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**.
- **Las preguntas 1 y 2 deben contestarse en la misma hoja.**
- **La pregunta 3 debe contestarse un una hoja distinta 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 **14 de Febrero de 2021** en el Moodle de la asignatura junto con la fecha y lugar de la revisión.

(3 puntos) 1. `Set<E>` es un interfaz que define las operaciones de un conjunto finito de elementos de clase `E`. Un conjunto es una colección de elementos en la que no existe orden y en la que no hay elementos repetidos. **Se pide:** implementar la clase `MiSet<E>` que implemente el interfaz `Set<E>`, mostrado a continuación, **utilizando internamente un atributo de tipo `Map<E,E>`** (no está permitido añadir otros atributos):

```
public interface Set<E> {
    /** Devuelve true si el conjunto está vacío y false en caso contrario. */
    public boolean isEmpty();

    /** Devuelve el número de elementos del conjunto. */
    public int size();

    /** Añade el elemento ''elem'' al conjunto. Devuelve true si el
      * elemento ya estaba contenido en el conjunto y false en caso
      * contrario. En caso de que ''elem'' sea null el método debe lanzar
      * la excepción IllegalArgumentException y no se añade al conjunto. */
    public boolean add(E elem);

    /** Elimina ''elem'' del conjunto. Devuelve true si el elemento
      * estaba en el conjunto y ha sido eliminado y false en caso
      * contrario. En caso de que ''elem'' sea null el método debe
      * lanzar la excepción IllegalArgumentException. */
    public boolean remove(E elem);

    /** Devuelve true si el elemento ''o'' está en conjunto. En caso
      * de que ''elem'' sea null el método debe devolver false. */
    public boolean contains(Object o);

    /** Devuelve todos los elementos contenidos en el conjunto en una
      * estructura Iterable. El orden de los elementos en el Iterable no
      * es relevante. */
    public Iterable<E> getElements();

    /** Devuelve un nuevo conjunto intersección, que contiene los elementos
      *  que están simultáneamente en los conjuntos ''set'' y ''this''.
      *  El conjunto ''set'' nunca será null */
    public Set<E> intersection(Set<E> set);
}
```

Se dispone de la clase `HashTableMap<K,V>`, que implementa el interfaz `Map<K,V>` y que dispone de un constructor sin parámetros para crear un `Map` vacío.

IMPORTANTE!! En la descripción de los interfaces entregada en el examen hay una errata en el interfaz `Map<E,E>`: Falta el método:

```
  /** Returns true if the map contains an entry with the key argument,
   * and false otherwise. */
  public boolean containsKey(Object key) throws InvalidKeyException;
```

Solución:

```
public class MiSet<E> implements Set<E> {

  private Map<E,E> elements;

  @Override
  public boolean isEmpty() {
    return elements.isEmpty();
  }

  @Override
  public int size() {
    return elements.size();
  }

  @Override
  public boolean add(E elem) {
    if (elem == null) {
      throw new IllegalArgumentException();
    }
    E old = elements.put(elem, elem);
    return old != null;
  }

  @Override
  public boolean remove(E elem) {
    if (elem == null) {
      throw new IllegalArgumentException();
    }
    E old = elements.remove(elem);
    return old != null;
  }

  @Override
  public boolean contains(Object o) {
    return elements.containsKey(o);
  }

  @Override
  public Iterable<E> getElements() {
    return elements.keys();
  }

  @Override
  public Set<E> intersection (Set<E> set) {
    Set<E> res = new MiSet<>();

    for (E e: this.getElements()) {
      if (set.contains(e)) {
        res.add(e);
      }
    }
    return res;
  }
}
```

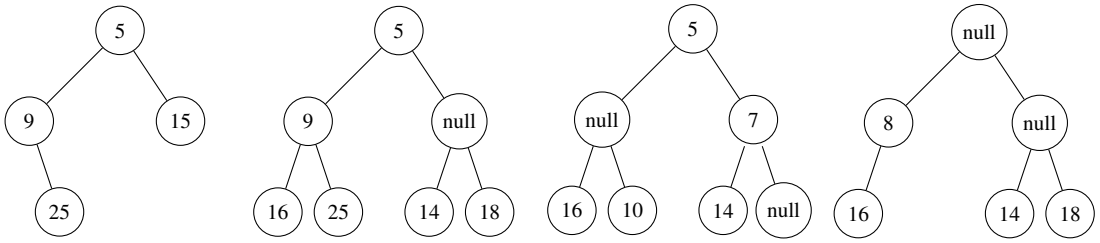
(3½ puntos) 2. **Se pide:** Implementar en Java el método:

```
static int sumaNodos2Hijos (BinaryTree<Integer> tree)
```

que recibe como parámetro un árbol binario `tree` y devuelve la suma de los elementos contenidos en los nodos del árbol que tienen 2 hijos. El árbol `tree` podrá contener elementos null, que no computan en la suma.

El árbol `tree` podría ser **`null`**, en cuyo caso el método debe lanzar la `IllegalArgumentException`. En caso de que `tree` esté vacío, el método debe devolver `0`.

Dados los siguientes árboles, el resultado sería:



`5` `14` `12` `0`

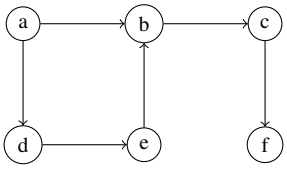
Solución:

```
public static int sumaNodos2Hijos(BinaryTree<Integer> tree) {
  if (tree == null) {
    throw new IllegalArgumentException();
  }
  if (tree.isEmpty()) {
    return 0;
  }

  return sumaNodos2Hijos(tree,tree.root());
}

private static int sumaNodos2Hijos(BinaryTree<Integer> tree,
                                   Position<Integer> v) {
  int suma = 0;
  if (tree.hasLeft(v) && tree.hasRight(v) && v.element() != null ) {
    suma += v.element();
  }
  if (tree.hasLeft(v)) {
    suma += sumaNodos2Hijos(tree,tree.left(v));
  }
  if (tree.hasRight(v)) {
    suma += sumaNodos2Hijos(tree,tree.right(v));
  }

  return suma;
}
```

(3½ puntos) 3. Se quiere implementar en Java el método

`isReachableInNSteps(DirectedGraph<V, E> g, Vertex<V> from, Vertex<V> to, int n)`

que, dado un grafo *dirigido* `g`, dos vértices `from` y `to`, y un valor numérico `n`, devuelve **`true`** si el nodo `to` es alcanzable desde el nodo `from` atravesando, como máximo, `n` aristas. Los valores de los parámetros nunca serán **`null`** y siempre serán valores correctos. Por ejemplo, dado el siguiente grafo `g`, las llamadas a `isReachableInNSteps` deben devolver:




```
isReachableInNSteps(g,a,c,1) -> false
isReachableInNSteps(g,a,c,2) -> true
isReachableInNSteps(g,a,b,2) -> true
isReachableInNSteps(g,b,a,2) -> false
```

Se dispone del siguiente código, que contiene errores:

```
1   public static <V,E> boolean isReachableInNSteps (DirectedGraph<V, E> g,
2                                                    Vertex<V> from,
3                                                    Vertex<V> to,
4                                                    int n) {
5     return isReachableInNStepsError(g, from, to, n);
6   }
7
8   public static <V,E> boolean isReachableInNSteps (DirectedGraph<V, E> g,
9                                                    Vertex<V> from,
10                                                   Vertex<V> to,
11                                                   int n) {
12
13    if (from == to) {
14      return true;
15    }
16    if  (n >= 0) {
17      return false;
18    }
19
20    boolean reachable = false;
21    Iterator<Edge<E>> it = g.edges(from).iterator();
22    while (it.hasNext()) {
23      reachable = isReachableInNSteps(g, g.endVertex(from), to, n, visited);
24    }
25    return reachable;
26  }
```

Se pide: Indicar cuáles son las líneas erróneas y cuál sería el código correcto de las líneas que contienen errores. Por ejemplo:

`L5 -> return isReachableInNSteps(g, from, to, n);`

El código dado tiene 4 líneas con errores (además del error de la línea 5 ya indicado) y las líneas erróneas podrían tener como máximo 2 errores.

Solución:

```
public static <V,E> boolean isReachableInNSteps (DirectedGraph<V, E> g,
                                                 Vertex<V> from,
                                                 Vertex<V> to,
                                                 int n) {
  return isReachableInNSteps(g, from, to, n);
}

public static <V,E> boolean isReachableInNSteps (DirectedGraph<V, E> g,
                                                 Vertex<V> from,
                                                 Vertex<V> to,
                                                 int n) {

  if (from == to) {
    return true;
  }
  if  (n >= 0) { // ERROR -> if (n <= 0)
    return false;
  }

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