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 Ingeniería 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 y 5 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 2018** en el Moodle de la asignatura junto con la fecha y lugar de la revisión.

(2 puntos) 1. Queremos implementar un método en Java que compruebe si un árbol binario cumple la propiedad que caracteriza a los montículos (*heaps*):

> *El elemento contenido en cada nodo es mayor o igual que el elemento contenido en su padre, si tiene padre.*

Se quiere, además, que sea razonablemente eficiente y, en particular, que no realice operaciones innecesarias para determinar si se cumple o no la propiedad. Asumimos que los nodos del árbol contienen objetos de la clase `Integer`, que ningún objeto contenido en un nodo es `null`. El árbol al que aplicamos el método puede estar vacío, en cuyo caso debemos devolver que cumple la propiedad. Nos proporcionan el siguiente código **erróneo** para resolver este problema:

```
1  // Returns true iff tree satisfies the heap-order property
2  public static boolean hasHeapProperty (BinaryTree<Integer> tree) {
3    return tree.isEmpty() || hasHeapProperty(tree,tree.root());
4  }
5
6  public static boolean hasHeapProperty (BinaryTree<Integer> tree,
7                                         Position<Integer> node) {
8
9    if (node.element() < tree.parent(node).element() && tree.parent(node) != null) {
10      return false;
11   }
12
13   boolean res = true;
14   if (tree.hasLeft(node)) {
15      res = hasHeapProperty (tree,tree.left(node));
16   }
17   if (tree.hasRight(node)) {
18      hasHeapProperty (tree,tree.right(node));
19   }
20   return res;
21 }
```

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

Solución:

Apartado (a):

- L9, es necesario invertir el orden del `if` para evitar posibles `NullPointerException`

```
if (tree.parent(node)!=null &&  node.element()<tree.parent(node).element())
```

- L17, es necesario recoger el valor de la llamada recursiva en la variable `res`

```
res = heap (tree,tree.right(node));
```

Apartado (b):

- L18, es necesario considerar el valor de `res` para no recorrer la rama derecha

```
if (res && tree.hasRight(node))
```

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 boolean hasHeapProperty (BinaryTree<Integer> tree) {
  return tree.isEmpty() || hasHeapProperty(tree,tree.root());
}

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

  if (tree.parent(node) != null &&node.element() <tree.parent(node).element()) {
     return false;
  }

  boolean res = true;
  if (tree.hasLeft(node)) {
     res = hasHeapProperty (tree,tree.left(node));
  }
  if (res && tree.hasRight(node)) {
     res = hasHeapProperty (tree,tree.right(node));
  }
  return res;
}
```

(2½ puntos) 2. **Se pide:** Implementar en Java un método con la cabecera

```
public static boolean hasHeapPropertyGen(Tree<Integer> tree)
```

con el mismo objetivo que el código de la pregunta anterior, pero aplicado a un árbol cualquiera. Fijaos que en la pregunta anterior el árbol era `BinaryTree<Integer>` y ahora es `Tree<Integer>`.

Solución:

```
public static boolean hasHeapPropertyGen(Tree<Integer> tree) {
    return tree.isEmpty() || hasHeapPropertyGen(tree, t.root());
}

public static boolean hasHeapPropertyGen (Tree<Integer> tree,
                                          Position<Integer> node) {

  if (tree.parent(node) != null && node.element() < tree.parent(node).element()) {
    return false;
  }

  Iterator<Position<Integer>> it = tree.children(node).iterator();
  boolean res = true;
  while(it.hasNext() && res) {
    res = hasHeapPropertyGen(tree,it.next());
  }
  return res;
```

```
}
```

(1½ puntos) 3. La clase `Book` implementa un libro con un nombre (`String`) y un precio (`Double`), tiene un constructor `Book(String nombre, double precio)` y los getters `getNombre()` y `getPrecio()`.

El método `printCheapestBooks`, que recibe como parámetros un entero `nElems` y un array de libros `books`, debería imprimir, en orden ascendiente de precios, un máximo de `nElem` libros.

```
1  static void printCheapestBooks(int nElems, Book[] books) {
2    PriorityQueue<Double,Book> pq = new HeapPriorityQueue<Double,Book>();
3
4    for (Book book : books) pq.enqueue(book,book); // ordenar los libros usando pq
5
6    int i = 0;
7    while (i < nElems) {
8      Book book = pq.dequeue();
9      System.out.println("El libro " + book.getNombre() + " vale " + book.getPrecio());
10     i++;
11   }
12 }
```

Dado el array `books` abajo mostramos dos ejecucciones correctas:

```
Book[] books = new Book[] { new Book("b1",7.25), new Book("b2",3.0),
                            new Book("b3",10.0) };
```

Ejemplo 1:

```
printCheapestBooks(2,books);

El libro b2 vale 3.0
El libro b1 vale 7.25
```

Ejemplo 2:

```
printCheapestBooks(4,books);

El libro b2 vale 3.0
El libro b1 vale 7.25
El libro b3 value 10.0
```

Se pide: Identificad tres errores en el método `printCheapestBooks` que causan que el código no ejecute correctamente, que pueden provocar una excepción, o que el código no compile, y arreglad dichos errores. Se puede asumir que las líneas 1 y 2 no contienen errores, es decir, no hay errores en la cabecera y la creación de la cola con prioridad es correcta. Para contestar, seguid el formato indicado en la pregunta 1, es decir, el número de línea y cómo debería ser el código en dicha línea para resolver el error.

Solución:

Error 1: en la línea 4 se llama `pq.enqueue(book,book)` pero el primer parametro debería ser un `Double`; no compila.
Arreglo:

```
for (Book book : books) pq.enqueue(book.getPrecio(),book);
```

Error 2: en la línea 7 no se comprueba si `pq` esta vacío; el código puede lanzar una excepción.
Arreglo:

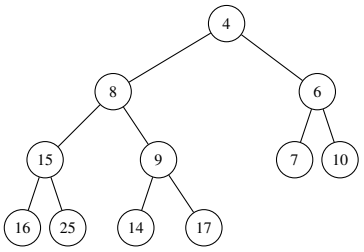
```
    while (i < nElems && !pq.isEmpty()) {
```

Error 3: en la línea 8 `pq.dequeue()` no se devuelve un `Book` sino un `Entry<Double,Book>`; el código no compila.

Arreglo:

```
Book book = pq.dequeue().getValue();
```

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



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

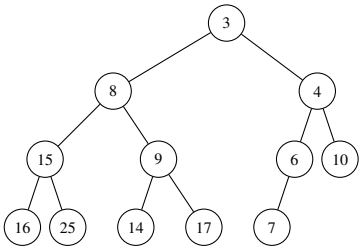
(a) `enqueue(3)`

(b) `dequeue()`

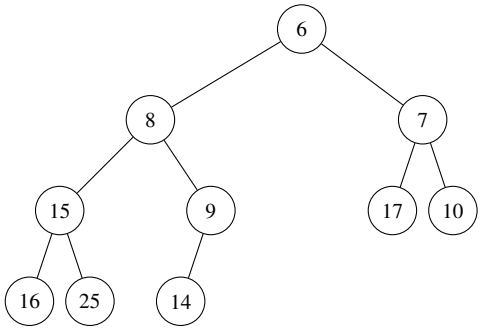
IMPORTANTE: NO apliquéis `dequeue()` sobre el montículo resultante de ejecutar `enqueue(3)`. Tanto `enqueue(3)` como `dequeue()` se aplican sobre el montículo tal como está en el dibujo.

Solución:

Apartado (a)



Apartado (b)



(2 puntos) 5. **Se pide:** Implementar en Java el método

```
public static void aumentarPrecio (Map<String,Double> precios,
                                   String prod, Double coste)
```

que, dada una tabla de precios `precios`, un producto `prod` y un coste `coste`, debe aumentar en dicha tabla el precio del producto teniendo en cuenta lo siguiente: (1) si el producto ya está en la tabla aumentar su precio con el valor de `coste`; (2) si el producto no está en la tabla, se establece como precio del producto el valor de `coste`. La clave del map es el identificador del producto y su valor asociado es el precio. Por ejemplo, el map `precios = <"1",10>,<"3",13>` indica que tenemos dos productos, uno con identificador `"1"` y precio `10` y otro producto con identificador `"3"` y precio `13`.

Por ejemplo, dado `precios=[<"1",10>,<"3",13>]`, la llamada a `aumentarPrecio(precios,"1",2)` debe dejar en el map `precios=[<"1",12>,<"3",13>]`.

O dado, `precios=[<"2",14>,<"8",17>]` la llamada `aumentarPrecio(precios,"5",12)` deberá dejar `precios=[<"2",14>,<"8",17>, <"5",12>]`.

Solución:

```
public static void aumentarPrecio (Map<String,Double> precios,
                                   String prod, Double coste) {

  Double precio = precios.get(prod);
  precio = (precio != null ? coste + precio : coste);
  precios.put(prod,precio);
}
```