constructor sin parámetros para crear una lista vacía.

Los elementos contenidos en la lista nunca serán **`null`**, pero `list` podría ser **`null`**, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`.

Por ejemplo, la llamada `getElementosValle([1,3,2,4,2,1,3])` devolverá `[2,1]`, `getElementosValle([1,2])` devolverá `[]` y `getElementosValle([4,4,3,4,3,4,4,5])` devolverá `[3,3]`.

Solución:

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

  PositionList<Integer> result = new NodePositionList<>();
  if (list.size() > 2) {
    getElementosValle(list, list.next(list.first()), result);
  }
  return result;
}

private static void getElementosValle(PositionList<Integer> list,
                                      Position<Integer> cursor,
                                      PositionList<Integer> result) {
  if (cursor == list.last()) {
    return;
  }
  Position<Integer> prev = list.prev(cursor);
```

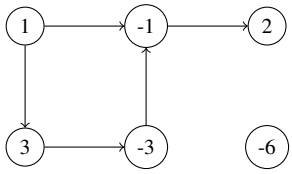
```
    Position<Integer> next = list.next(cursor);
    if (prev.element() < cursor.element() &&
        cursor.element() < next.element()) {
      result.addLast(cursor.element());
    }
    getElementosValle(list, next, result);
  }
```

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

```
<E> Iterable<Integer> getPositiveReachable (DirectedGraph<Integer, E> g,
                                            Vertex<Integer> n)
```

El método `getPositiveReachable` recibe un grafo *dirigido* `g`, con nodos de tipo `Integer`, y un vértice `n` y debe devolver una `PositionList` con los elementos de los vértices alcanzables desde `n` que son *positivos*, es decir, que su valor sea > 0.

Por ejemplo, dado el siguiente grafo `g`, las llamadas a `getPositiveReachable` deben devolver (`n(X)` hace referencia al nodo con valor `X`):




```
getPositiveReachable(g,n(1)) -> [1,3,2]
getPositiveReachable(g,n(3)) -> [3,2]
getPositiveReachable(g,n(2)) -> [2]
getPositiveReachable(g,n(-1)) -> [2]
getPositiveReachable(g,n(-3)) -> [2]
getPositiveReachable(g,n(-6)) -> []
```

NOTA: Recordad que el interfaz `Set<E>` representa un conjunto y tiene los métodos
`void` `add(E elem)`, que añade un elemento al conjunto,
`void` `remove(E elem)`, que lo elimina del conjunto y
`boolean` `contains(Object o)` que devuelve **`true`** si el elemento está contenido en el conjunto.

La clase `HashTableMapSet` implementa el interfaz `Set` y dispone de un constructor sin parámetros para crear un conjunto vacío.

Solución:

```
<E> PositionList<Integer> getPositiveReachable (DirectedGraph<Integer, E> g,
                                                Vertex<Integer> n) {
  Set<Vertex<Integer>> visited = new HashTableMapSet<>();
  PositionList<Integer> result = new NodePositionList<>();
  getPositiveReachable(g, n, visited, result);
  return result;
}


private static <V,E> void getPositiveReachable (DirectedGraph<Integer, E> g,
                                                Vertex<Integer> n,
                                                Set<Vertex<Integer>> visited,
                                                PositionList<Integer> result) {

  if (n.element() > 0) {
    result.addLast(n.element());
  }

  visited.add(n);

  for (Edge<E> edge: g.outgoingEdges(n)) {
    if (!visited.contains(g.endVertex(edge))) {
      getPositiveReachable(g, g.endVertex(edge), visited, result);
```

```
        }
      }
    }
```

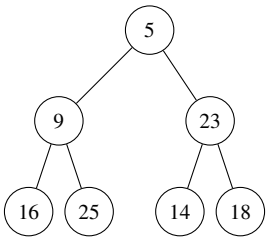
(3 puntos) 3. **Se pide:** implementar el método

```
public static boolean allParentsOrdered(BinaryTree<Integer> tree) {
```

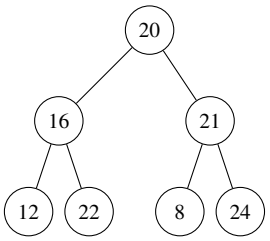
que recibe como parámetro un árbol binario `BinaryTree<Integer>` y debe devolver **`true`** si todos los nodos internos del árbol están *ordenados*. Decimos que un nodo interno está ordenado cuando el valor contenido en el nodo es mayor que el valor de su hijo izquierdo y menor que el valor de su hijo derecho.

El árbol `tree` siempre será un árbol binario propio, es decir, todo nodo interno siempre contendrá dos hijos. Tanto `tree` como los elementos que contiene nunca serán **`null`**.

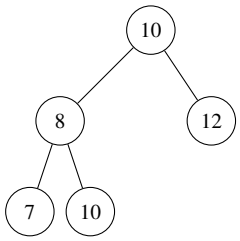
Dados los siguientes árboles el resultado de llamar al método `allParentsOrdered(tree)` sería:



`false`



`true`



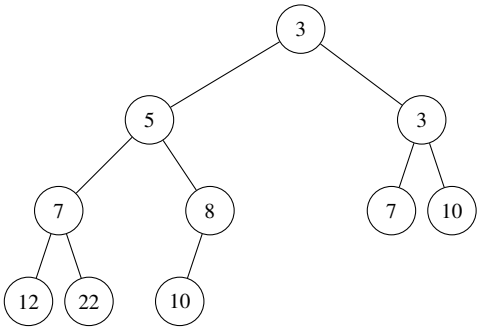
`true`

Solución:

```
public static boolean allParentsOrdered(BinaryTree<Integer> tree) {
  return allParentsOrdered(tree, tree.root());
}

private static boolean allParentsOrdered(BinaryTree<Integer> tree,
                                         Position<Integer> node) {
  if (tree.isExternal(node)) {
    return true;
  }
  if (tree.left(node).element() >= node.element() ||
      tree.right(node).element() <= node.element()) {
    return false;
  }
  return allParentsOrdered(tree, tree.left(node)) &&
         allParentsOrdered(tree, tree.right(node));
}
```

(1 punto) 4. Dado el siguiente montículo (*heap*)



Se pide: Dibujar el estado final del montículo después de ejecutar las siguientes operaciones.

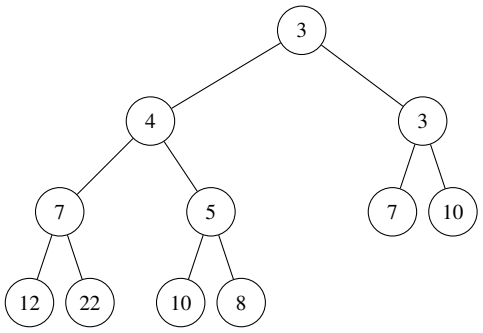
(a) `enqueue(4)`

(b) `dequeue()`

IMPORTANTE: Las operaciones `enqueue(4)` y `dequeue()` deben aplicarse sobre el montículo inicial del dibujo. **NO** apliquéis `dequeue()` sobre el montículo resultante de ejecutar `enqueue(4)`.

Solución:

Apartado (a)



Apartado (b)