







Ejercicio 8a MPS

Participantes:

- Fernando Calvo Díaz
- Álvaro Acedo Espejo
- Miguel Moya Castillo

Informe Jacoco:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 AvlTree		97 %		92 %	7	71	5	197	1	28	0	1
 AvlNode		100 %		96 %	1	33	0	40	0	20	0	1
Total	16 of 783	97 %	7 of 111	93 %	8	104	5	237	1	48	0	2

Como se puede observar en el informe, ninguna clase está testeada al 100% en todos sus métodos o ramas.

Empezamos por el **AvlTree**:

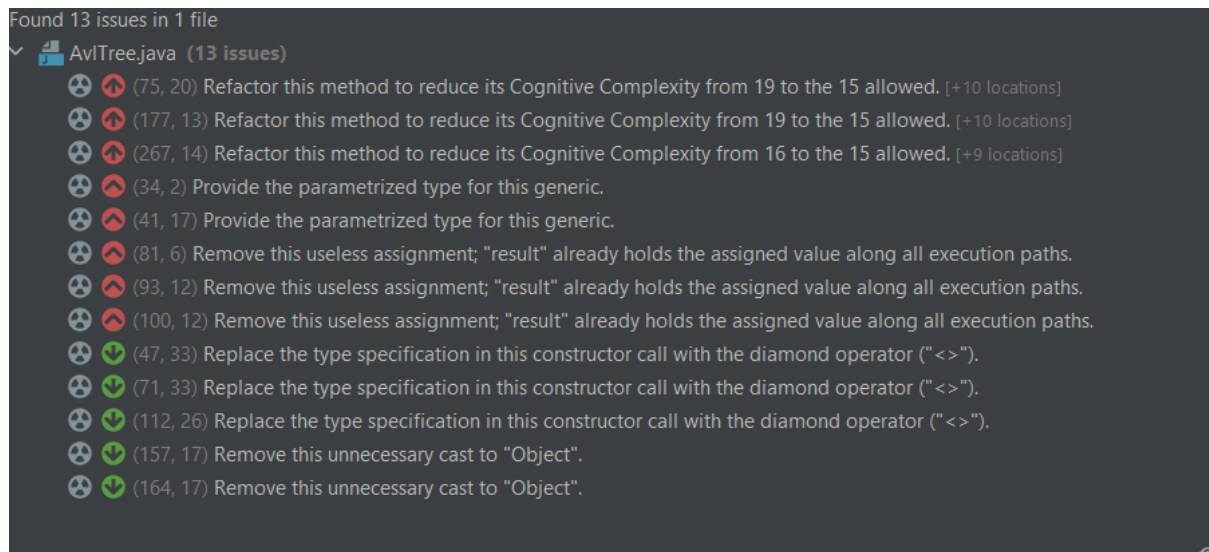
- No se realiza ninguna prueba que use el método ***insert()***
- En el método ***insertAVLNode()***, hay un switch de 3 opciones que nunca pasará por la opción default ya que los valores que le llegan solo son 1 o -1
- En el método ***searchNode()*** hay un ***if(top == null)*** con 2 opciones posibles de las cuales solo se explora una, eso hace que nunca entre en el if y siempre vaya por el else por lo que se salta una línea de código entera
- En el método ***searchNode()*** no se evalúa una rama del ***if(nodeFound != null)***. Además dentro de esa condición, hay otros 2 if cuyas ramas no se exploran: del ***if(successor.hasOnlyALeftChild())*** la condición nunca se cumple por lo que no entra al if y la línea 134 nunca se recorre, y del ***if(successor.hasOnlyARightChild())*** la condición siempre se cumple por lo que hay una rama no explorada
- En el método ***findSuccessor()***, en el ***while(node.hasParent() && (node.getParent().getRight() == node))*** no se comprueba una de 4 ramas

Por otro lado, en la clase **AvlNode**:

- No se ha comprobado una de las cuatro ramas del método ***hasOnlyARightChild()***.

Análisis Sonarqube:

Solamente la clase AvlTree.java presenta problemas de calidad de código. Se adjunta una captura de ellos:



Estas 8 primera cosas son “bad smell” y deberían ser cambiadas:

- 1.- El método public AvlNode<T> searchNode(AvlNode<T> targetNode) se puede refactorizar para reducir su complejidad.
 - 2.- El método public int searchClosestNode(AvlNode<T> node) se puede refactorizar para reducir su complejidad.
 - 3.- El método public void rebalance(AvlNode<T> node) se puede refactorizar para reducir su complejidad.
 - 4.- El parámetro Comparator comparator en el constructor. Proporcionar su tipo parametrizado.
 - 5.- Lo mismo cuando se pasa como parámetro en el constructor del arbol.
 - 6.- En el método public AvlNode<T> searchNode(AvlNode<T> targetNode). En la primera comprobación(línea 81) de top == null la asignación result == null es inútil pues está ya inicializado a null.
 - 7.- En el mismo método en la línea 93 se hace la misma asignación.
 - 8.- En el mismo método en la línea 100 se hace la misma asignación.
- Las demás son minoritarias y no suponen un problema.

¿Cómo arreglar el código?

Refactorización del método public int searchClosestNode(AvlNode<T> node)

```
public AvlNode<T> searchNode(AvlNode<T> targetNode) {  
    AvlNode<T> currentNode = top;  
    while (currentNode != null) {  
        int comparison = compareNodes(targetNode, currentNode)  
        if (comparison < 0) {  
            currentNode = currentNode.getLeft();  
        } else if (comparison > 0) {  
            currentNode = currentNode.getRight();  
        } else {  
            return currentNode;  
        }  
    }  
    return null;  
}
```

Refactorización del método public int searchClosestNode(AvlNode<T> node):

```
public int searchClosestNode(AvlNode<T> node) {
    AvlNode<T> currentNode = top;
    int result = 0;

    while (currentNode != null) {
        int comparison = compareNodes(node, currentNode);
        if (comparison < 0) {
            if (currentNode.hasLeft()) {
                currentNode = currentNode.getLeft();
            } else {
                result = -1;
                1 break;
            }
        } else if (comparison > 0) {
            if (currentNode.hasRight()) {
                currentNode = currentNode.getRight();
            } else {
                result = 1;
                2 break;
            }
        } else {
            3 break;
        }
    }

    if (currentNode != null) {
        node.setClosestNode(currentNode);
    }

    return result;
}
```

Aunque si que es verdad que se podría reducir el número de breaks

Refactorización del método public void rebalance(AvlNode<T> node):

```
public void rebalance(AvlNode<T> node) {
    AvlNode<T> currentNode;
    boolean notFinished;

    currentNode = node;
    notFinished = true;

    1 while (notFinished) {
        2 if (getBalance(currentNode) == -2) {
            3 if (height(currentNode.getLeft().getLeft()) >= height(currentNode.getLeft().getRight())) {
                leftRotation(currentNode);
            } 4 else {
                doubleLeftRotation(currentNode);
            }
        }
        5 if (getBalance(currentNode) == 2) {
            6 if (height(currentNode.getRight().getRight()) >= height(currentNode.getRight().getLeft())) {
                rightRotation(currentNode);
            } 7 else {
                doubleRightRotation(currentNode);
            }
        }
    }

    8 if (currentNode.hasParent()) {
        currentNode.getParent().updateHeight();
        currentNode = currentNode.getParent();
    } 9 else {
        setTop(currentNode);
        notFinished = false;
    }
}
```

Si que es verdad que el SonarLit pone que se puede mejorar la complejidad, pero habría que cambiar más métodos, como por ejemplo hacer una rotación de 3 nodos. Aún así. con este código se ha reducido bastante su complejidad.

Pruebas de caja Negra:

- Creación de un árbol vacío y se comprueba que el tamaño del árbol es cero.
- Inserción de elementos en un árbol vacío: Insertar varios elementos en un árbol vacío y se comprueba que el tamaño del árbol es igual al número de elementos insertados.
- Búsqueda de elementos en un árbol no vacío: Dado un árbol con algunos elementos, se comprueba la búsqueda de un elemento que esté insertado en el árbol y devuelve el resultado correcto. También, la búsqueda de un elemento que no está en el árbol devuelve null.
- Eliminación de elementos de un árbol no vacío: se crea un árbol con algunos elementos y se elimina algún elemento, comprobando que el tamaño del árbol se reduce en uno y que la búsqueda del elemento eliminado devuelve null.

- Recorrido en orden del árbol: se crea un árbol con algunos elementos y se comprueba que el recorrido en orden del árbol devuelve los elementos en orden ascendente.

Pruebas de caja Blanca:

Las pruebas de caja blanca podrían ayudar a algunos de los fallos del código presentado. Las pruebas de este tipo hacen más fácil el asegurar la cobertura de todas las ramas y caminos de ejecución del código. Con los test de caja blanca habría sido sencillo resolver los problemas del switch del *insertAvlNode()* o las ramas no comprobadas, como las de *findSuccessor()* o *hasOnlyRightChild()*.

A continuación, proponemos unas pruebas de caja blanca, las cuales consideramos que habrían sido útiles:

- Prueba de inserción y rotación de nodos: se insertan varios elementos en el árbol para forzar una rotación y se comprueba que el árbol resultante tiene la estructura correcta.
- Prueba de eliminación de nodos con un solo hijo: se crea un árbol donde algunos nodos tienen un solo hijo y se eliminan algunos de estos nodos, comprobando que el árbol resultante tiene la estructura correcta.
- Prueba de rebalanceo del árbol: se crea un árbol desequilibrado y se insertan nodos para forzar el rebalanceo del árbol, comprobando que el árbol resultante tiene la estructura correcta.