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**.
- **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 **1 de Febrero 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 del ejercicio.

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

```
public static <E> void invertir (PositionList<E> list)
```

que recibe como parámetro una lista `list` y que **modifica la lista de entrada**, invirtiendo el orden de los elementos contenidos en `list`. La lista `list` puede ser **`null`**, en cuyo caso el método debe lanzar la exepción `IllegalArgumentException`.

Por ejemplo, dada `list = [1,2,3,4,5,null,7,8]`, la llamada `invertir(list)` deberá dejar `list` con el contenido `[8,7,null,5,4,3,2,1]`.

Solución:

```
public static <E> void invertir (PositionList<E> list) {
  if (list == null) {
    throw new IllegalArgumentException();
  }

  if (list.size() > 1) {
    invertir(list,list.first(), list.last());
  }
}

private static <E> void invertir(PositionList<E> list,
                                 Position<E> first,
                                 Position<E> cursor) {
  Position<E> prev = list.prev(cursor);
  list.addLast(cursor.element());
  list.remove(cursor);

  if (first != cursor) {
    invertir(list,first,prev);
  }
}
```

(3 puntos) 2. Se dispone del siguiente código, que está incompleto:

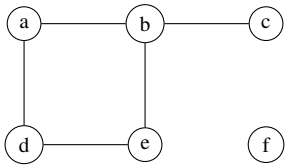
```
public static <V,E> PositionList<Vertex<V>> findOneSimplePath (UndirectedGraph<V, E> g,
                                                               Vertex<V> from,
                                                               Vertex<V> to) {
  Set<Vertex<V>> visited = new HashTableMapSet<>();
  PositionList<Vertex<V>> path = new NodePositionList<>();
  boolean found = findOneSimplePath(g, from, to, visited, path);
  return /** Fragmento 0 **/;
}

public static <V,E> boolean findOneSimplePath (UndirectedGraph<V, E> g,
                                               Vertex<V> from,
                                               Vertex<V> to,
                                               Set<Vertex<V>> visited,
                                               PositionList<Vertex<V>> path) {
  if (/** Fragmento 1 **/) {
    return false;
  }
  path.addLast(from);
  if (/** Fragmento 2 **/ ) {
    return true;
  }
  visited.add(from);
  boolean found = false;
  Iterator<Edge<E>> it = g.edges(from).iterator();
  while (/** Fragmento 3 **/ ) {
    found = findOneSimplePath(g, g.opposite(from, it.next()), to, visited, path);
  }

  if (!found) {
    /** Fragmento 4 **/
  }
  return found;
}
```

Dicho código pretende, dado un grafo *no dirigido* `g` y dos vértices `from` y `to`, devolver un **camino simple** cualquiera, desde el nodo `from` al nodo `to`, si éste es alcanzable o **`null`** en caso contrario. Recuerda que un camino simple es un camino que no repite nodos.

Por ejemplo, dado el siguiente grafo `g`, las llamadas a `findOneSimplePath` deben devolver:




```
findOneSimplePath(g,a,c) -> [a,b,c] o [a,d,e,b,c]
findOneSimplePath(g,a,f) -> null
```

Se pide: Indicar el código necesario en cada uno de los fragmentos para que el programa funcione correctamente. Por ejemplo, para el fragmento 0 la respuesta sería: `(found? path : null)`.

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

Solución:

```
public static <V,E> PositionList<Vertex<V>> findOneSimplePath (UndirectedGraph<V,
Vertex<V> from,
Vertex<V> to) {

  Set<Vertex<V>> visited = new HashTableMapSet<>();
  PositionList<Vertex<V>> path = new NodePositionList<>();
  boolean found = findOneSimplePath(g, from, to, visited, path);
  return found ? path : null;
```

```
}

public static <V,E> boolean findOneSimplePath (UndirectedGraph<V, E> g,
                                               Vertex<V> from,
                                               Vertex<V> to,
                                               Set<Vertex<V>> visited,
                                               PositionList<Vertex<V>> path) {

  if  (visited.contains(from)) {
    return false;
  }

  path.addLast(from);
  if (from == to) {
    return true;
  }

  visited.add(from);
  boolean found = false;
  Iterator<Edge<E>> it = g.edges(from).iterator();
  while (it.hasNext() && !found) {
    found = findOneSimplePath(g, g.opposite(from, it.next()), to, visited, path);
  }
  if (!found) {
    path.remove(path.last());
    visited.remove(from);
  }
  return found;
}
```

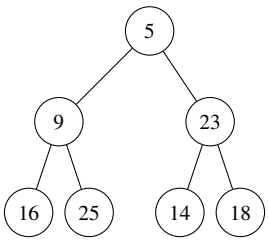
(3 puntos) 3. **Se pide:** implementar el método

```
public static PositionList<Pair<Position<Integer>,Position<Integer>>>
       incumplenHOP (BinaryTree<Integer> tree) {
```

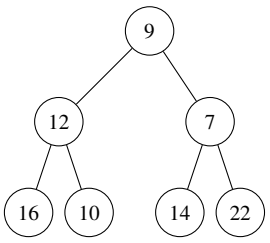
que recibe como parámetro un árbol binario `Tree<Integer>` y debe devolver todos los pares *(padre, hijo)* que incumplen la *heap-order-property*. La *heap-order-property* establece lo siguiente: *el elemento contenido en cada nodo es mayor o igual que el elemento contenido en su padre, si tiene padre.* Tanto `tree` como su contenido nunca serán **`null`**.

El objeto devuelto es de la clase `PositionList<Pair<Position<Integer>,Position<Integer>>>`, es decir, una lista que contiene objetos de tipo `Pair`. Podéis utilizar el constructor `Pair<>(e1,e2)` para crear elemenentos de tipo `Pair`. Los componentes de cada par deben ser objetos de tipo `Position<Integer>`, es decir, el nodo padre y el nodo hijo que incumplen la propiedad reseñada. En el par anterior, `e1` corresponderá al nodo padre y `e2` al nodo hijo. El orden de los pares dentro de la lista resultado no es relevante.

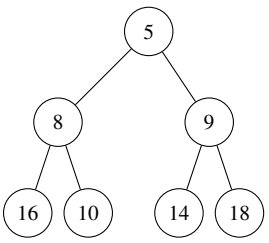
Dados los siguientes `tree` y teniendo en cuenta que `p(x)` hace referencia al nodo de tipo `Position<Integer>` que contiene el valor `x`, el resultado de llamar al método `incumplenHOP(tree)` sería:



`[<p(23),p(14)>,<p(23),p(18)>]`



`[<p(12),p(10)>,<p(9),p(7)>]`



`[]`

Solución:

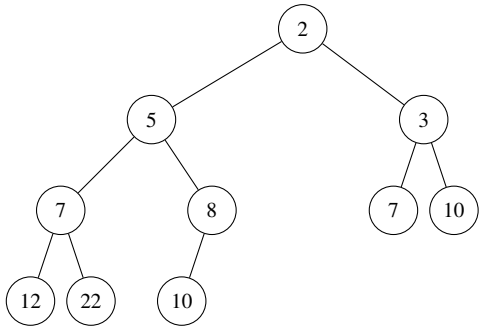
```
public static PositionList<Pair<Position<Integer>,Position<Integer>>>
                incumplenHOP (BinaryTree<Integer> tree) {

    PositionList<Pair<Position<Integer>,Position<Integer>>> res;
    res = new NodePositionList<>();
    incumplenHOP (tree,tree.root(), res);
    return res;
}

void incumplenHOP (BinaryTree<Integer> tree,
                   Position<Integer> node,
                   PositionList<Pair<Position<Integer>,Position<Integer>>> res) {

    if (tree.parent(node)!=null && node.element()<tree.parent(node).element()) {
      res.addLast(new Pair<>(tree.parent(node), node));
    }
    if (tree.hasLeft(node)) {
      incumplenHOP  (tree,tree.left(node),res);
    }
    if (tree.hasRight(node)) {
      incumplenHOP  (tree,tree.right(node),res);
    }
}
```

(1½ puntos) 4. Dado el siguiente montículo (*heap*)



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

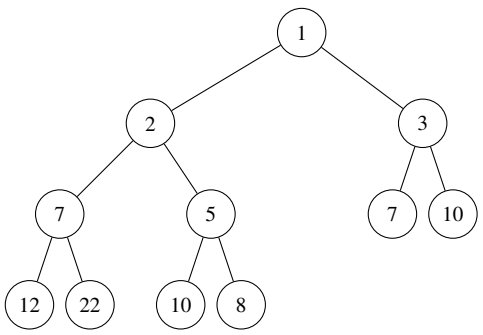
(a) `enqueue(1)`

(b) `dequeue()`

IMPORTANTE: NO apliquéis `dequeue()` sobre el montículo resultante de ejecutar `enqueue(1)`. Tanto `enqueue(1)` como `dequeue()` se aplican sobre el montículo tal como está en el dibujo.

Solución:

Apartado (a)



Apartado (b)