

**Data Structures and Algorithms 2**

**Course Project 2023**

Matthias Bartolo\* (0436103L)

\*B.Sc. It (Hons) Artificial Intelligence (Second Year)

Study-unit: **Data Structures and Algorithms 2**

Code: **ICS2210**

Lecturers: **Dr Guillaumier & Dr Abela**

## Introduction

Data Structures are known as one of the key components, and still an active area of research in Computer Science. There are various types of data structures, which also vary in usage, the tree data structure is one of the most common data structures in today’s age. Furthermore, there exists more than one type of tree data structures, for example: Binary Search Trees, AVL Trees and Red-Black Trees. However, one might wonder which of the following trees is superior, in different contexts?

**What is an unbalanced Binary Search Tree?**

An unbalanced Binary Search Tree (BST) is a binary tree, whereby every node in the tree can have a maximum of up to two children. Furthermore, every node in the tree, abides by the rule that all the nodes in the left Subtree have a smaller value than the current node value, similarly, all the nodes in the right Subtree have a larger value than the current node value. Moreover, a BST does not have any balance condition, and thus degenerate.

**What is an AVL Tree?**

An AVL Tree is a BST, whereby every node in the tree, also abides by a balance condition. The balance condition states that the height of the left Subtree and the height of the right Subtree does not differ by no more than one. Upon Insertion or Deletion of nodes in the AVL Tree, if the balance condition is violated, the AVL Tree will perform a relevant rotation, to rebalance the tree. Moreover, since the AVL Tree has this balance condition in place, the tree will not degenerate, when compared to BST.

**What is a Red-Black Tree?**

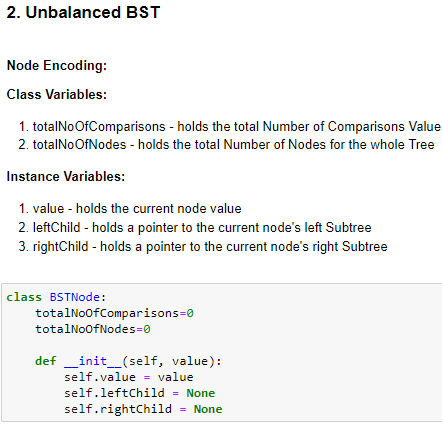
A Red-Black Tree (RBT) is a binary tree, whereby every node in the tree, also contains an extra property, which denotes the node’s colour. Properties of an RBT include that:

1. Nodes in the RBT would either have a red or black colour.
2. The root node of the RBT would have a black colour.
3. Children of red nodes would have a black colour.
4. Null Leaves would have a black colour.
5. Paths from any node to Null Leaves, would have the same number of black nodes.

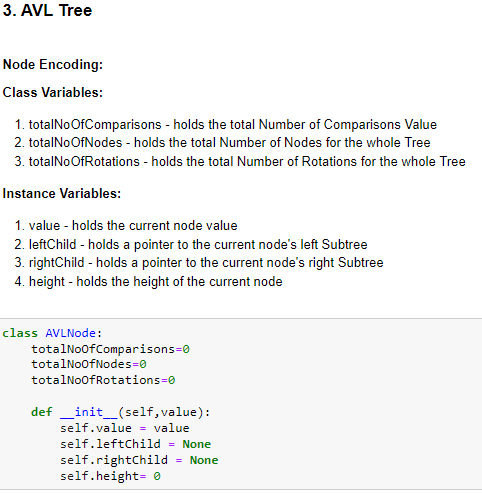
Additionally, upon Insertion or Deletion of nodes in the RBT, if one of the previously mentioned properties is violated, the RBT would either perform a recolouring or a relevant rotation, to maintain the RBT’s integrity.

## Implementation

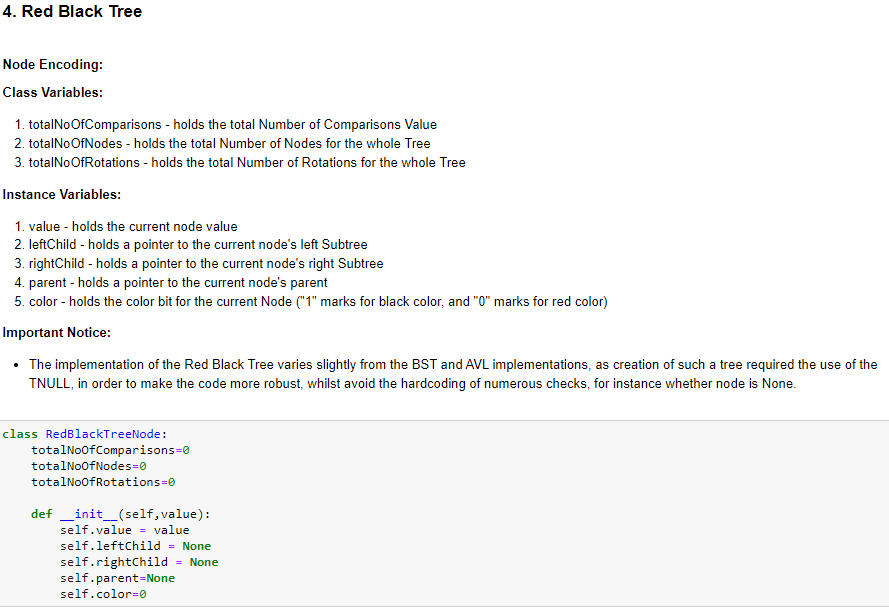
The previously mentioned trees were all implemented in a jupyter notebook. Implementation of each tree node can be observed through Figures 1 to 3.



**Figure 1: Node Encoding of BST**

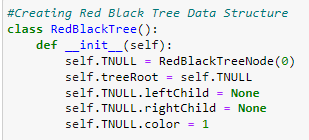


**Figure 2: Node Encoding of AVL Tree**



**Figure 3: Node Encoding of RBT**

Furthermore, implementation of the Red-Black Tree data structure required the creation of a new **RedBlackTree** class, to keep track of the Tree Root. Nevertheless, such implementation also utilises the TNULL type to make the code more robust, whilst to avoid the hardcoding of numerous checks, for instance whether node is None. This implementation of the Red-Black Tree class can be observed in Figure 4.



**Figure 4: RedBlackTree class implementation**

Implementation of Insert and Delete operations for each tree, were frivolous, however the RBT Insertion and Deletion required a relevant method to Repair the Tree after the respective operation. Note that the AVL Tree implementation also required a method, which checks whether the tree requires rebalancing after an insertion or deletion operation, and if so, would perform a relevant rotation.

The **RBT-RepairTreeInsert** method, checks whether there are any red, red violations in the tree, and if so will apply one of the following fix cases, until the tree does not incur any more red, red violations. Implementation of this method can be viewed in Figure 5 and was inspired from [1-2].

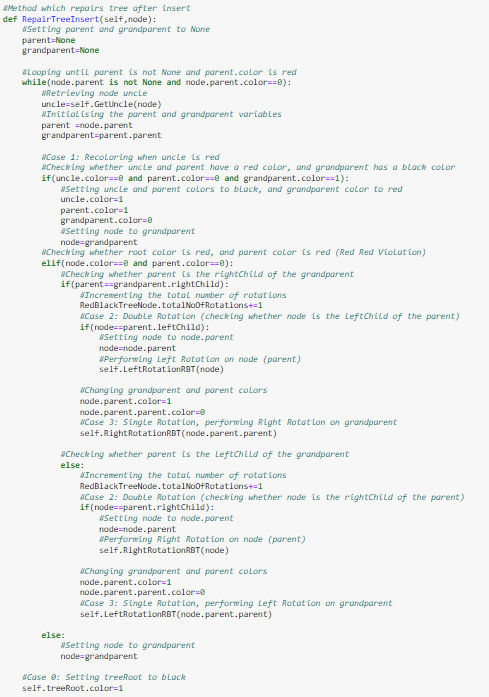
The Insert Fix Cases consist of the following:

1. Case 1: Setting tree Root to black.
2. Case 2: Recolouring when uncle is red.
3. Case 3: Performing Recolouring and Double Rotation when uncle is black.
4. Case 4: Performing Recolouring and Single Rotation when uncle is black.

The **RBT-RepairTreeDelete** method, is a little more complex than the RepairTreeInsert method. Moreover, this method checks whether the deleted node is replaced by a black child, and thus such child is marked as double black. The function continues to loop whilst applying one of the following fix cases, until the double black is converted into a single black. Implementation of this method can be viewed in Figure 6 and was inspired from [1-2].

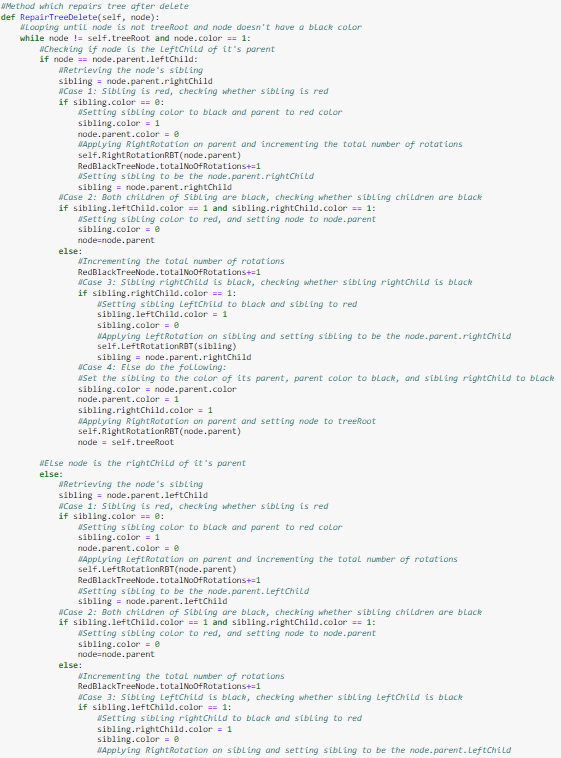
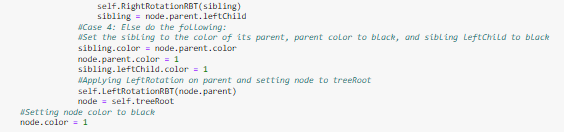
The Delete Fix Cases consist of the following:

1. Case 1: Performing Recolouring and Single Rotation when sibling is red.
2. Case 2: Recolouring when both sibling children are black.
3. Case 3: Performing Recolouring and Double Rotation when sibling is black, and sibling right Child is black.
4. Case 4: Performing Recolouring and Single Rotation when sibling is black.



**Figure 5: RepairTreeInsert**

**method**



**Figure 6: RepairTreeDelete**

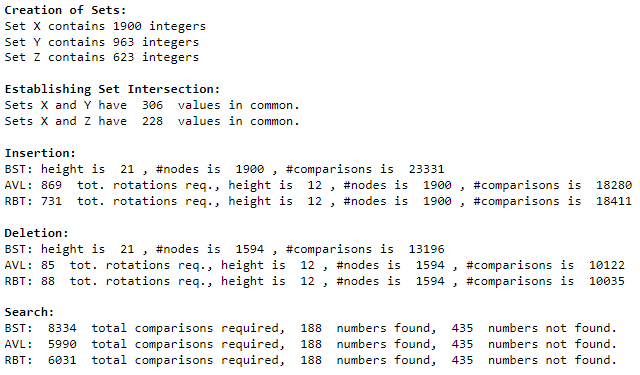
**method**

## Hypothesis and Comparisons of Output

Prior to generating the test cases below, the following hypothesis (expected outcome) was formed:” The RBT and the AVL Tree will perform better than the BST, as a BST does not have a balance condition in place, and thus worst-case height of BST will be **O(n)**. Nevertheless, the AVL Tree and the RBT will perform similarly, however the AVL Tree will sometimes outperform the RBT with respect to tree traversal, as according to [3], the AVL Tree has a worst-case height of **O(1.44\*log2n)**, whilst the RBT has a worst-case height of **O(2\*log2n)**. “

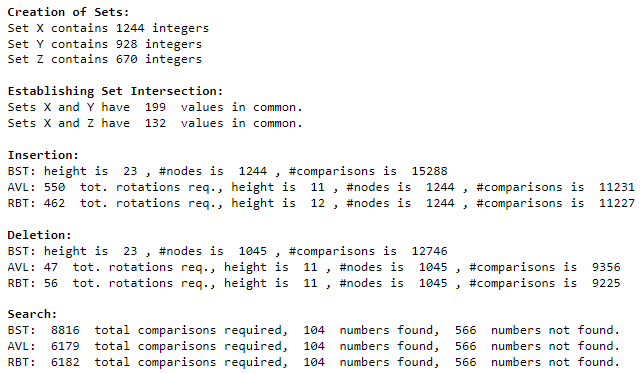
The following test cases, shown in Figures 7-17, were generated, when running the program multiple times.

**Test Case 1:**



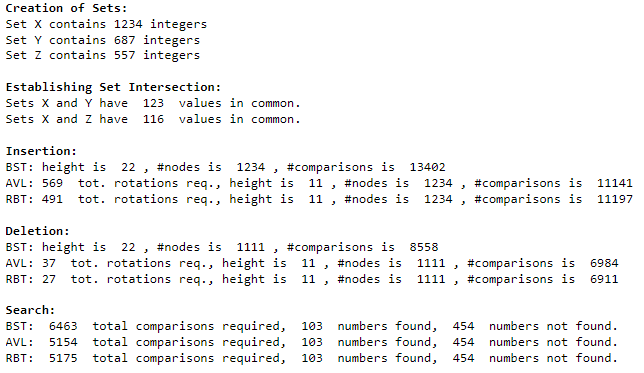
**Figure 7: Test Case 1 Results**

**Test Case 2:**



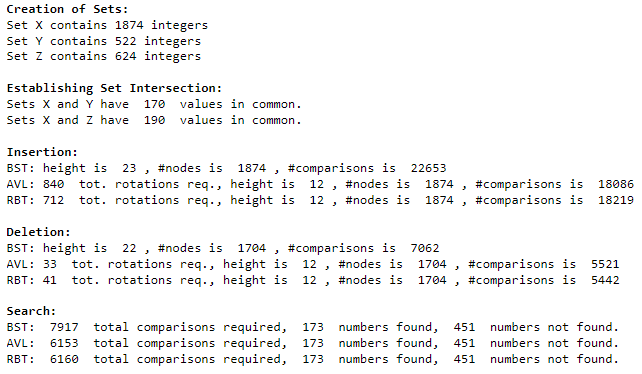
**Figure 8: Test Case 2 Results**

**Test Case 3:**



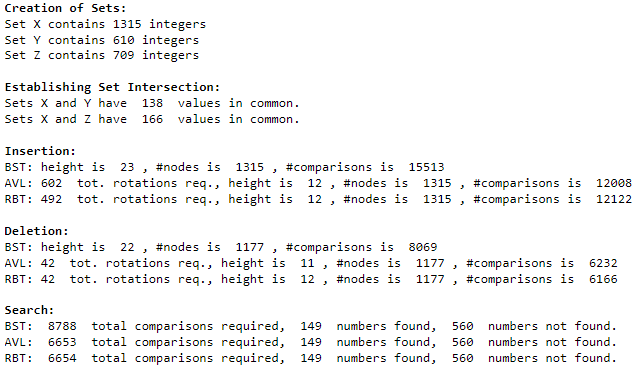
**Figure 9: Test Case 3 Results**

**Test Case 4:**



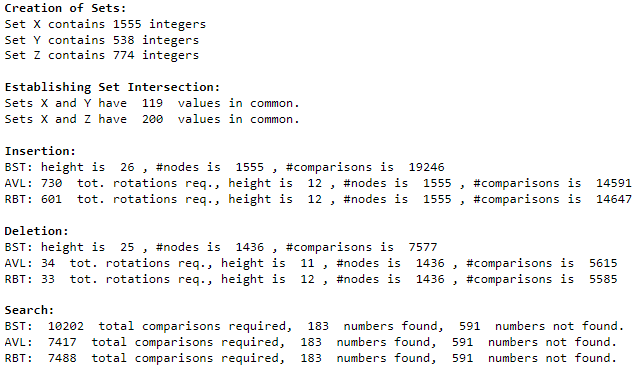
**Figure 10: Test Case 4 Results**

**Test Case 5:**



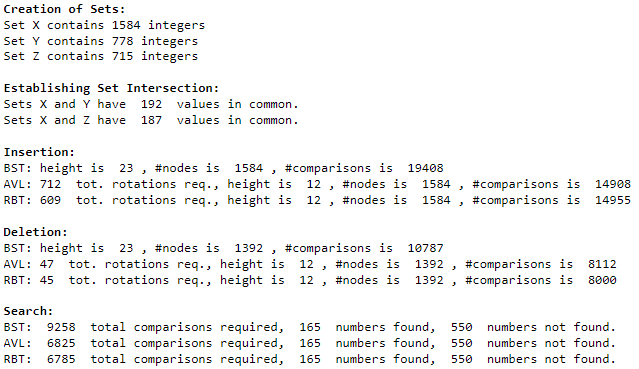
**Figure 11: Test Case 5 Results**

**Test Case 6:**



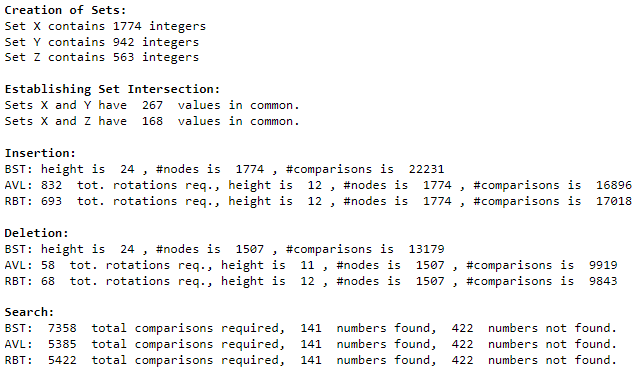
**Figure 12: Test Case 6 Results**

**Test Case 7:**



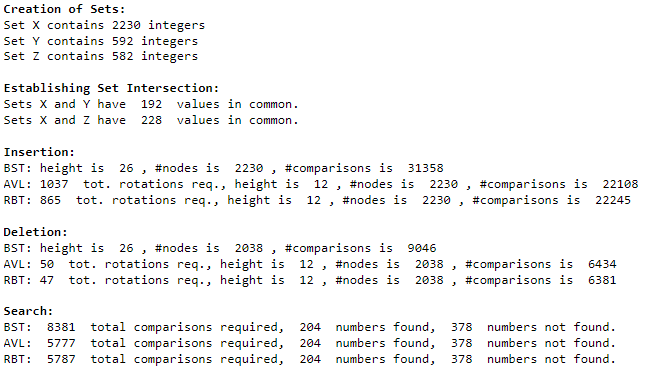
**Figure 13: Test Case 7 Results**

**Test Case 8:**



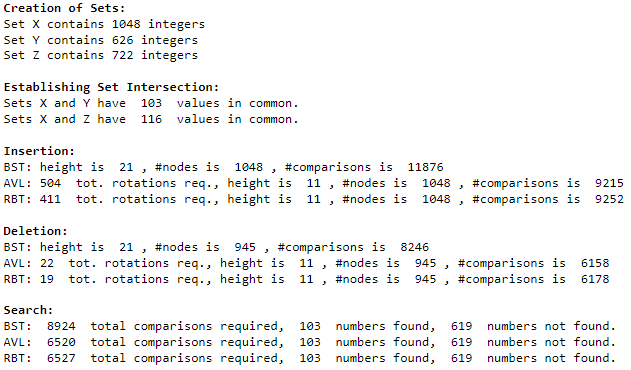
**Figure 14: Test Case 8 Results**

**Test Case 9:**



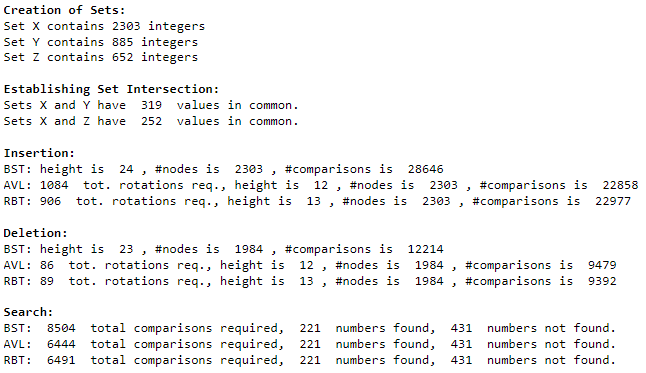
**Figure 15: Test Case 9 Results**

**Test Case 10:**



**Figure 16: Test Case 10 Results**

**Test Case 11:**



**Figure 17: Test Case 11 Results**

Observing the result above, one can conclude that the hypothesis above generally holds, however some also interesting patterns were discovered.

When comparing the **Insertion** operation for all the Trees, it was observed that the total number of comparisons and height of the AVL Tree was less than the BST and the RBT. Furthermore, the total number of comparisons and height of RBT also seemed to be quite similar to the AVL Tree. However, it was also interesting to note that the total number of rotations for RBT were smaller than those in the AVL Tree.

When comparing the **Deletion** operation for all the Trees, it was generally observed that the total number of comparisons and the total number of rotations in the RBT was less than the BST and the AVL Tree. Furthermore, the height of RBT also seemed to be quite similar to the AVL Tree, being that the AVL Tree would have the least height of all the Trees. However, it was also interesting to note that sometimes, the AVL Tree would have less rotations when compared to the RBT, although the difference observed in this case was quite minimal. This can be observed in Test Cases 4 and 8.

When comparing the **Search** operation for all the Trees, it was observed that the total number of comparisons for the AVL Tree was less than the BST and the RBT. However, the total number of comparisons of the RBT also seemed to be quite similar to the AVL Tree.

## Conclusion

From the above comparisons, one can deduce that in order to prioritize Insertion and Deletion, an RBT should be used, since it would have a similar height to an AVL Tree but less rotations. On the other hand, in order to prioritize searching, an AVL Tree should be used, since it is more balanced than an RBT, and thus traversing the tree would be faster. It was also deduced, that a BST proved to be quite inefficient when compared to the AVL Tree and the RBT, since the unbalanced BST would degenerate.

Thus, Real-world applications for each tree can be the following:

**Unbalanced Binary Search Tree** – utilised in education to teach students how to implement a simple tree data structure. However, such tree should not be used in other real-life applications, as such tree is shown to be inefficient.

**AVL Tree** – utilised over databases, as searching through an AVL Tree is proven to be the most optimal when compared to the other trees mentioned above [4]. Moreover, databases are constantly being queried by many users, and thus utilising such tree would be quite beneficial.

**Red-Black Tree** – utilised over sets or hash maps, as insertion or deletion through an RBT is proven to be the most optimal when compared to the other trees mentioned above [4]. Furthermore, sets or hash maps are constantly being modified by either adding or removing values from the set, and therefore using such tree would be fairly advantageous.

## References

[1] T. Cormen, C. Leiserson, R. Rivest, C. Stein,” Red-Black Trees”, Introduction to Algorithms Third Edition, 2009 [Online]. Available: <https://sd.blackball.lv/library/Introduction_to_Algorithms_Third_Edition_(2009).pdf> .[Accessed: 22- Mar- 2022]

[2] S. Woltman, “Red-Black Tree (FULLY EXPLAINED, WITH JAVA CODE)”, HappyCoder.eu, 2021 [Online]. Available: <https://www.happycoders.eu/algorithms/red-black-tree-java/>. [Accessed: 22- Mar- 2022]

[3] K. Guillaumeir, ICS2210: Red-Black Trees, 2022 [Online]. Available: <https://www.um.edu.mt/vle/pluginfile.php/1176939/mod_resource/content/1/Red-Black%20Trees.pdf>. [Accessed: 22- Mar- 2022]

[4] U. Gupta, “Red Black Tree VS AVL Tree”, CodingNinjas.com 2023 [Online]. Available: <https://www.codingninjas.com/codestudio/library/red-black-tree-vs-avl-tree>. [Accessed: 22- Mar- 2022]

## Statement of Completion

|  |  |
| --- | --- |
| **Item** | **Completed (Yes/No/Partial)** |
|  | |
| Created sets X, Y, and Z without duplicates and showing intersections. | **Yes** |
| AVL tree insert | **Yes** |
| AVL tree delete | **Yes** |
| AVL tree search | **Yes** |
| RB tree insert | **Yes** |
| RB tree delete | **Yes** |
| RB tree search | **Yes** |
| Unbalanced BST insert | **Yes** |
| Unbalanced BST delete | **Yes** |
| Unbalanced BST search | **Yes** |
| Discussion comparing tree data structures | **Yes** |
| *If partial, explain what has been done* | |

## Plagiarism Declaration Form