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 **22 de Enero de 2019** en el Moodle de la asignatura junto con la fecha y lugar de la revisión.

(3 puntos) 1. En un árbol binario cuyos nodos contienen enteros no positivos, definimos el valor de un camino como la suma de los valores contenidos en los nodos que constituyen ese camino. **Se pide:** Implementar en Java el método:

```
static Integer maximoCamino (BinaryTree<Integer> tree)
```

que recibe como parámetro el árbol binario `tree`, cuyos nodos contienen enteros **positivos**, y devuelve **el valor del camino que empieza en la raíz y cuyo valor es máximo**. Si `tree` es `null`, el método debe lanzar la excepción `IllegalArgumentException`. Los nodos de `tree` no contienen elementos `null`.

Por ejemplo, dado el árbol del Ejercicio **??**, el método debe devolver 49, ya que el camino con mayor valor desde la raíz es el que termina en la hoja con valor 25, y el método devuelve $4+5+15+25=49$. **Nota:** aunque pueden generarse caminos y elegir aquel con valor máximo, no es necesario hacerlo para resolver el problema.

Solución:

```
public static double maximoCamino (BinaryTree<Integer> tree) {
  if (tree == null) {
    throw new IllegalArgumentException();
  }
  return maximoCamino(tree, tree.root());
}

public static Integer maximoCamino (BinaryTree<Integer> tree,
                                    Position<Integer> node) {

  Integer sumLeft = 0;
  Integer sumRight = 0;

  if (tree.hasLeft(node)) {
    sumLeft = maximoCamino(tree,tree.left(node));
  }
  if (tree.hasRight(node)) {
    sumRight = maximoCamino(tree,tree.right(node));
  }

  return node.element() + Math.max(sumLeft,sumRight);
}

/// Otra posible solucion

public static double maximoCamino (BinaryTree<Integer> tree) {
  if (tree == null) {
```

```
    throw new IllegalArgumentException();
  }
  return maximoCamino(tree, tree.root(), 0, 0);
}

public static Integer maximoCamino (BinaryTree<Integer> tree,
                                    Position<Integer> node,
                                    int current,
                                    int maximo) {

  current += node.element();

  if (tree.isExternal(node)) {
    return Math.max(maximo, current);
  }

  if (tree.hasLeft(node)) {
    maximo = maximoCamino(tree,tree.left(node), current, maximo);
  }
  if (tree.hasRight(node)) {
    maximo = maximoCamino(tree,tree.right(node), current, maximo);
  }
  return maximo;
}
```

(2½ puntos) 2. Decimos que un *map* es la inversa de otro cuando todos los elementos de los pares `<k,v>` del original aparecen en el inverso pero con `v` como nueva clave y `k` como nuevo valor – es decir, como pares `<v,k>`. Para que un map sea invertible, todos sus valores deben ser distintos y no nulos.

Se pide: Implementar en Java el método:

```
static Map<String,String> invertir (Map<String,String> map)
```

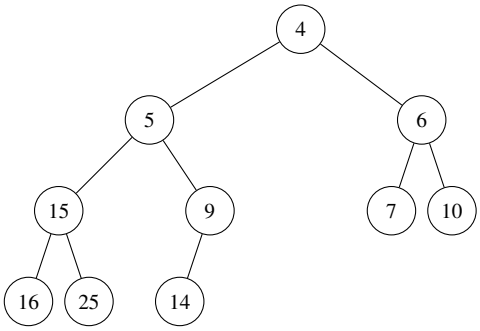
que recibe como parámetro un `Map<String, String>` invertible y devuelve el map inverso.

Por ejemplo, dado `map = [<"k1","v1">,<"k2","v2">,<"k3","v3">]`, el método debe devolver un nuevo map con los pares `[<"v1","k1">,<"v2","k2">,<"v3","k3">]`.

Solución:

```
public static Map<String,String> invertir (Map<String,String> map) {
  Map<String,String> mapres = new HashTableMap<>();
  Iterator<Entry<String,String>> it = map.entries();
  while(it.hasNext()) {
    Entry<String,String> entry = it.next();
    mapres.put(entry.getValue(), entry.getKey());
  }
  return mapres;
}
```

(2 puntos) 3. Dado el siguiente montículo (*heap*)



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

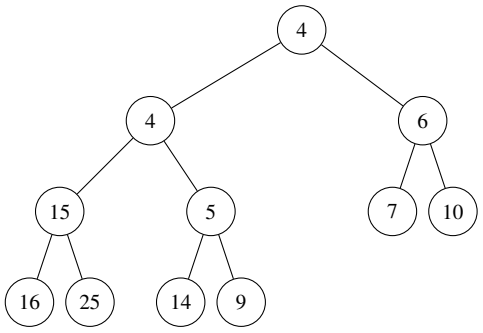
(a) `enqueue(4)`

(b) `dequeue()`

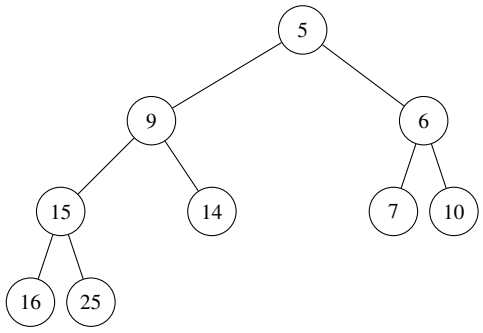
IMPORTANTE: NO apliquéis `dequeue()` sobre el montículo resultante de ejecutar `enqueue(4)`. Tanto `enqueue(4)` como `dequeue()` se aplican sobre el montículo tal como está en el dibujo.

Solución:

Apartado (a)



Apartado (b)



(2½ puntos) 4. Se pretende implementar en Java el método

```
public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
                                         Vertex<V> from,
                                         Vertex<V> to)
```

que devuelve `true` si desde el vértice `from` se puede alcanzar el vértice `to` y `false`, en caso contrario. Nos proporcionan el siguiente código **erróneo** para resolver este problema:

```
1  public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
2                                           Vertex<V> from,
3                                           Vertex<V> to) {
4    Set<Vertex<V>> visited = new HashTableMapSet<Vertex<V>>();
5    return isReachable(g, from, to, visited);
6  }
7
8  public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
9                                           Vertex<V> from,
10                                          Vertex<V> to,
11                                          Set<Vertex<V>> visited ) {
12
13   if (from == to) {
14     return false;
15   }
16
17   visited.add(from);
18   boolean reachable = false;
19   Iterator<Edge<E>> it = g.edges(from).iterator();
20   while (it.hasNext()) {
21     Vertex<V> other = g.opposite(from, it.next());
22     if (!visited.contains(other)) {
23       isReachable(g, other, to, visited);
24     }
25   }
26   return reachable;
27 }
```

Se pide: Determinar qué cambios son necesarios para que el código devuelva el resultado correcto, no se lance ninguna excepción sea más eficiente, evitando realizar operaciones innecesarias. Para contestar debéis indicar el número de línea en el que está el problema y como quedaría la línea para resolverlo.

Solución:

- L14, el método debe devolver `true` cuando ambos nodos sean iguales.

  ```
  return true;
  ```

- L20, es necesario parar el bucle cuando el valor de `reachable` tome valor `true` después de la llamada recursiva.

  ```
  while (it.hasNext() && !reachable) {
  ```

- L23, es necesario recoger el valor de la llamada recursiva a `isReachable` para poder saber si ha habido éxito en la búsqueda.

  ```
  reachable = isReachable(g, other, to, visited);
  ```

Este sería un posible código que cumple lo pedido. El código en negrita se añade o modifica el código anterior para tener código correcto y eficiente.

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

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

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

  visited.add(from);
  boolean reachable = false;
  Iterator<Edge<E>> it = g.edges(from).iterator();
  while (it.hasNext() && !reachable) {
    Vertex<V> other = g.opposite(from, it.next());
    if (!visited.contains(other)) {
      reachable = isReachable(g, other, to, visited);
    }
  }
  return reachable;
}
```