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?

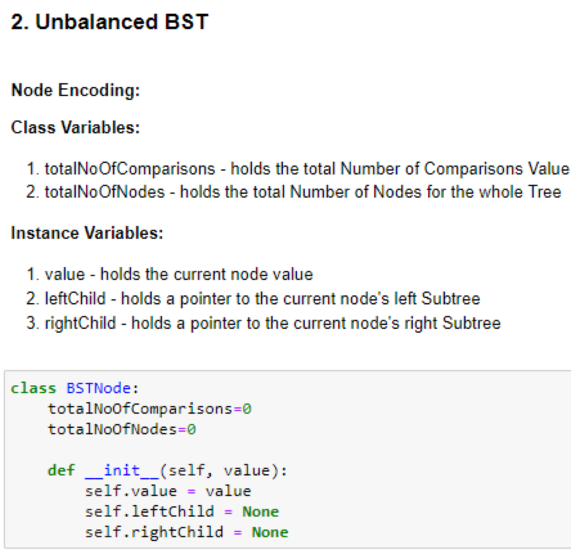
A Red-Black Tree (RBT) is a binary tree, whereby every node in the tree, also contains an extra property, which denotes the node's colour. Properties of an RBT include that:

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.



2. Unbalanced BST

Node Encoding:

Class Variables:

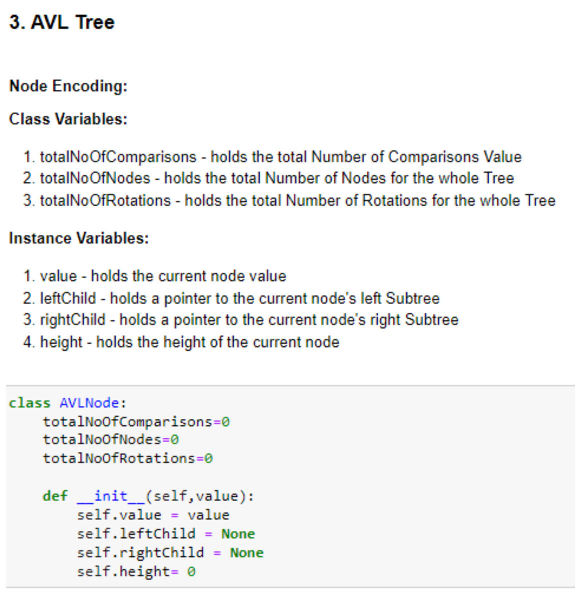
1. totalNoOfComparisons - holds the total Number of Comparisons Value
2. totalNoOfNodes - holds the total Number of Nodes for the whole Tree

Instance Variables:

1. value - holds the current node value
2. leftChild - holds a pointer to the current node's left Subtree
3. rightChild - holds a pointer to the current node's right Subtree

```
class BSTNode:
    totalNoOfComparisons=0
    totalNoOfNodes=0

    def __init__(self, value):
        self.value = value
        self.leftChild = None
        self.rightChild = None
```

Figure 1: Node Encoding of BST



3. AVL Tree

Node Encoding:

Class Variables:

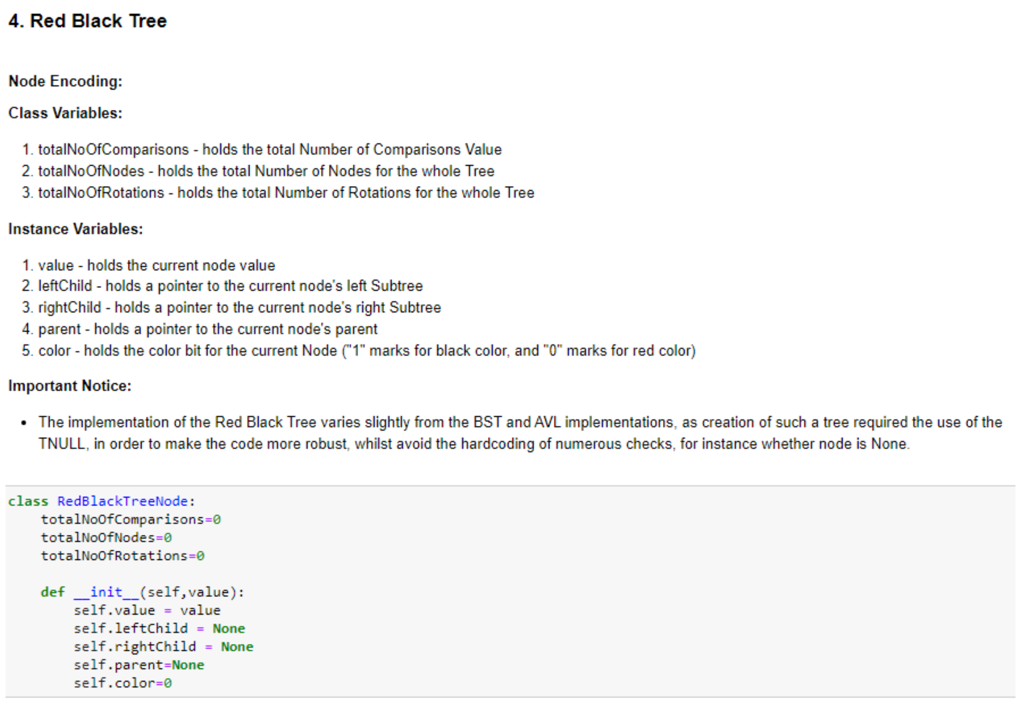
1. totalNoOfComparisons - holds the total Number of Comparisons Value
2. totalNoOfNodes - holds the total Number of Nodes for the whole Tree
3. totalNoOfRotations - holds the total Number of Rotations for the whole Tree

Instance Variables:

1. value - holds the current node value
2. leftChild - holds a pointer to the current node's left Subtree
3. rightChild - holds a pointer to the current node's right Subtree
4. height - holds the height of the current node

```
class AVLNode:
    totalNoOfComparisons=0
    totalNoOfNodes=0
    totalNoOfRotations=0

    def __init__(self,value):
        self.value = value
        self.leftChild = None
        self.rightChild = None
        self.height= 0
```

Figure 2: Node Encoding of AVL Tree



4. Red Black Tree

Node Encoding:

Class Variables:

1. totalNoOfComparisons - holds the total Number of Comparisons Value
2. totalNoOfNodes - holds the total Number of Nodes for the whole Tree
3. totalNoOfRotations - holds the total Number of Rotations for the whole Tree

Instance Variables:

1. value - holds the current node value
2. leftChild - holds a pointer to the current node's left Subtree
3. rightChild - holds a pointer to the current node's right Subtree
4. parent - holds a pointer to the current node's parent
5. color - holds the color bit for the current Node ("1" marks for black color, and "0" marks for red color)

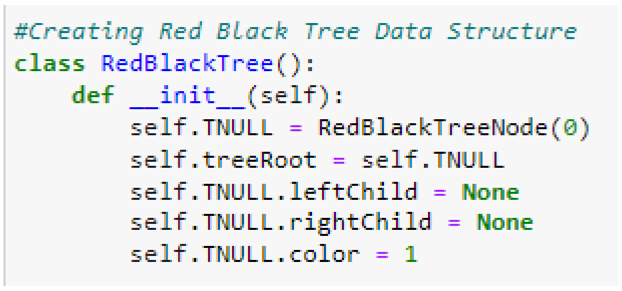
Important Notice:

- The implementation of the Red Black Tree varies slightly from the BST and AVL implementations, as creation of such a tree required the use of the TNULL, in order to make the code more robust, whilst avoid the hardcoding of numerous checks, for instance whether node is None.

```
class RedBlackTreeNode:
    totalNoOfComparisons=0
    totalNoOfNodes=0
    totalNoOfRotations=0

    def __init__(self,value):
        self.value = value
        self.leftChild = None
        self.rightChild = None
        self.parent=None
        self.color=0
```

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.



```python
#Creating Red Black Tree Data Structure
class RedBlackTree():
    def __init__(self):
        self.TNULL = RedBlackTreeNode(0)
        self.treeRoot = self.TNULL
        self.TNULL.leftChild = None
        self.TNULL.rightChild = None
        self.TNULL.color = 1
```

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].

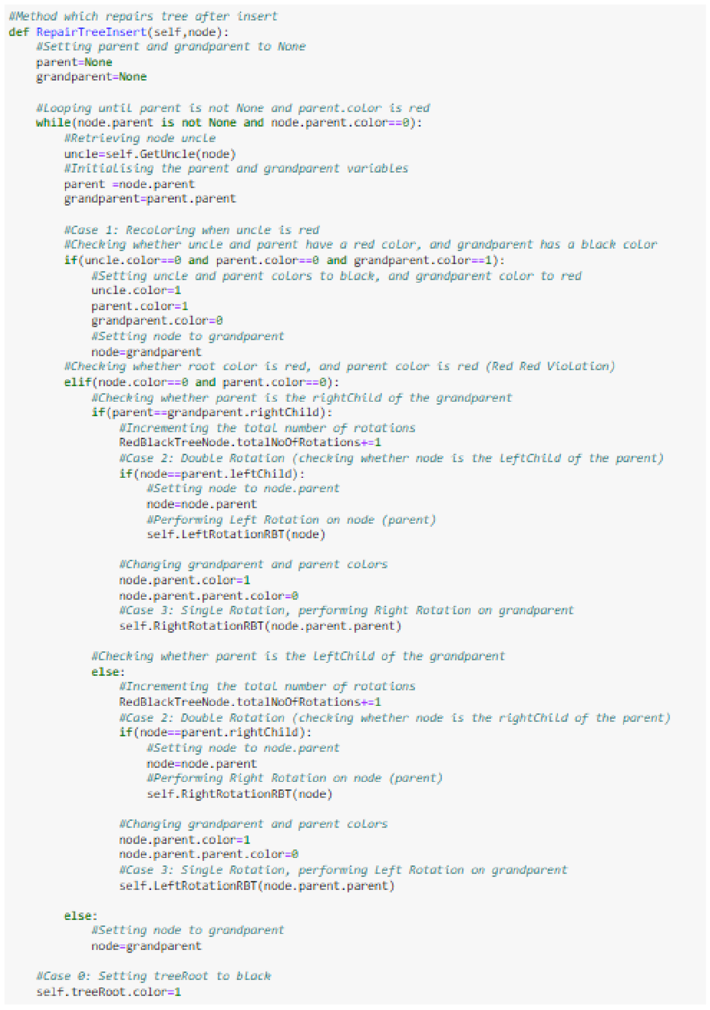
The Insert Fix Cases consist of the following:

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].

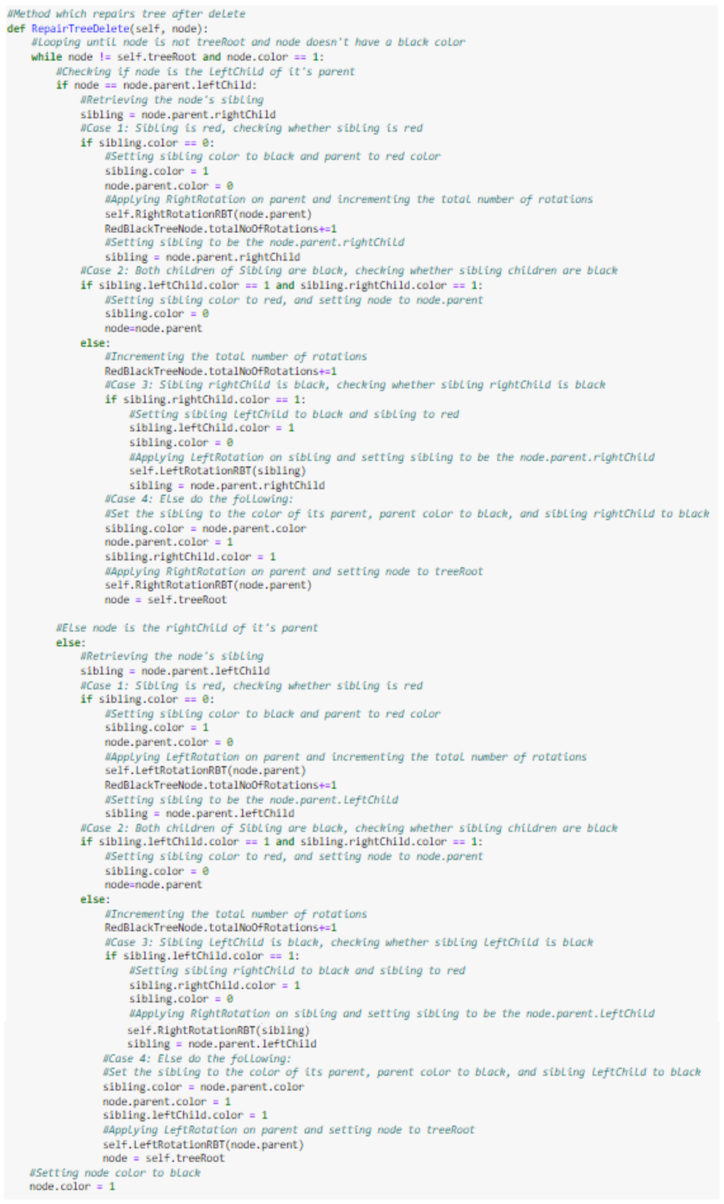
The Delete Fix Cases consist of the following:

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.



```
#Method which repairs tree after insert
def RepairTreeInsert(self,node):
    #Setting parent and grandparent to None
    parent=None
    grandparent=None

    #Looping until parent is not None and parent.color is red
    while(node.parent is not None and node.parent.color==0):
        #Retrieving node uncle
        uncle=self.GetUncle(node)
        #Initialising the parent and grandparent variables
        parent =node.parent
        grandparent=parent.parent

        #Case 1: Recoloring when uncle is red
        #Checking whether uncle and parent have a red color, and grandparent has a black color
        if(uncle.color==0 and parent.color==0 and grandparent.color==1):
            #Setting uncle and parent colors to black, and grandparent color to red
            uncle.color=1
            parent.color=1
            grandparent.color=0
            #Setting node to grandparent
            node=grandparent
        #Checking whether root color is red, and parent color is red (Red Red Violation)
        elif(node.color==0 and parent.color==0):
            #Checking whether parent is the rightChild of the grandparent
            if(parent==grandparent.rightChild):
                #Incrementing the total number of rotations
                RedBlackTreeNode.totalNoOfRotations+=1
                #Case 2: Double Rotation (checking whether node is the leftChild of the parent)
                if(node==parent.leftChild):
                    #Setting node to node.parent
                    node=node.parent
                    #Performing Left Rotation on node (parent)
                    self.LeftRotationRBT(node)

                #Changing grandparent and parent colors
                node.parent.color=1
                node.parent.parent.color=0
                #Case 3: Single Rotation, performing Right Rotation on grandparent
                self.RightRotationRBT(node.parent.parent)

            #Checking whether parent is the leftChild of the grandparent
            else:
                #Incrementing the total number of rotations
                RedBlackTreeNode.totalNoOfRotations+=1
                #Case 2: Double Rotation (checking whether node is the rightChild of the parent)
                if(node==parent.rightChild):
                    #Setting node to node.parent
                    node=node.parent
                    #Performing Right Rotation on node (parent)
                    self.RightRotationRBT(node)

                #Changing grandparent and parent colors
                node.parent.color=1
                node.parent.parent.color=0
                #Case 3: Single Rotation, performing Left Rotation on grandparent
                self.LeftRotationRBT(node.parent.parent)

        else:
            #Setting node to grandparent
            node=grandparent

    #Case 0: Setting treeRoot to black
    self.treeRoot.color=1
```

Figure 5: RepairTreeInsert method



```
#Method which repairs tree after delete
def RepairTreeDelete(self, node):
    #Looping until node is not treeRoot and node doesn't have a black color
    while node != self.treeRoot and node.color == 1:
        #Checking if node is the leftChild of it's parent
        if node == node.parent.leftChild:
            #Retrieving the node's sibling
            sibling = node.parent.rightChild
            #Case 1: Sibling is red, checking whether sibling is red
            if sibling.color == 0:
                #Setting sibling color to black and parent to red color
                sibling.color = 1
                node.parent.color = 0
                #Applying RightRotation on parent and incrementing the total number of rotations
                self.RightRotationRBT(node.parent)
                RedBlackTreeNode.totalNoOfRotations+=1
                #Setting sibling to be the node.parent.rightChild
                sibling = node.parent.rightChild
            #Case 2: Both children of Sibling are black, checking whether sibling children are black
            if sibling.leftChild.color == 1 and sibling.rightChild.color == 1:
                #Setting sibling color to red, and setting node to node.parent
                sibling.color = 0
                node=node.parent
            else:
                #Incrementing the total number of rotations
                RedBlackTreeNode.totalNoOfRotations+=1
                #Case 3: Sibling rightChild is black, checking whether sibling rightChild is black
                if sibling.rightChild.color == 1:
                    #Setting sibling leftChild to black and sibling to red
                    sibling.leftChild.color = 1
                    sibling.color = 0
                    #Applying LeftRotation on sibling and setting sibling to be the node.parent.rightChild
                    self.LeftRotationRBT(sibling)
                    sibling = node.parent.rightChild
                #Case 4: Else do the following:
                #Set the sibling to the color of its parent, parent color to black, and sibling rightChild to black
                sibling.color = node.parent.color
                node.parent.color = 1
                sibling.rightChild.color = 1
                #Applying RightRotation on parent and setting node to treeRoot
                self.RightRotationRBT(node.parent)
                node = self.treeRoot

        #Else node is the rightChild of it's parent
        else:
            #Retrieving the node's sibling
            sibling = node.parent.leftChild
            #Case 1: Sibling is red, checking whether sibling is red
            if sibling.color == 0:
                #Setting sibling color to black and parent to red color
                sibling.color = 1
                node.parent.color = 0
                #Applying LeftRotation on parent and incrementing the total number of rotations
                self.LeftRotationRBT(node.parent)
                RedBlackTreeNode.totalNoOfRotations+=1
                #Setting sibling to be the node.parent.leftChild
                sibling = node.parent.leftChild
            #Case 2: Both children of Sibling are black, checking whether sibling children are black
            if sibling.leftChild.color == 1 and sibling.rightChild.color == 1:
                #Setting sibling color to red, and setting node to node.parent
                sibling.color = 0
                node=node.parent
            else:
                #Incrementing the total number of rotations
                RedBlackTreeNode.totalNoOfRotations+=1
                #Case 3: Sibling leftChild is black, checking whether sibling leftChild is black
                if sibling.leftChild.color == 1:
                    #Setting sibling rightChild to black and sibling to red
                    sibling.rightChild.color = 1
                    sibling.color = 0
                    #Applying RightRotation on sibling and setting sibling to be the node.parent.leftChild
                    self.RightRotationRBT(sibling)
                    sibling = node.parent.leftChild
                #Case 4: Else do the following:
                #Set the sibling to the color of its parent, parent color to black, and sibling leftChild to black
                sibling.color = node.parent.color
                node.parent.color = 1
                sibling.leftChild.color = 1
                #Applying LeftRotation on parent and setting node to treeRoot
                self.LeftRotationRBT(node.parent)
                node = self.treeRoot
    #Setting node color to black
    node.color = 1
```

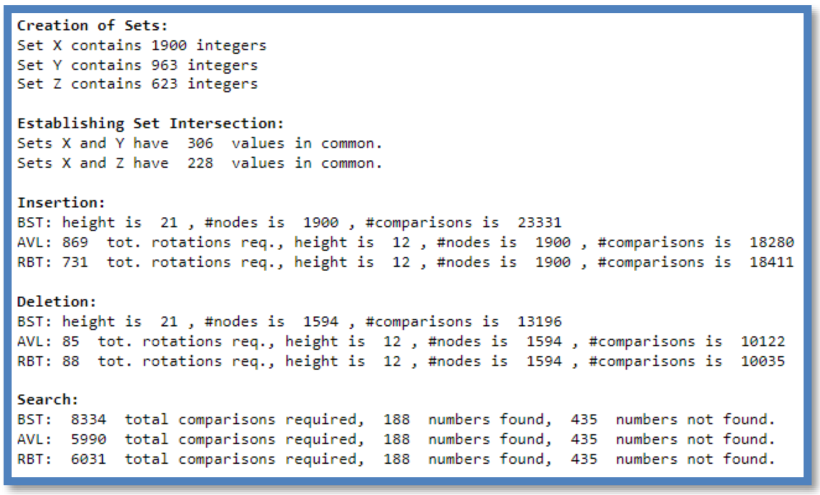
Figure 6: RepairTreeDelete method

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 **$O(1.44*\log_2 n)$**, whilst the RBT has a worst-case height of **$O(2*\log_2 n)$**. "

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

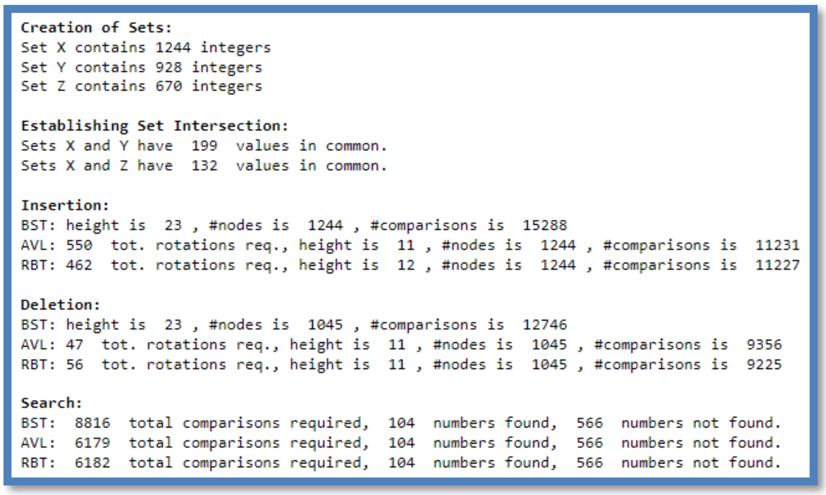
Test Case 1:



```
Creation of Sets:
Set X contains 1900 integers
Set Y contains 963 integers
Set Z contains 623 integers

Establishing Set Intersection:
Sets X and Y have  306  values in common.
Sets X and Z have  228  values in common.

Insertion:
BST: height is  21 , #nodes is  1900 , #comparisons is  23331
AVL: 869  tot. rotations req., height is  12 , #nodes is  1900 , #comparisons is  18280
RBT: 731  tot. rotations req., height is  12 , #nodes is  1900 , #comparisons is  18411

Deletion:
BST: height is  21 , #nodes is  1594 , #comparisons is  13196
AVL: 85  tot. rotations req., height is  12 , #nodes is  1594 , #comparisons is  10122
RBT: 88  tot. rotations req., height is  12 , #nodes is  1594 , #comparisons is  10035

Search:
BST:  8334  total comparisons required,  188  numbers found,  435  numbers not found.
AVL:  5990  total comparisons required,  188  numbers found,  435  numbers not found.
RBT:  6031  total comparisons required,  188  numbers found,  435  numbers not found.
```

Figure 7: Test Case 1 Results

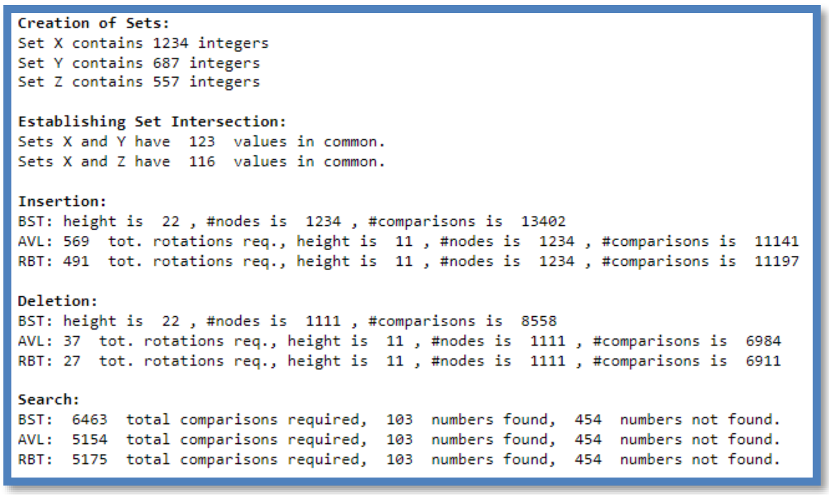
Test Case 2:



```
Creation of Sets:
Set X contains 1244 integers
Set Y contains 928 integers
Set Z contains 670 integers

Establishing Set Intersection:
Sets X and Y have  199  values in common.
Sets X and Z have  132  values in common.

Insertion:
BST: height is  23 , #nodes is  1244 , #comparisons is  15288
AVL: 550  tot. rotations req., height is  11 , #nodes is  1244 , #comparisons is  11231
RBT: 462  tot. rotations req., height is  12 , #nodes is  1244 , #comparisons is  11227

Deletion:
BST: height is  23 , #nodes is  1045 , #comparisons is  12746
AVL: 47  tot. rotations req., height is  11 , #nodes is  1045 , #comparisons is  9356
RBT: 56  tot. rotations req., height is  11 , #nodes is  1045 , #comparisons is  9225

Search:
BST:  8816  total comparisons required,  104  numbers found,  566  numbers not found.
AVL:  6179  total comparisons required,  104  numbers found,  566  numbers not found.
RBT:  6182  total comparisons required,  104  numbers found,  566  numbers not found.
```

Figure 8: Test Case 2 Results

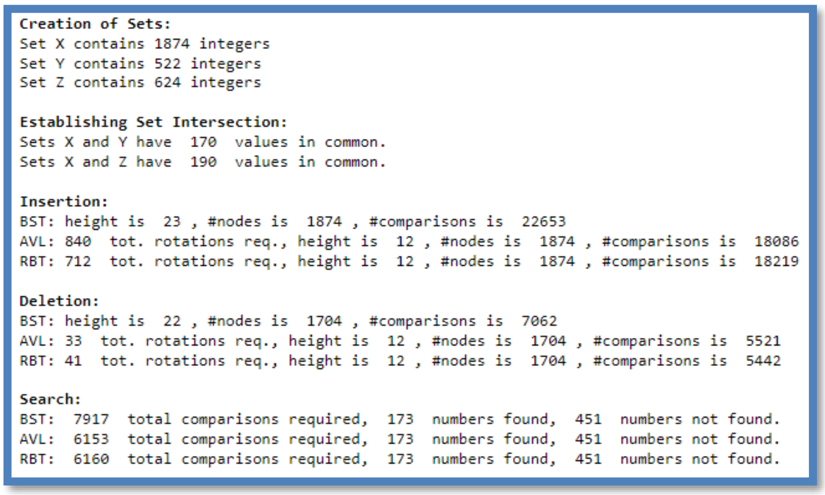
Test Case 3:



```
Creation of Sets:
Set X contains 1234 integers
Set Y contains 687 integers
Set Z contains 557 integers

Establishing Set Intersection:
Sets X and Y have  123  values in common.
Sets X and Z have  116  values in common.

Insertion:
BST: height is  22 , #nodes is  1234 , #comparisons is  13402
AVL: 569  tot. rotations req., height is  11 , #nodes is  1234 , #comparisons is  11141
RBT: 491  tot. rotations req., height is  11 , #nodes is  1234 , #comparisons is  11197

Deletion:
BST: height is  22 , #nodes is  1111 , #comparisons is  8558
AVL: 37  tot. rotations req., height is  11 , #nodes is  1111 , #comparisons is  6984
RBT: 27  tot. rotations req., height is  11 , #nodes is  1111 , #comparisons is  6911

Search:
BST:  6463  total comparisons required,  103  numbers found,  454  numbers not found.
AVL:  5154  total comparisons required,  103  numbers found,  454  numbers not found.
RBT:  5175  total comparisons required,  103  numbers found,  454  numbers not found.
```

Figure 9: Test Case 3 Results

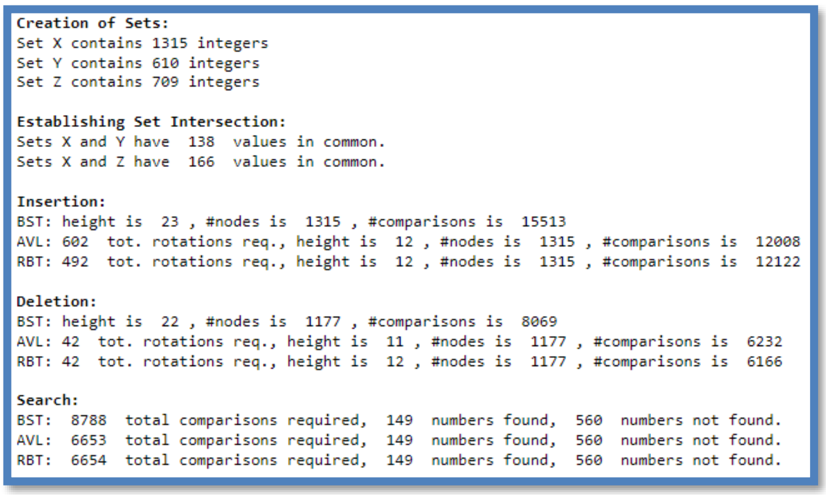
Test Case 4:



```
Creation of Sets:
Set X contains 1874 integers
Set Y contains 522 integers
Set Z contains 624 integers

Establishing Set Intersection:
Sets X and Y have  170  values in common.
Sets X and Z have  190  values in common.

Insertion:
BST: height is  23 , #nodes is  1874 , #comparisons is  22653
AVL: 840  tot. rotations req., height is  12 , #nodes is  1874 , #comparisons is  18086
RBT: 712  tot. rotations req., height is  12 , #nodes is  1874 , #comparisons is  18219

Deletion:
BST: height is  22 , #nodes is  1704 , #comparisons is  7062
AVL: 33  tot. rotations req., height is  12 , #nodes is  1704 , #comparisons is  5521
RBT: 41  tot. rotations req., height is  12 , #nodes is  1704 , #comparisons is  5442

Search:
BST:  7917  total comparisons required,  173  numbers found,  451  numbers not found.
AVL:  6153  total comparisons required,  173  numbers found,  451  numbers not found.
RBT:  6160  total comparisons required,  173  numbers found,  451  numbers not found.
```

Figure 10: Test Case 4 Results

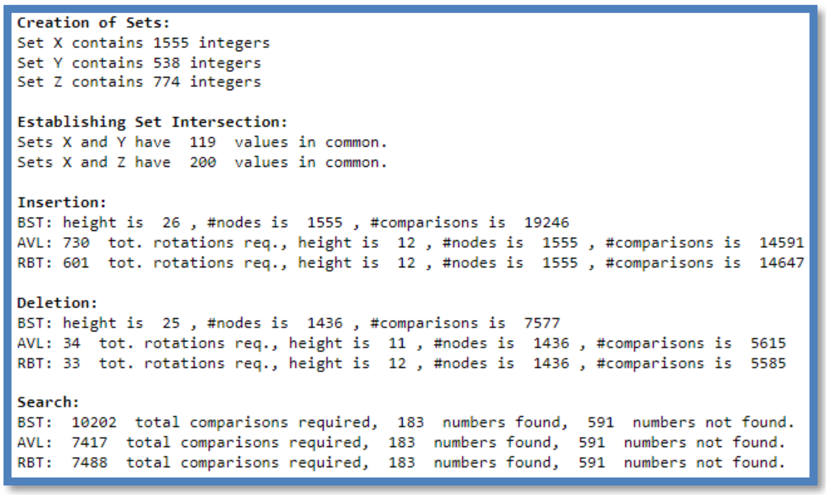
Test Case 5:



```
Creation of Sets:
Set X contains 1315 integers
Set Y contains 610 integers
Set Z contains 709 integers

Establishing Set Intersection:
Sets X and Y have  138  values in common.
Sets X and Z have  166  values in common.

Insertion:
BST: height is  23 , #nodes is  1315 , #comparisons is  15513
AVL: 602  tot. rotations req., height is  12 , #nodes is  1315 , #comparisons is  12008
RBT: 492  tot. rotations req., height is  12 , #nodes is  1315 , #comparisons is  12122

Deletion:
BST: height is  22 , #nodes is  1177 , #comparisons is  8069
AVL: 42  tot. rotations req., height is  11 , #nodes is  1177 , #comparisons is  6232
RBT: 42  tot. rotations req., height is  12 , #nodes is  1177 , #comparisons is  6166

Search:
BST:  8788  total comparisons required,  149  numbers found,  560  numbers not found.
AVL:  6653  total comparisons required,  149  numbers found,  560  numbers not found.
RBT:  6654  total comparisons required,  149  numbers found,  560  numbers not found.
```

Figure 11: Test Case 5 Results

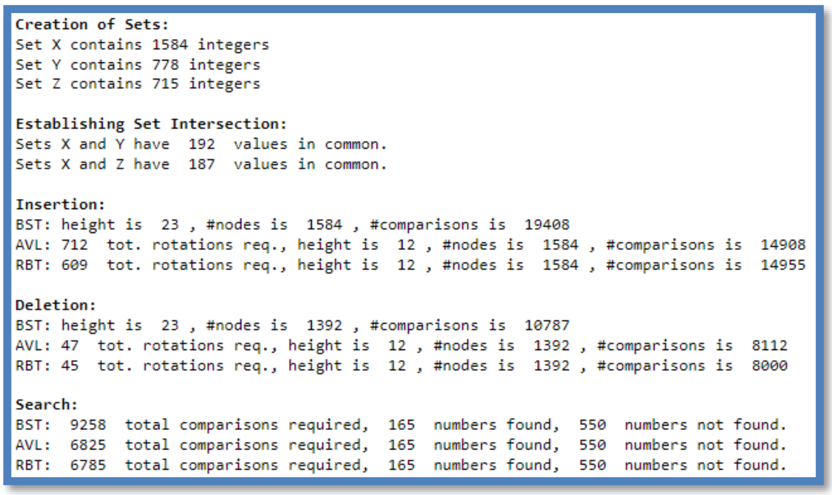
Test Case 6:



```
Creation of Sets:
Set X contains 1555 integers
Set Y contains 538 integers
Set Z contains 774 integers

Establishing Set Intersection:
Sets X and Y have  119  values in common.
Sets X and Z have  200  values in common.

Insertion:
BST: height is  26 , #nodes is  1555 , #comparisons is  19246
AVL: 730  tot. rotations req., height is  12 , #nodes is  1555 , #comparisons is  14591
RBT: 601  tot. rotations req., height is  12 , #nodes is  1555 , #comparisons is  14647

Deletion:
BST: height is  25 , #nodes is  1436 , #comparisons is  7577
AVL: 34  tot. rotations req., height is  11 , #nodes is  1436 , #comparisons is  5615
RBT: 33  tot. rotations req., height is  12 , #nodes is  1436 , #comparisons is  5585

Search:
BST:  10202  total comparisons required,  183  numbers found,  591  numbers not found.
AVL:  7417  total comparisons required,  183  numbers found,  591  numbers not found.
RBT:  7488  total comparisons required,  183  numbers found,  591  numbers not found.
```

Figure 12: Test Case 6 Results

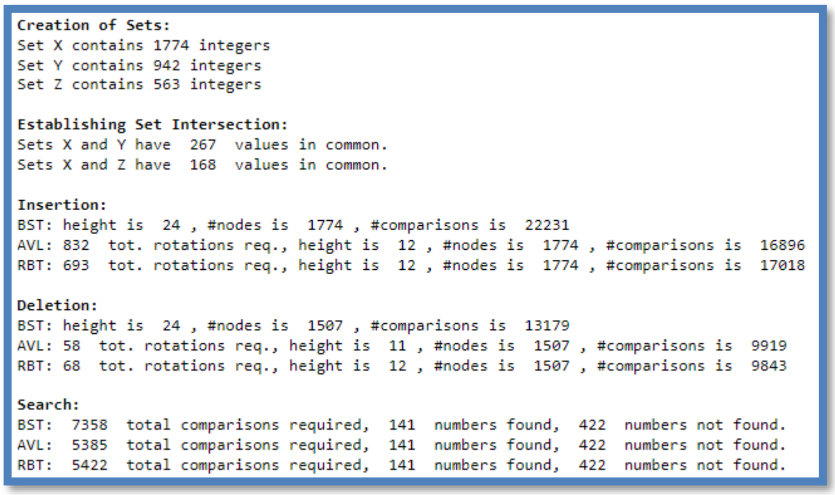
Test Case 7:



```
Creation of Sets:
Set X contains 1584 integers
Set Y contains 778 integers
Set Z contains 715 integers

Establishing Set Intersection:
Sets X and Y have  192  values in common.
Sets X and Z have  187  values in common.

Insertion:
BST: height is  23 , #nodes is  1584 , #comparisons is  19408
AVL: 712  tot. rotations req., height is  12 , #nodes is  1584 , #comparisons is  14908
RBT: 609  tot. rotations req., height is  12 , #nodes is  1584 , #comparisons is  14955

Deletion:
BST: height is  23 , #nodes is  1392 , #comparisons is  10787
AVL: 47  tot. rotations req., height is  12 , #nodes is  1392 , #comparisons is  8112
RBT: 45  tot. rotations req., height is  12 , #nodes is  1392 , #comparisons is  8000

Search:
BST:  9258  total comparisons required,  165  numbers found,  550  numbers not found.
AVL:  6825  total comparisons required,  165  numbers found,  550  numbers not found.
RBT:  6785  total comparisons required,  165  numbers found,  550  numbers not found.
```

Figure 13: Test Case 7 Results

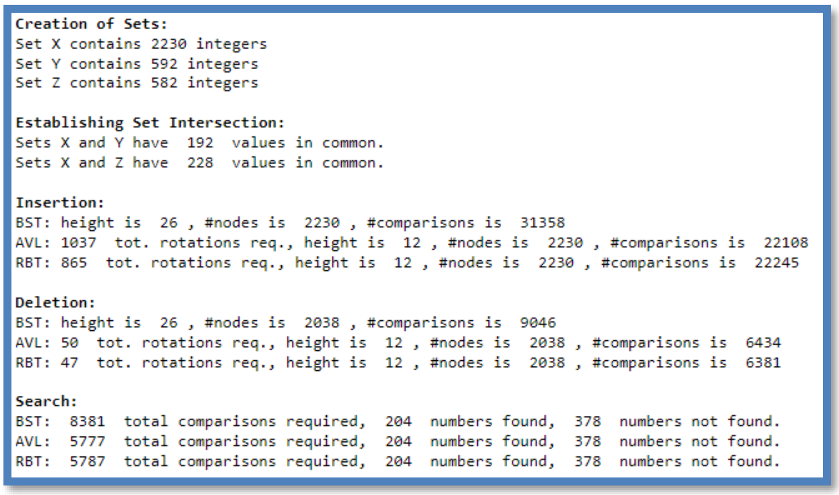
Test Case 8:



```
Creation of Sets:
Set X contains 1774 integers
Set Y contains 942 integers
Set Z contains 563 integers

Establishing Set Intersection:
Sets X and Y have  267  values in common.
Sets X and Z have  168  values in common.

Insertion:
BST: height is  24 , #nodes is  1774 , #comparisons is  22231
AVL: 832  tot. rotations req., height is  12 , #nodes is  1774 , #comparisons is  16896
RBT: 693  tot. rotations req., height is  12 , #nodes is  1774 , #comparisons is  17018

Deletion:
BST: height is  24 , #nodes is  1507 , #comparisons is  13179
AVL: 58  tot. rotations req., height is  11 , #nodes is  1507 , #comparisons is  9919
RBT: 68  tot. rotations req., height is  12 , #nodes is  1507 , #comparisons is  9843

Search:
BST:  7358  total comparisons required,  141  numbers found,  422  numbers not found.
AVL:  5385  total comparisons required,  141  numbers found,  422  numbers not found.
RBT:  5422  total comparisons required,  141  numbers found,  422  numbers not found.
```

Figure 14: Test Case 8 Results

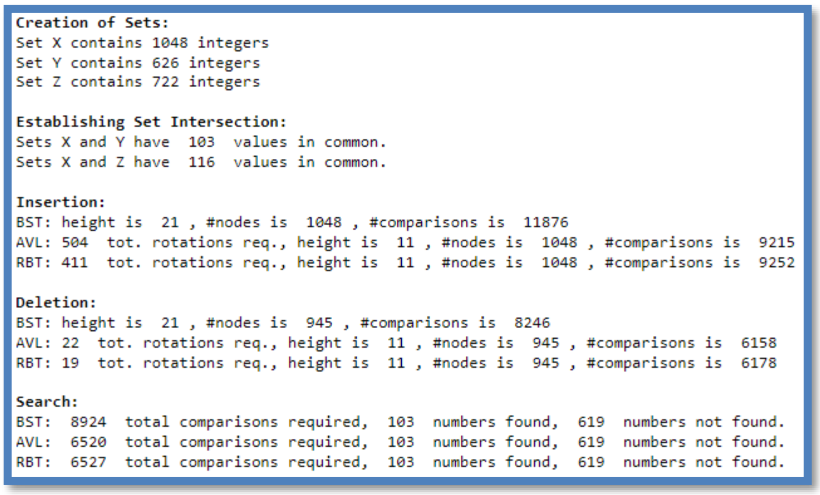
Test Case 9:



```
Creation of Sets:
Set X contains 2230 integers
Set Y contains 592 integers
Set Z contains 582 integers

Establishing Set Intersection:
Sets X and Y have  192  values in common.
Sets X and Z have  228  values in common.

Insertion:
BST: height is  26 , #nodes is  2230 , #comparisons is  31358
AVL: 1037  tot. rotations req., height is  12 , #nodes is  2230 , #comparisons is  22108
RBT: 865  tot. rotations req., height is  12 , #nodes is  2230 , #comparisons is  22245

Deletion:
BST: height is  26 , #nodes is  2038 , #comparisons is  9046
AVL: 50  tot. rotations req., height is  12 , #nodes is  2038 , #comparisons is  6434
RBT: 47  tot. rotations req., height is  12 , #nodes is  2038 , #comparisons is  6381

Search:
BST:  8381  total comparisons required,  204  numbers found,  378  numbers not found.
AVL:  5777  total comparisons required,  204  numbers found,  378  numbers not found.
RBT:  5787  total comparisons required,  204  numbers found,  378  numbers not found.
```

Figure 15: Test Case 9 Results

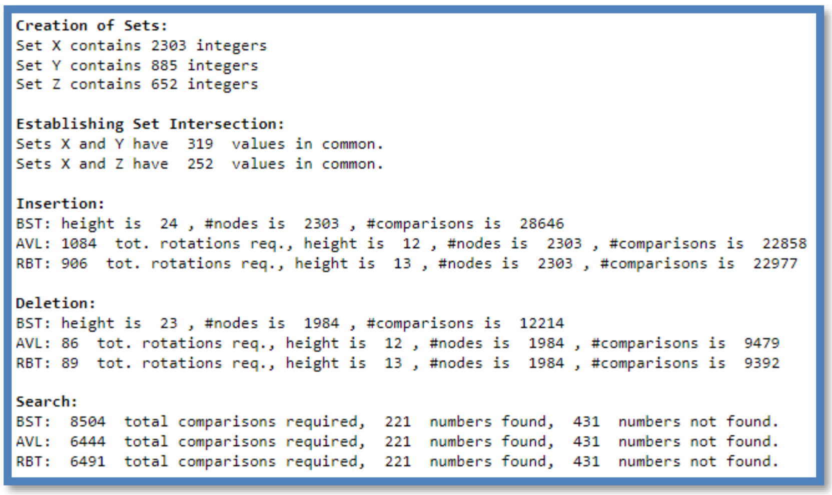
Test Case 10:



```
Creation of Sets:
Set X contains 1048 integers
Set Y contains 626 integers
Set Z contains 722 integers

Establishing Set Intersection:
Sets X and Y have  103  values in common.
Sets X and Z have  116  values in common.

Insertion:
BST: height is  21 , #nodes is  1048 , #comparisons is  11876
AVL: 504  tot. rotations req., height is  11 , #nodes is  1048 , #comparisons is  9215
RBT: 411  tot. rotations req., height is  11 , #nodes is  1048 , #comparisons is  9252

Deletion:
BST: height is  21 , #nodes is  945 , #comparisons is  8246
AVL: 22  tot. rotations req., height is  11 , #nodes is  945 , #comparisons is  6158
RBT: 19  tot. rotations req., height is  11 , #nodes is  945 , #comparisons is  6178

Search:
BST:  8924  total comparisons required,  103  numbers found,  619  numbers not found.
AVL:  6520  total comparisons required,  103  numbers found,  619  numbers not found.
RBT:  6527  total comparisons required,  103  numbers found,  619  numbers not found.
```

Figure 16: Test Case 10 Results

Test Case 11:



```
Creation of Sets:
Set X contains 2303 integers
Set Y contains 885 integers
Set Z contains 652 integers

Establishing Set Intersection:
Sets X and Y have  319  values in common.
Sets X and Z have  252  values in common.

Insertion:
BST: height is  24 , #nodes is  2303 , #comparisons is  28646
AVL: 1084  tot. rotations req., height is  12 , #nodes is  2303 , #comparisons is  22858
RBT: 906  tot. rotations req., height is  13 , #nodes is  2303 , #comparisons is  22977

Deletion:
BST: height is  23 , #nodes is  1984 , #comparisons is  12214
AVL: 86  tot. rotations req., height is  12 , #nodes is  1984 , #comparisons is  9479
RBT: 89  tot. rotations req., height is  13 , #nodes is  1984 , #comparisons is  9392

Search:
BST:  8504  total comparisons required,  221  numbers found,  431  numbers not found.
AVL:  6444  total comparisons required,  221  numbers found,  431  numbers not found.
RBT:  6491  total comparisons required,  221  numbers found,  431  numbers not found.
```

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]

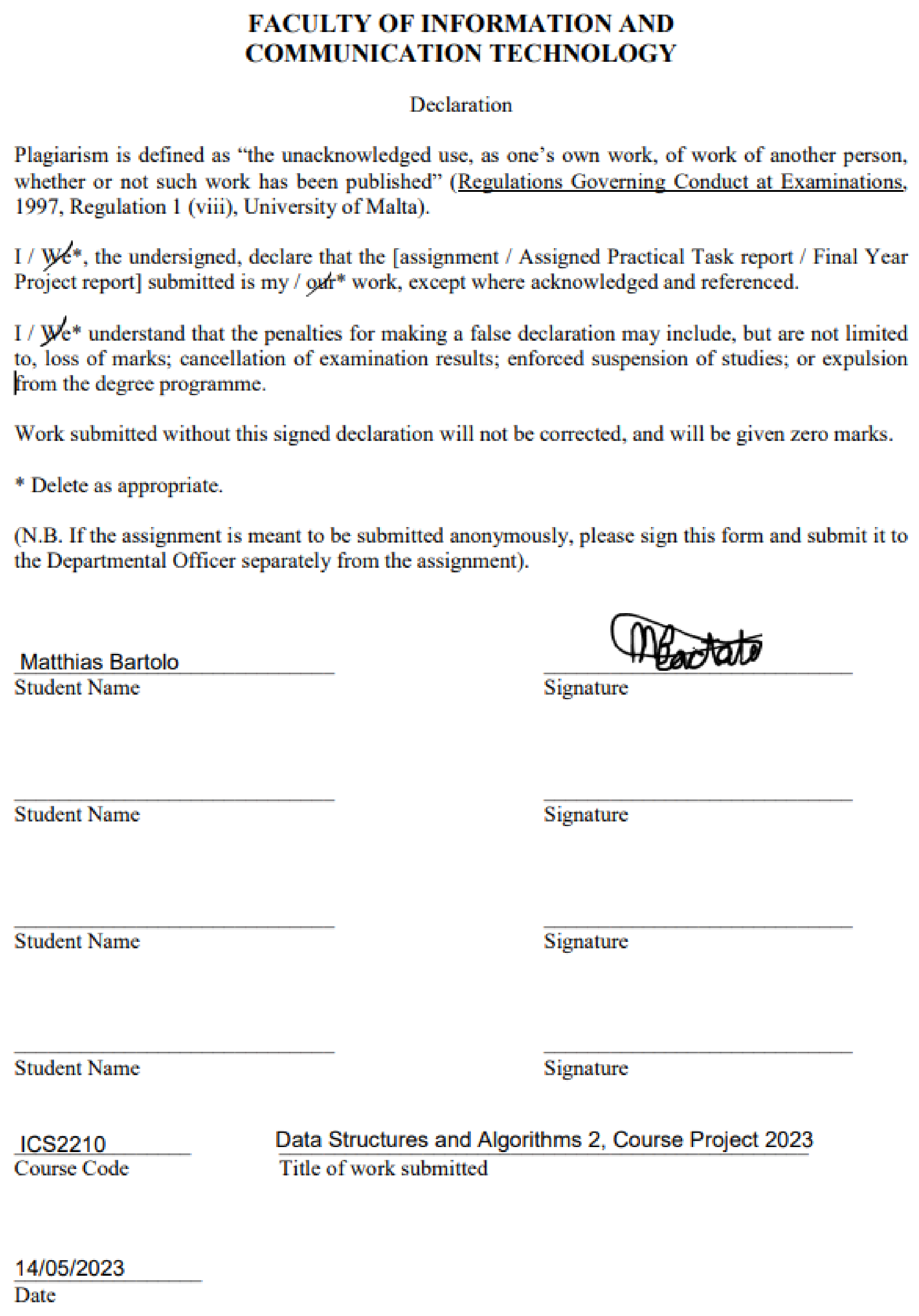
[3] K. Guillaumeir, ICS2210: Red-Black Trees, 2022 [Online]. Available: https://www.um.edu.mt/vle/pluginfile.php/1176939/mod_resource/content/1/Red-Black%20Trees.pdf. [Accessed: 22- Mar- 2022]

[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



FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I / ~~We~~*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / ~~our~~* work, except where acknowledged and referenced.

I / ~~We~~* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Matthias Bartolo	[illegible]
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature
ICS2210	Data Structures and Algorithms 2, Course Project 2023
Course Code	Title of work submitted

14/05/2023
Date