Algoritmos y Estructuras de Datos: Examen 1 (recuperación) (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 **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 **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.

(3½ puntos) 1. **Se pide:** Implementar en Java el método:

```
<E> boolean existeEnTodas (PositionList<PositionList<E>> listas, E e)
```

que recibe como parámetro una lista de listas y un elemento `e` y devuelve **`true`** si `e` está en todas las listas contenidas en `listas`.

La lista de entrada nunca será **`null`** ni podrá contener listas **`null`**. Sin embargo, tanto `e` como los valores contenidos en cada una de las listas internas sí podrán ser **`null`**, en cuyo caso deberán ser tratados como cualquier otro valor.

Por ejemplo, dada `l1=[[1,2,3],[2,4],[2],[null,2]]`, la llamada `estaEnTodas(l1,2)` deberá devolver **`true`** y la llamada `estaEnTodas(l1,1)` deberá devolver **`false`**. Asimismo, dada `l2=[[1,null,3],[2,null],[null,2],[9,null,8]]`, la llamada `estaEnTodas(l2,2)` deberá devolver **`false`** y la llamada `estaEnTodas(l2,null)` deberá devolver **`true`**.

Solución:

```
public static <E> boolean existeEnTodas (PositionList<PositionList<E>> listas, E e

  Position<PositionList<E>> cursor = listas.first();
  boolean todasOk = true;
  while (cursor != null && todasOk) {
    todasOk = member(cursor.element(),e);
    cursor = listas.next(cursor);
  }
  return todasOk;

}

private static <E> boolean member (PositionList<E> list, E element) {
  Position<E> cursor = list.first();

  while (cursor != null && !eqNull(cursor.element(),element)) {
    cursor = list.next(cursor);
  }
  return cursor!=null;
}

private static <E> boolean eqNull(E o1, E o2) {
  return o1 == o2 || o1 != null && o1.equals(o2);
}
```

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

```
public static <E> Iterable<E> filtrarNenN (Iterable<E> iterable, int n)
```

que recibe como parámetro un `Iterable<Integer>` y debe devolver un nuevo `Iterable` que contenga los elementos que ocupan las posiciones `0,n,2*n,...` en el mismo orden relativo en que los devuelve `iterable`. El iterable nunca será **`null`**, pero podrá contener elementos **`null`**, en cuyo caso no serán añadidos al resultado y no se tendrán en cuenta en el cómputo de la posición de los elementos. Se dispone de la clase `NodePositionList<E>` y de la clase `ArrayIndexedList<E>` que implementan, respectivamente, las interfaces `PositionList<E>` e `IndexedList<E>` y que disponen de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, si un iterador sobre `iterable` devuelve `1,2,null,3,4,null,null,null,5,6,7,8`, la llamada `filtrarNenN(iter,2)` devolverá un iterable con los elementos `1,3,5,7`, la llamada `filtrarNenN(iter,3)` devolverá los elementos `1,4,7`.

Solución:

```
public static <E> Iterable<E> filtrarNenN (Iterable<E> iterable, int n) {
  PositionList<E> res = new NodePositionList<>();
  int c = 0;
  for (E e: iterable) {
    if (e != null && c % n == 0) {
      res.addLast(e);
    }
    if (e!= null) {
      c++;
    }
  }

 return res;
}
```

(3 puntos) 3. Dado el interfaz `MultiSet<E>`:

```
public interface MultiSet<Element> {

    // Devuelve el número de elementos contenidos en el multiset
    // Complejidad: O(1)
    public int size();

    // Añade elem al multiset
    // Complejidad: O(1)
    public void add(Element elem);

    // Borra del multiset, si elem está contenido en el conjunto, una
    // aparición de elem. Devuelve true si ha borrado algún elemento y
    // false si no lo ha borrado
    // Complejidad: O(1)
    public boolean remove(Element elem);

    // Devuelve true si elem está en el multiset
    // Complejidad: O(1)
    public boolean contains(Element elem);
}
```

Se pide: Implementar la clase `MultiSetMap<E>`, que implementa el interfaz `MultiSet` utilizando para ello un atributo de tipo `Map<Element,Integer>`, que contendrá los elementos contenidos en el multiset (la clave, de tipo `Element`) y su número de apariciones (el valor, de tipo `Integer`). Recuerda que un

multiset es una colección que puede tener elementos repetidos, pero en la que no es relevante el orden de estos elementos. Es necesario inicializar el/los attibutos y **crear el constructor** sin parámetros de la clase `MultiSetMap`, que crea un multiset vacío. Por simplicidad, podéis asumir que nunca se llamará a ningún método con `elem == null`.

INVARIANTE: La implementación debe garantizar que *nunca hay una entrada en el map cuyo número de apariciones sea 0.*

NOTA IMPORTANTE: Es necesario usar, además del atributo de tipo `Map`, al menos otros atributo para conseguir la complejidad indicada en los métodos. Asumid que las operaciones `get`, `put` y `containsKey` sobre un `Map` tienen complejidad $O(1)$.

En las llamadas y comentarios adjuntos que siguen se puede ver la ejecución y el resultado esperado de una secuencia de llamadas a métodos del multiset.

```
MultiSet<Integer> m = new MultiSetMap ();  // m = {}
m.add(2);                                  // m = {2}
m.add(2);                                  // m = {2,2}
m.add(4);                                  // m = {2,2,4}
m.size();                                  // 3
m.contains(3);                             // false
m.contains(2);                             // true
m.remove(3);                               // false
m.size();                                  // 3
m.remove(2);                               // true, m = {2,4}
m.size();                                  // 2
m.contains(2);                             // true
```

Solución:

```
public class MultiSetMap<Element> implements MultiSet<Element>{

  private Map<Element,Integer> elements;
  private int size;

  public MultiSetMap() {
    this.elements = new HashTableMap<Element,Integer>();
    this.size = 0;
  }

  public void add(Element elem) {
    Integer nelem = elements.get(elem);
    elements.put(elem, (nelem==null)?1:nelem+1);
    size++;
  }

  public boolean remove(Element elem) {
    Integer nelem = elements.get(elem);
    if (nelem == null) {
      return false;
    }
    if (nelem == 1) {
      elements.remove(elem);
    }
    else {
      elements.put(elem,elements.get(elem)-1);
    }

    size--;
    return true;
```

```
}

public boolean contains(Element elem) {
  return elements.containsKey(elem);
}
public int size() {
  return size;
}
```

(1 punto) 4. **Se pide:** Indicar la complejidad de los métodos `method1` y `method2`:

```
<E> int method1 (PositionList<E> l) {

  int count = 0;
  for (Position<E> c = l.first();
       c != null;
       c = l.next(c)) {
    Position<E> c2 = l.last();
    while (c2 != null) {
      count++;
      c2 = l.prev(c2);
    }
  }
  return count;
}
```

```
int method2 (int[] l){
  int counter = 0;
  for (int j=l.length; j > 0; j--){
    for (int i=0; i < l.length; i++){
      counter++;
    }
  }
  return counter;
}
```

Solución:

- El método `method1` tiene complejidad $O(n^2)$
- El método `method2` tiene complejidad $O(n^2)$