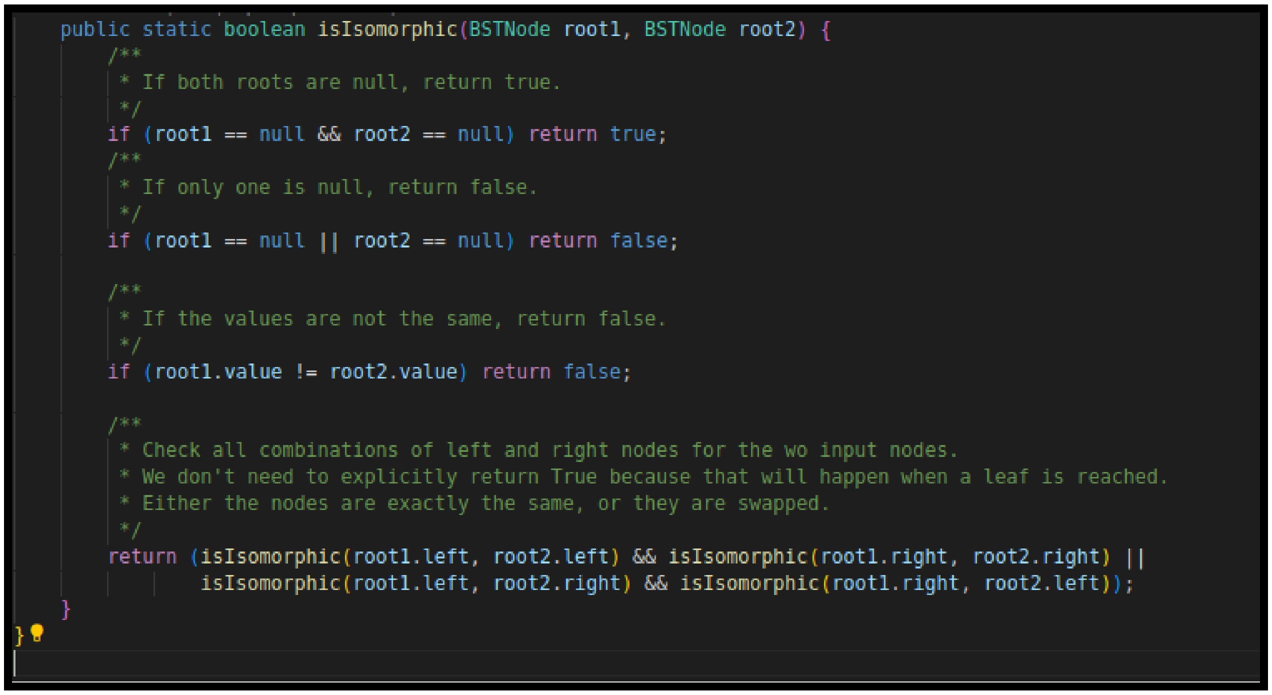
Programming Assignment 02
1DV516

Name: Tadj cazaubon
Student: tc222gf
Email: tc222gf@student.lnu.se

Problem 05

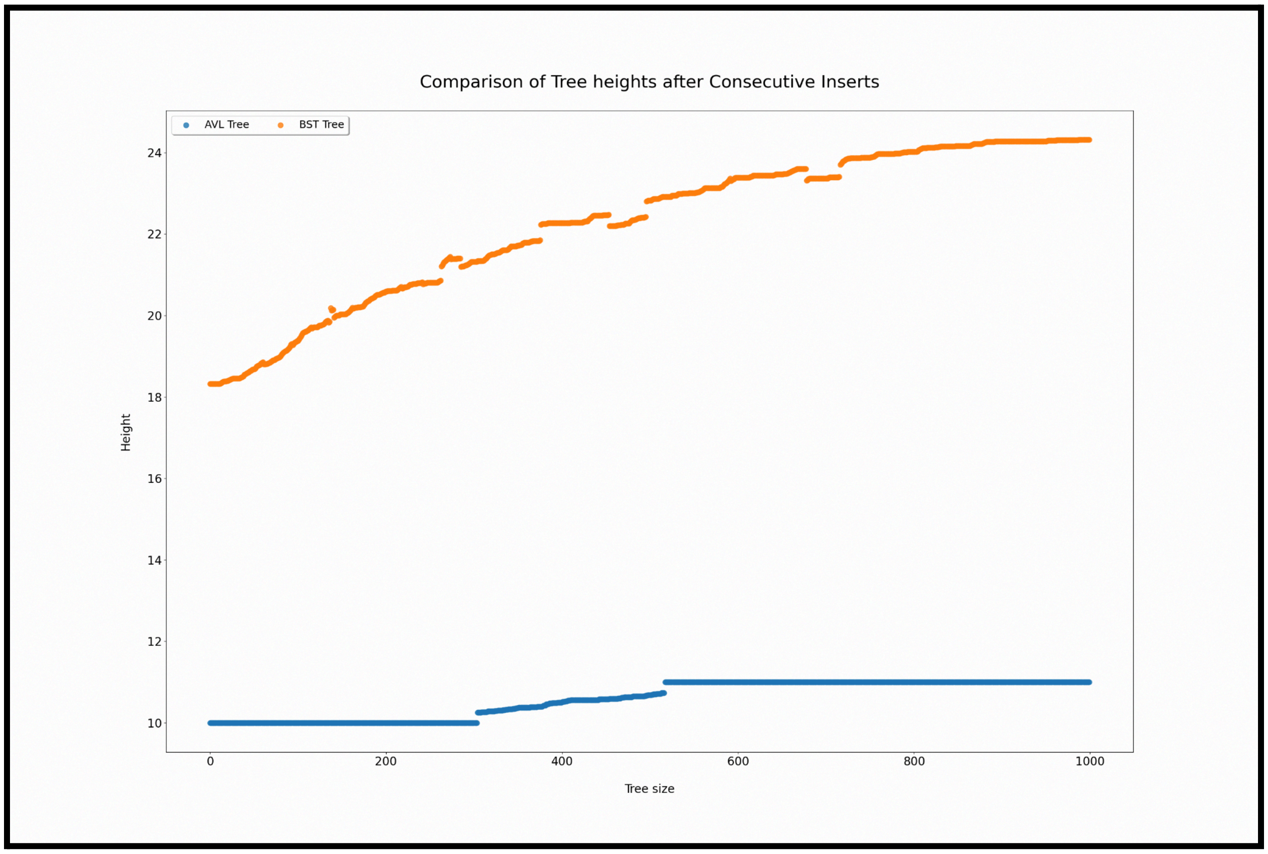
Taking the implementation of the Binary Search tree in Problem 04, I extend it with the 'BSTIso' class. In finding if two Trees are isomorphic, I have created the method isIsomorphic(), which takes in two root Nodes of trees and recursively searches through them. At each Node, it's checked whether the current Node of both trees are equal. If any combination of two of a node's children exists for the corresponding node of the other tree, it's considered to still be isomorphic. This continues until either a non-corresponding node combination is reached, or the entire tree is
iterated through. It is my assumption then that the complexity of this algorithm is O(min(m,n)), where m and n are the number of nodes in the two trees. This is because in the worst case, the algorithm needs to traverse all nodes in both trees. However, the traversal stops as soon as a non-isomorphic structure is found, so if one tree is significantly smaller than the other, the algorithm will not need to traverse all nodes in the larger tree.



```java
    public static boolean isIsomorphic(BSTNode root1, BSTNode root2) {
        /**
         * If both roots are null, return true.
         */
        if (root1 == null && root2 == null) return true;
        /**
         * If only one is null, return false.
         */
        if (root1 == null || root2 == null) return false;

        /**
         * If the values are not the same, return false.
         */
        if (root1.value != root2.value) return false;

        /**
         * Check all combinations of left and right nodes for the wo input nodes.
         * We don't need to explicitly return True because that will happen when a leaf is reached.
         * Either the nodes are exactly the same, or they are swapped.
         */
        return (isIsomorphic(root1.left, root2.left) && isIsomorphic(root1.right, root2.right) ||
                isIsomorphic(root1.left, root2.right) && isIsomorphic(root1.right, root2.left));
    }
}
```

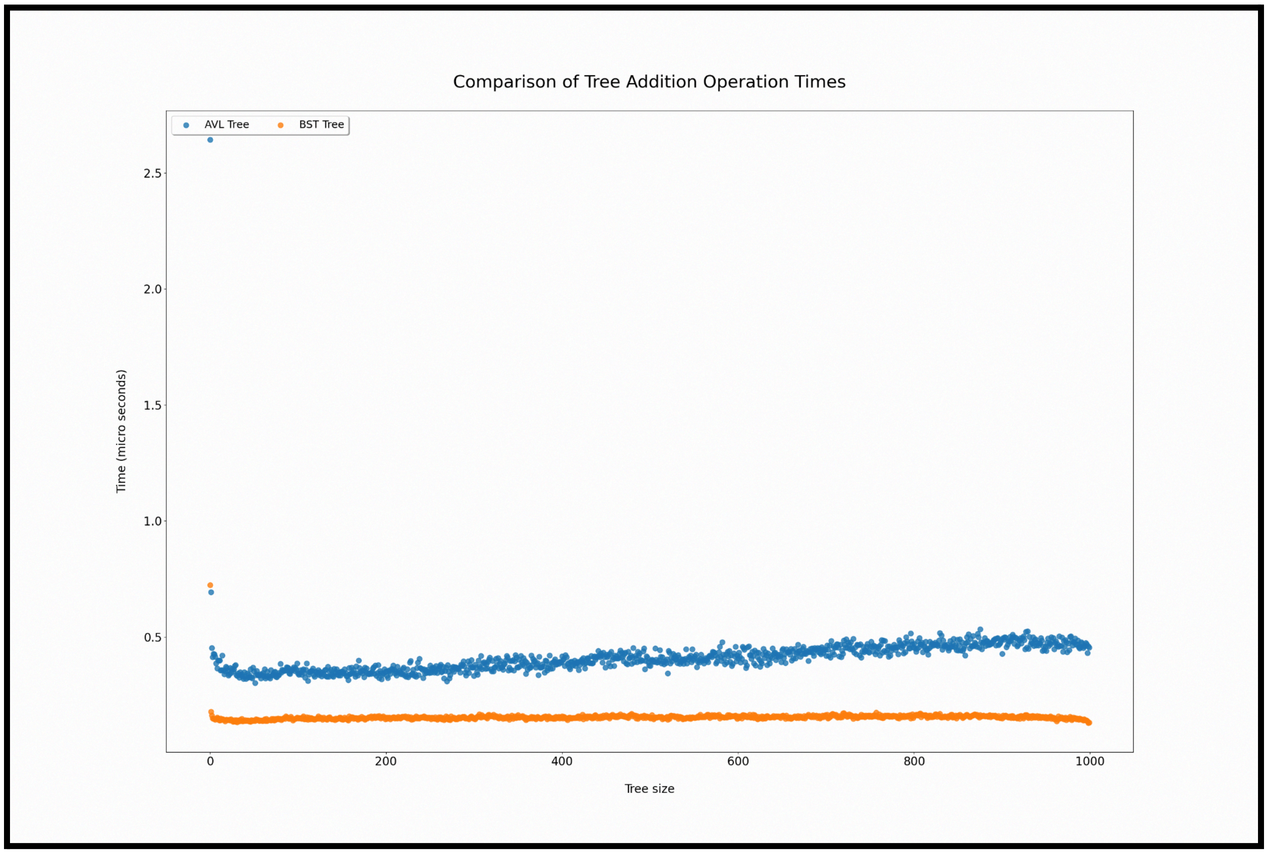
Problem 06

To compare BST and AVL trees, I run a test of 60 samples where I generate randomized trees of each type and given size, then insert a number of randomly generated values into them. The height of the trees can be seen in the following graph, illustrating the difference in depth during insertion.



The self balancing nature of the AVL tree is on display here, retaining a much lower height and growth rate in comparison to the BST. While the BST climbs from a height of 16 to 24 after 100 insertions, the AVL's height only increases by 1, from 10 to 11.

I also looked at the time taken for the number of insertions in random trees, as seen below.

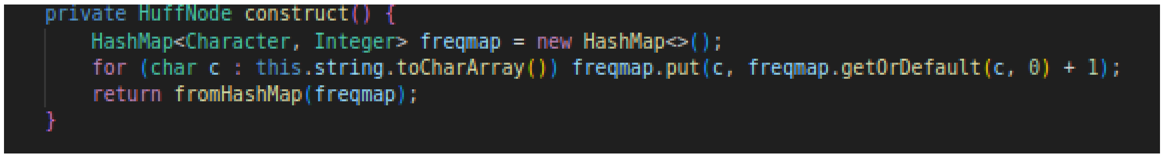


We can see that the AVL tree here takes a lot more time per insertion comparatively. This however is understandable, as for every insertion, the tree must balance, which involves more operations as the number of items in each subtree grows.

From the results discussed, it becomes obvious that these trees are very differently suited for different scenarios. The AVL tree is suited for quick searching and retrieval of data, whilst frequent insertions and deletions are costly and should be avoided. While the BST lacks the self balancing property, insertion and deletion are much faster, but searching may suffer due to deepening subtrees over time.

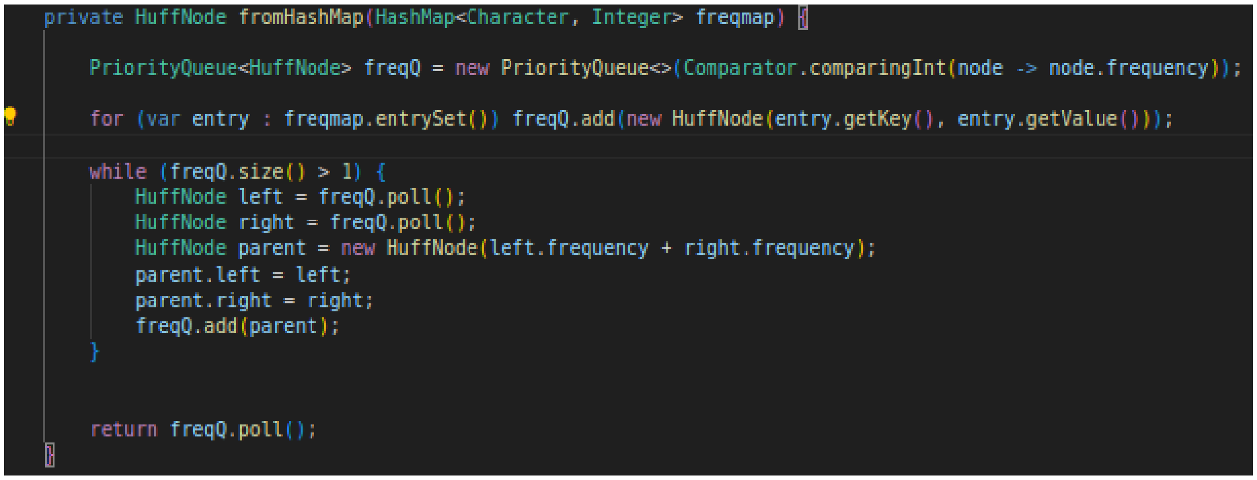
Problem 07

I implemented a Huffman Tree in the simplest form I could. Characters are stored in Node objects with the key and value, as well as their left and right children. A string is read and the frequencies are noted in a hashmap.



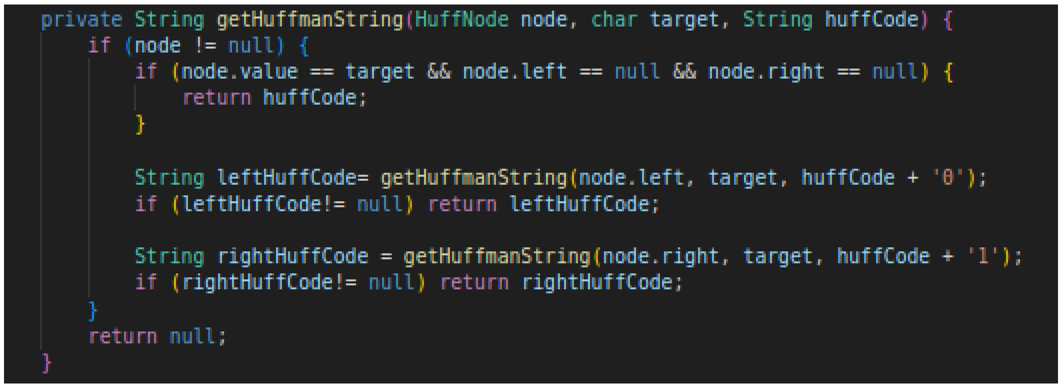
```
private HuffNode construct() {
    HashMap<Character, Integer> freqmap = new HashMap<>();
    for (char c : this.string.toCharArray()) freqmap.put(c, freqmap.getOrDefault(c, 0) + 1);
    return fromHashMap(freqmap);
}
```

From here, I am creating a priority queue of the nodes via the frequency for easy handling. Left and right siblings are popped off and a parent is made for them with the sum of their frequencies.



```
private HuffNode fromHashMap(HashMap<Character, Integer> freqmap) {

    PriorityQueue<HuffNode> freqQ = new PriorityQueue<>(Comparator.comparingInt(node -> node.frequency));

    for (var entry : freqmap.entrySet()) freqQ.add(new HuffNode(entry.getKey(), entry.getValue()));

    while (freqQ.size() > 1) {
        HuffNode left = freqQ.poll();
        HuffNode right = freqQ.poll();
        HuffNode parent = new HuffNode(left.frequency + right.frequency);
        parent.left = left;
        parent.right = right;
        freqQ.add(parent);
    }


    return freqQ.poll();
}
```

The final, most frequency node is returned to us as our root node. It becomes trivial then to walk our tree and either add a ‘0’ or ‘1’ depending on us going left or finding an item.



```
private String getHuffmanString(HuffNode node, char target, String huffCode) {
    if (node != null) {
        if (node.value == target && node.left == null && node.right == null) {
            return huffCode;
        }

        String leftHuffCode= getHuffmanString(node.left, target, huffCode + '0');
        if (leftHuffCode!= null) return leftHuffCode;

        String rightHuffCode = getHuffmanString(node.right, target, huffCode + '1');
        if (rightHuffCode!= null) return rightHuffCode;
    }
    return null;
}
```