

Ejercicio 8.b: Implementación de proyecto Java sobre árboles AVL

Participantes:

- Iker Gálvez castillo
- José María Fernández Cantón
- José Luis López Ruiz
- Álvaro Yuste Moreno
- Rocío Gómez Mancebo

AvlNode

Añadir dos tests para probar al completo el método `hasOnlyARightChild()`

```
public boolean hasOnlyARightChild() {  
    return (hasRight() && !hasLeft());  
}
```

```
@Test  
public void testNotALeftChild() {  
    assertFalse("testHasRight", node.hasRight());  
    AvlNode<Integer> node2 = new AvlNode<Integer>(6);  
    node.setRight(node2);  
    assertTrue(node.hasOnlyARightChild());  
}
```

```
@Test  
public void testJustALeftChild() {  
    AvlNode<Integer> node2 = new AvlNode<Integer>(3);  
    node.setLeft(node2);  
    assertFalse(node.hasOnlyARightChild());  
}
```

Cobertura final tras los cambios:

avlTree > avl > AvlNode

AvlNode

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods
updateHeight()		100 %		100 %	0 5	0 8	0 1
AvlNode(Object)		100 %	n/a		0 1	0 8	0 1
isLeaf()		100 %		100 %	0 3	0 1	0 1
hasOnlyLeftChild()		100 %		100 %	0 3	0 1	0 1
hasOnlyRightChild()		100 %		100 %	0 3	0 1	0 1
hasParent()		100 %		100 %	0 2	0 1	0 1
hasLeft()		100 %		100 %	0 2	0 1	0 1
hasRight()		100 %		100 %	0 2	0 1	0 1
setLeft(AvlNode)		100 %	n/a		0 1	0 2	0 1
setParent(AvlNode)		100 %	n/a		0 1	0 2	0 1
setRight(AvlNode)		100 %	n/a		0 1	0 2	0 1
setItem(Object)		100 %	n/a		0 1	0 2	0 1
setHeight(int)		100 %	n/a		0 1	0 2	0 1
setClosestNode(AvlNode)		100 %	n/a		0 1	0 2	0 1
getLeft()		100 %	n/a		0 1	0 1	0 1
getParent()		100 %	n/a		0 1	0 1	0 1
getRight()		100 %	n/a		0 1	0 1	0 1
getItem()		100 %	n/a		0 1	0 1	0 1
getHeight()		100 %	n/a		0 1	0 1	0 1
getClosestNode()		100 %	n/a		0 1	0 1	0 1
Total	0 of 158	100 %	0 of 26	100 %	0 33	0 40	0 20

AvlTree

Implementar un test para probar el método insert(T item)

```
public void insert(T item) {
    AvlNode<T> node = new AvlNode<T>(item);
    insertAvlNode(node);
}
```

```
@Test
public void testInsertingLeftElementUsingInsert() throws Exception
{
    avlTree.insert(3);
    avlTree.insert(2);

    String tree = " | 3 | 2";
    assertEquals("testInsertingLeftElement", tree,
avlTree.toString());
}
```

Implementar un test para probar al completo el método `searchNode(AvlNode)`

```

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

    currentNode = top;
    if (top == null) {
        result = 0;
    } else {
        int comparison;
        boolean notFound = true;
        while (notFound) {
            comparison = compareNodes(node, currentNode);
            if (comparison < 0) {
                if (currentNode.hasLeft()) {
                    currentNode = currentNode.getLeft();
                } else {
                    notFound = false;
                    node.setClosestNode(currentNode);
                    result = -1;
                }
            } else if (comparison > 0) {
                if (currentNode.hasRight()) {
                    currentNode = currentNode.getRight();
                } else {
                    notFound = false;
                    node.setClosestNode(currentNode);
                    result = 1;
                }
            } else {
                notFound = false;
                node.setClosestNode(currentNode);
                result = 0;
            }
        }
    }

    return result;
}

```

```

@Test
public void search_node_con_vacio_es_null() {
    AvlNode<Integer> targetNode = new AvlNode<>(1);
    assertNull(avlTree.searchNode(targetNode));
}

```

Implementar un test para probar al completo el método findSuccessor(AvlNode<T> node):

```
public AvlNode<T> findSuccessor(AvlNode<T> node) {
    AvlNode<T> result;

    if (node.hasRight()) {
        AvlNode<T> tmp = node.getRight();
        while (tmp.hasLeft()) {
            tmp = tmp.getLeft();
        }
        result = tmp;
    } else {
        while (node.hasParent() && (node.getParent().getRight() == node)) {
            node = node.getParent();
        }
        result = node.getParent();
    }
    return result;
}
```

```
@Test
public void testFindSuccessor() throws Exception {
    node = avlTree.search(22);
    assertEquals("testFindSuccessor", avlTree.search(24),
        avlTree.findSuccessor(node));
    node = avlTree.search(4);
    assertEquals("testFindSuccessor", avlTree.search(8),
        avlTree.findSuccessor(node));
    node = avlTree.search(24);
    assertEquals("testFindSuccessor", null,
        avlTree.findSuccessor(node));
    node = avlTree.search(12);
    assertEquals("testFindSuccessor", avlTree.search(14),
        avlTree.findSuccessor(node));
}
```

Hemos añadido la búsqueda de todos los sucesores comprobando que devuelve realmente el valor que debe ser. Observamos que al realizar la búsqueda del nodo de mayor valor no encuentra, es decir, es null, siguiendo así la estructura AVL.

Implementar un test para probar al completo el método insertAvlNode(AvlNode<T> node):

```

public void insertAvlNode(AvlNode<T> node) {
    ♦ if (avlIsEmpty()) {
        insertTop(node);
    } else {
        int result = searchClosestNode(node);

        ♦ switch (result) {
            case -1:
                insertNodeLeft(node);
                break;
            case +1:
                insertNodeRight(node);
                break;
            default:
                break;
        }
    }
}

```

```

@Test
public void testInsertingIdenticalNodes() throws Exception {
    Comparator<?> comp = Comparator.comparingInt((Integer o) -> o);
    AvlTree<Integer> tree = new AvlTree<>(comp);
    tree.insertTop(new AvlNode<>(2));
    tree.insertAvlNode(new AvlNode<>(2));
    String expected = " | 2";
    assertEquals("testInsertingIdenticalNodes", expected,
        tree.toString());
}

```

Implementar un test para probar al completo el método deleteNode(AvlNode<T> node):

```

public void deleteNode(AvlNode<T> node) {
    AvlNode<T> nodeFound;

    nodeFound = searchNode(node);
    ♦ if (nodeFound != null) {
        ♦ if (nodeFound.isLeaf()) {
            deleteLeafNode(nodeFound);
        } else if (nodeFound.hasOnlyALeftChild()) {
            deleteNodeWithALeftChild(nodeFound);
        } else if (nodeFound.hasOnlyARightChild()) {
            deleteNodeWithARightChild(nodeFound);
        } else { // has two children
            AvlNode<T> successor = findSuccessor(nodeFound);
            T tmp = successor.getItem();
            successor.setItem(nodeFound.getItem());
            nodeFound.setItem(tmp);
            ♦ if (successor.isLeaf()) {
                deleteLeafNode(successor);
            } else {
                deleteNodeWithARightChild(successor);
            }
        }
    }
}

```

MPS - Ejercicio 8

Nos hemos visto en la necesidad de modificar la función `deleteNode()` del código original según un error que encontramos a la hora de elaborar los métodos de prueba, este método inicialmente contenía estas condiciones:

```
if (successor.isLeaf()) {
    deleteLeafNode(successor);
} else if (successor.hasOnlyALeftChild()) {
    deleteNodeWithALeftChild(successor);
} else if (successor.hasOnlyARightChild()) {
    deleteNodeWithARightChild(successor);
}
```

Este cambio ha sido necesario debido a que se llama a la función `findSuccessor`, que tiene dos opciones, si tiene una rama hacia la derecha desde el nodo origen, recorre esta rama hasta encontrar el más pequeño de esta rama, es decir, avanza hacia la izquierda hasta el final, lo que quiere decir que es imposible que tenga un hijo hacia la izquierda aquel que hemos tomado como sucesor ya que el sucesor será el más hacia la izquierda.

Por otra parte, si avanzamos por la parte derecha, necesitamos una rama inicial desde donde avanzamos por la derecha hasta el final para encontrar el sucesor (debido a que tiene que ser el valor más cercano al nodo inicial, es decir, si vas por la izquierda tienes que pillar el mayor de esa rama, y, si vas por la derecha, el menor) donde, al final de este, ha de existir un nodo a la izquierda para poder entrar por esta condición, es decir, la rama derecha ha de estar vacía porque si entra por la rama derecha no puede tener un hijo a la izquierda, y si entra por la rama izquierda obligatoriamente nos va a obligar a tener una diferencia de altura de más de dos, por lo cual hará que el propio árbol se equilibrará.

Además, hemos borrado el `else if` ya que, una vez quitado esto, o es una hoja o tiene hijo derecho, por lo que no tiene sentido comprobar otra condición extra.

```
@Test
public void testDeleteNode() {
    AvlNode<Integer> node;

    node = new AvlNode<Integer>(14);
    avlTree.insertAvlNode(node);

    node = new AvlNode<Integer>(9);
    avlTree.insertAvlNode(node);

    node = new AvlNode<Integer>(30);
    avlTree.insertAvlNode(node);

    node = new AvlNode<Integer>(7);
    avlTree.insertAvlNode(node);

    node = new AvlNode<Integer>(11);
```

MPS - Ejercicio 8

```
avlTree.insertAvlNode(node);
node = new AvlNode<Integer>(25);
avlTree.insertAvlNode(node);

node = new AvlNode<Integer>(38);
avlTree.insertAvlNode(node);

node = new AvlNode<Integer>(6);
avlTree.insertAvlNode(node);

node = new AvlNode<Integer>(8);
avlTree.insertAvlNode(node);
node = new AvlNode<Integer>(10);
avlTree.insertAvlNode(node);

node = new AvlNode<Integer>(13);
avlTree.insertAvlNode(node);

node = new AvlNode<Integer>(22);
avlTree.insertAvlNode(node);

node = new AvlNode<Integer>(27);
avlTree.insertAvlNode(node);

node = new AvlNode<Integer>(35);
avlTree.insertAvlNode(node);



node = new AvlNode<Integer>(40);
avlTree.insertAvlNode(node);

node = new AvlNode<Integer>(12);
avlTree.insertAvlNode(node);










































node = new AvlNode<Integer>(21);
avlTree.insertAvlNode(node);

avlTree.delete(42);
}
```

Cobertura final tras los cambios:

 avlTree >  avl >  AvlTree

AvlTree

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods
● rebalance(AvlNode)		100 %		100 %	0 7	0 17	0 1
● searchClosestNode(AvlNode)		100 %		100 %	0 7	0 23	0 1
● searchNode(AvlNode)		100 %		100 %	0 7	0 20	0 1
● deleteNode(AvlNode)		100 %		100 %	0 6	0 16	0 1
● leftRotation(AvlNode)		100 %		100 %	0 3	0 13	0 1
● rightRotation(AvlNode)		100 %		100 %	0 3	0 13	0 1
● findSuccessor(AvlNode)		100 %		100 %	0 5	0 10	0 1
● deleteLeafNode(AvlNode)		100 %		100 %	0 3	0 8	0 1
● getBalance(AvlNode)		100 %		100 %	0 3	0 7	0 1
● inOrder(AvlNode)		100 %		100 %	0 2	0 6	0 1
● insertAvlNode(AvlNode)		100 %		100 %	0 4	0 9	0 1
● deleteNodeWithALeftChild(AvlNode)		100 %		n/a	0 1	0 5	0 1
● deleteNodeWithARightChild(AvlNode)		100 %		n/a	0 1	0 5	0 1
● insertNodeLeft(AvlNode)		100 %		n/a	0 1	0 4	0 1
● insertNodeRight(AvlNode)		100 %		n/a	0 1	0 4	0 1
● height(AvlNode)		100 %		100 %	0 2	0 5	0 1
● doubleLeftRotation(AvlNode)		100 %		n/a	0 1	0 4	0 1
● doubleRightRotation(AvlNode)		100 %		n/a	0 1	0 4	0 1
● AvlTree(Comparator)		100 %		n/a	0 1	0 4	0 1
● insert(Object)		100 %		n/a	0 1	0 3	0 1
● search(Object)		100 %		n/a	0 1	0 2	0 1
● compareNodes(AvlNode, AvlNode)		100 %		n/a	0 1	0 1	0 1
● setTop(AvlNode)		100 %		n/a	0 1	0 3	0 1
● delete(Object)		100 %		n/a	0 1	0 2	0 1
● avlIsEmpty()		100 %		100 %	0 2	0 1	0 1
● toString()		100 %		n/a	0 1	0 2	0 1
● insertTop(AvlNode)		100 %		n/a	0 1	0 2	0 1
● getTop()		100 %		n/a	0 1	0 1	0 1
Total	0 of 615	100 %	0 of 81	100 %	0 69	0 194	0 28