Algoritmos y Estructuras de Datos: Examen 2

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 2020** en el Moodle de la asignatura junto con la fecha y lugar de la revisión.

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

```
static <E> Map<E,Integer> contarInstancias (PositionList<E> input)
```

que recibe como parámetro una lista `input` y devuelve un `Map` cuyas claves deben ser los elementos de `input` y donde el valor asociado a cada clave debe ser el número de veces que dicha clave aparece en `input`. La lista nunca será `null`, pero podrá contener elementos `null`, que no deben ser contabilizados ni incluidos en el `Map`.

Por ejemplo, para una lista `i1 = ["a","b",null,"c","b"]`, la llamada `contarInstancias(i1)`, devolverá el map con entries `[<"a",1>,<"b",2>,<"c",1>]`; para `i2 = [4,3,null,2,4,2,null]`, la llamada `contarInstancias(i2)`, devolverá el map con entries `[<4,2>,<3,1>,<2,2>]`. Se dispone de la clase `HashTableMap`, que implementa el interfaz `Map` y que dispone de un constructor sin parámetros para crear un `Map` vacío.

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

```
static <E> void imprimirPorOrdenApariciones (Map<E,Integer> map)
```

que recibe como parámetro el `Map` devuelto en el ejercicio anterior y que imprime todos los elementos contenidos en `map` en orden ascendiente respecto al número de apariciones. El argumento `map` nunca será `null`. Se dispone de la clase `HeapPriorityQueue` que implementa el interfaz `PriorityQueue` y que dispone de un constructor sin parámetros para crear una `PriorityQueue` vacía.

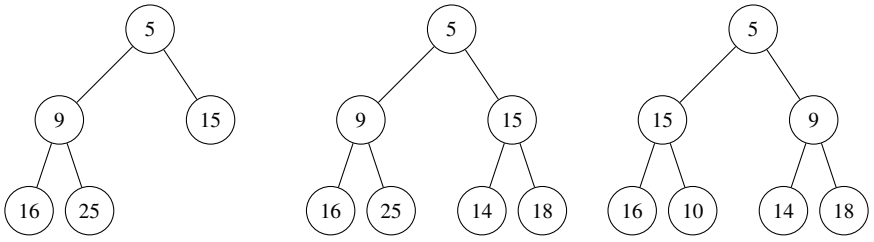
Por ejemplo, dado `map1 = [<"a",1>,<"b",2>,<"c",1>]` la invocación `imprimirPorOrdenApariciones(map1)` deberá imprimir `a c b` o bien `c a b`, y dado `map2 = [<4,2>,<3,1>,<2,2>]` la invocación `imprimirPorOrdenApariciones(map2)` deberá imprimir `3 2 4` o `3 4 2`.

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

```
static boolean estanHijosOrdenados (BinaryTree<Integer> tree)
```

que recibe como parámetro un árbol binario propio `tree` que no contendrá elementos `null`. Un árbol binario es *propio* si todos los nodos del árbol, o bien son nodos *hoja*, o bien tienen dos hijos. El método `estanHijosOrdenados()` debe devolver `true` si para todos los nodos con hijos del árbol, el valor de su hijo izquierdo es menor o igual que el valor de su hijo derecho, y `false` en otro caso. El árbol `tree` podría ser `null`, en cuyo caso el método debe lanzar la `IllegalArgumentException`. En caso de que el árbol `tree` esté vacío, el método debe devolver `true`.

Dados los siguientes árboles, el resultado sería:



`true` `true` `false`

(3 puntos) 4. Se pretende implementar en Java el método `getVerticesAlcanzables` que, dado un grafo *dirigido* `g` y un vértice `n`, devuelve el conjunto de vértices alcanzables desde `n` **pasando sólo por vértices cuyo valor sea positivo**, es decir, que existe un camino en el grafo desde `n` hasta cada uno de dichos vértices sin pasar por ningún vértice con valor negativo. El vértice `n` siempre será un vértice que contendrá un elemento positivo.

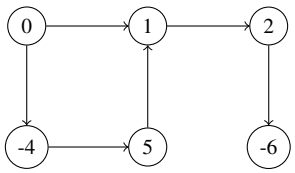
```
<E> Set<Vertex<Integer>> getVerticesAlcanzables (DirectedGraph<Integer, E> g,
                                                 Vertex<Integer> n) {
  Set<Vertex<Integer>> visited = new HashTableMapSet<Vertex<Integer>>();
  visited.add(n);
  getVerticesAlcanzablesRec(g,n,visited);
  return visited;
}
<E> void getVerticesAlcanzablesRec (DirectedGraph<Integer, E> g,
                                    Vertex<Integer> n,
                                    Set<Vertex<Integer>> visited) {

  // COMPLETAR ESTE METODO
}
```

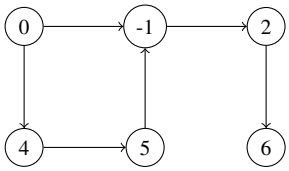
Se pide: Completar el código del método `getVerticesAlcanzablesRec` para que implemente la funcionalidad indicada. El grafo nunca será `null` ni contendrá vértices con elementos `null`.

NOTA: Para añadir elementos al conjunto `visited` podéis usar el método `visited.add(n)`, que añade el vértice `n` al conjunto `visited`.

Por ejemplo, dados los grafos que siguen, el método pedido, empezando en el nodo que contiene `0`, debe devolver los conjuntos reseñados:



$\{0,1,2\}$



$\{0,4,5\}$