L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Artificial Intelligence

ICS2210 - Data Structures and Algorithms 2

Course Project

Jan Lawrence Formosa
(0435502L)

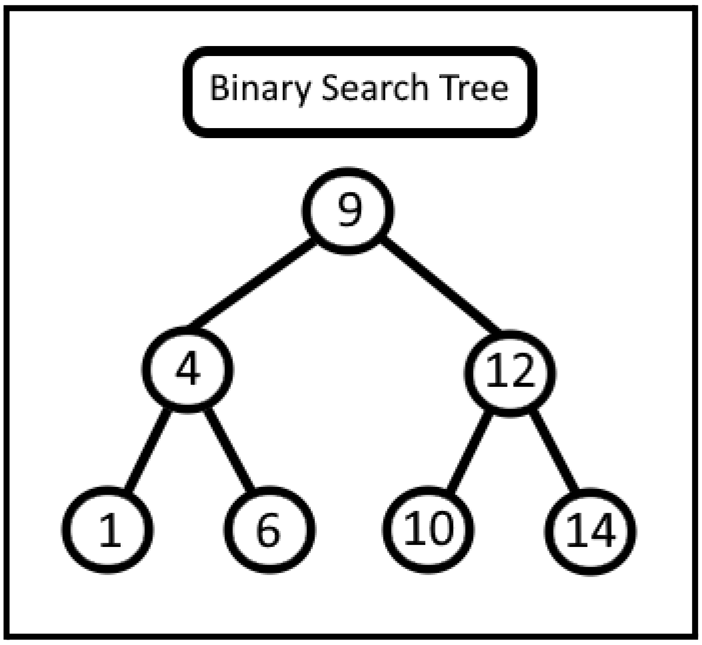
Statement Of Completion

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**
Unbalanced BST Insert	**Yes**
Unbalanced BST Delete	**Yes**
Unbalanced BST Search	**Yes**
Discussion Comparing Tree Data Structures	**Yes**

Introduction

Binary Search Trees[1] are data structures used in numerous indexing and searching algorithms.

They are rooted data structures where the value of each internal node is less than all the values on the node’s right subtree and greater than all the values on the node’s left subtree.



Example of a simple Binary Search Tree

While there are multiple different types of Binary Search Trees, this project will provide an implementation and evaluation on three specific three types.

First, an Unbalanced Binary Search Tree which will act as a comparison point for balanced and unbalanced trees.

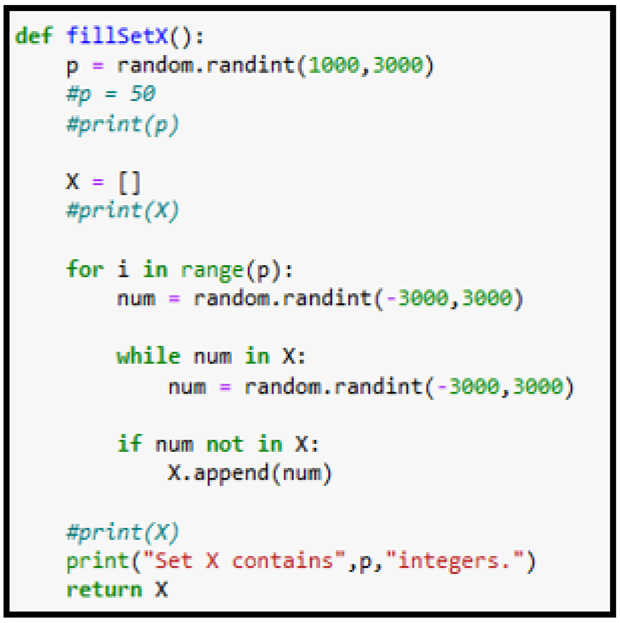
Then two self-balancing trees, namely the AVL Tree[2] and Red Black Tree[3]. These trees will not only be compared with the unbalanced BST but also compared to one another to see which of the self-balancing trees is the most efficient.

Program Structure

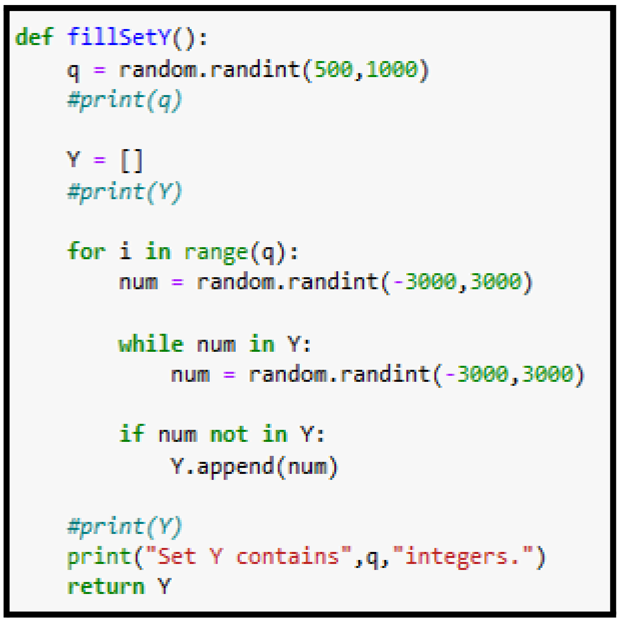
The first step in the program is to create three sets X, Y and Z. These sets will be used for insertion, deletion and searching respectively within the trees.

Each set contains a list of numbers ranging between -3000 and 3000 with no duplicates.

While the amount of numbers in each list is different each time the program is run, the set X will always contain between 1000 and 3000 unique numbers while sets Y and Z will always contain between 500 and 1000 unique numbers.



```python
def fillSetX():
    p = random.randint(1000,3000)
    #p = 50
    #print(p)

    X = []
    #print(X)

    for i in range(p):
        num = random.randint(-3000,3000)

        while num in X:
            num = random.randint(-3000,3000)

        if num not in X:
            X.append(num)

    #print(X)
    print("Set X contains",p,"integers.")
    return X
```

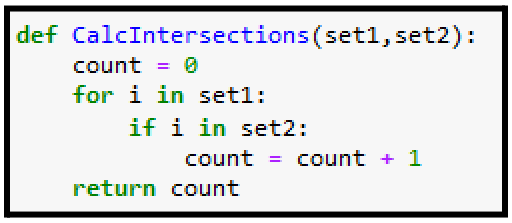


```python
def fillSetY():
    q = random.randint(500,1000)
    #print(q)

    Y = []
    #print(Y)

    for i in range(q):
        num = random.randint(-3000,3000)

        while num in Y:
            num = random.randint(-3000,3000)

        if num not in Y:
            Y.append(num)

    #print(Y)
    print("Set Y contains",q,"integers.")
    return Y
```

Functions to fill sets X and Y

It should be noted that whilst each list has no duplicates, it is expected and required for there to be overlap between sets for the deletion and search to serve a purpose.

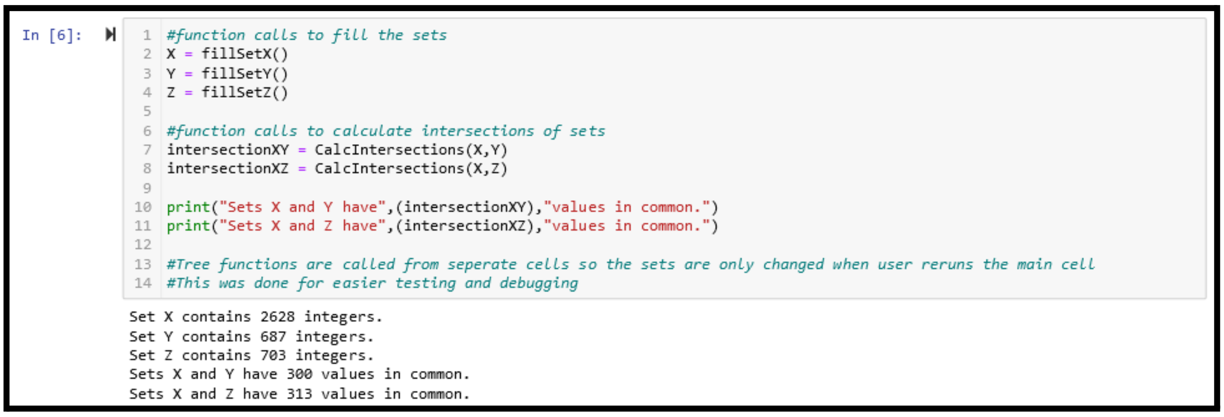
The number of matching numbers across the sets is obtained via the “CalcIntersection” function.



```python
def CalcIntersections(set1,set2):
    count = 0
    for i in set1:
        if i in set2:
            count = count + 1
    return count
```

Function to find intersection of two sets

After, all the functions to fill the sets and get their intersections are called from the same cell, with their respective results being displayed.



```python
In [6]:
#function calls to fill the sets
X = fillSetX()
Y = fillSetY()
Z = fillSetZ()

#function calls to calculate intersections of sets
intersectionXY = CalcIntersections(X,Y)
intersectionXZ = CalcIntersections(X,Z)

print("Sets X and Y have",(intersectionXY),"values in common.")
print("Sets X and Z have",(intersectionXZ),"values in common.")

#Tree functions are called from seperate cells so the sets are only changed when user reruns the main cell
#This was done for easier testing and debugging
```

```
Set X contains 2628 integers.
Set Y contains 687 integers.
Set Z contains 703 integers.
Sets X and Y have 300 values in common.
Sets X and Z have 313 values in common.
```

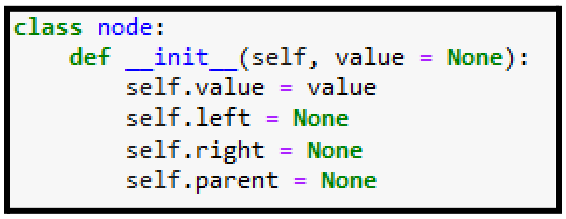
Cell calling respective functions and outputting the results

After these sets are generated, they can be used by the trees to perform insertion, deletion and search operations.

Unbalanced Binary Search Tree

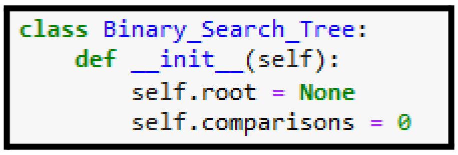
The Unbalanced Binary Search Tree is the easiest BST to implement since, as the name implies, it is not balanced. This means that one side of the tree can be larger than the other and there will be no issue.

Before defining the tree and its functions we also need to define the node, the node containing the value to be stored, any child nodes and its parent node.



```
class node:
    def __init__(self, value = None):
        self.value = value
        self.left = None
        self.right = None
        self.parent = None
```

Inside the “Binary_Search_Tree” class we also define the root and the number of comparisons.

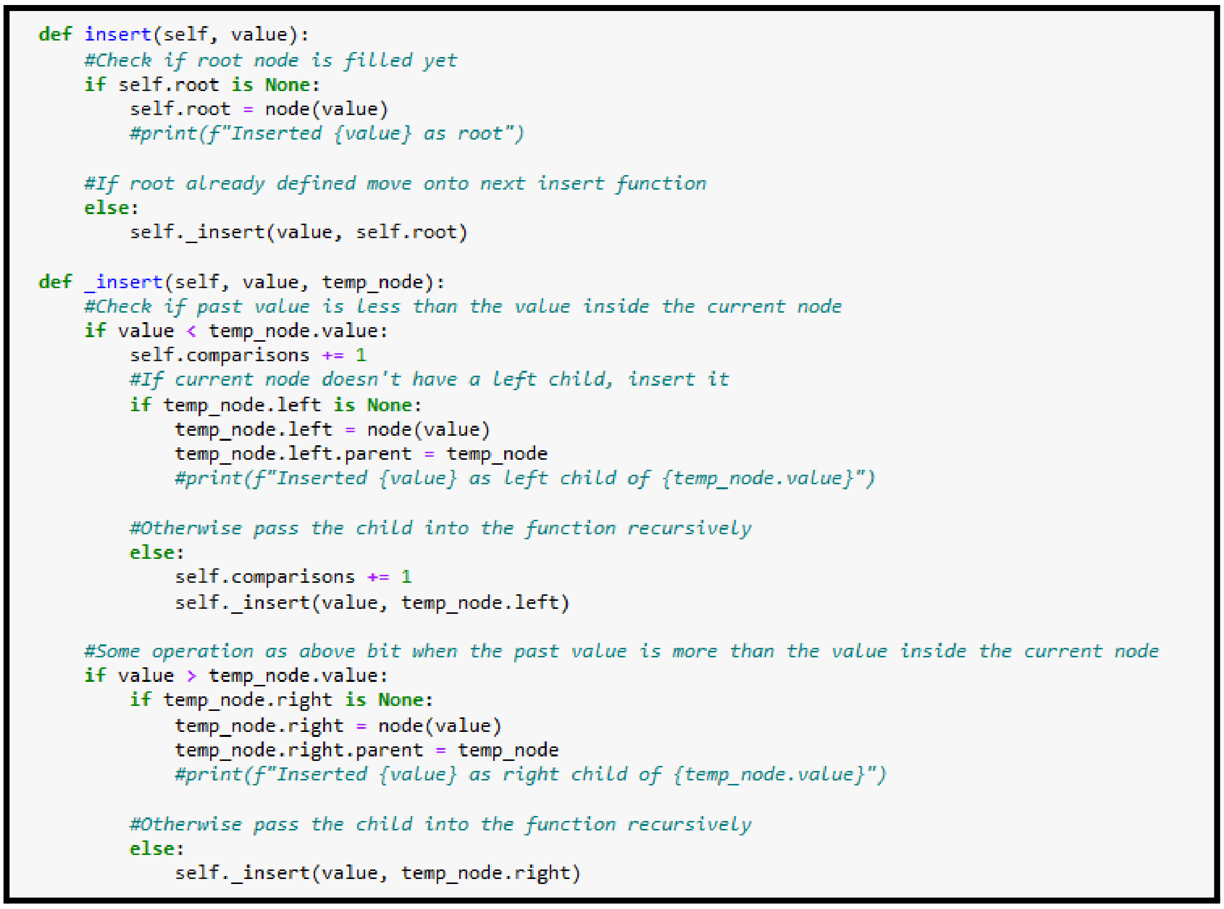


```
class Binary_Search_Tree:
    def __init__(self):
        self.root = None
        self.comparisons = 0
```

To insert in the tree, the program first checks if a root node exists. If no root exists then the current node becomes the root. In the case that the root already exists, the “_insert” function is called.

In the “_insert” function, the program checks if the previous value is less than the value inside the current node. If this is the case then the program also checks whether the current node has a left child or not. If there is no left child then the previous value becomes the child and the current node becomes its parent. When a child already exists it means the end of the tree hasn't been reached so the “_insert” function is called recursively until the end is reached.

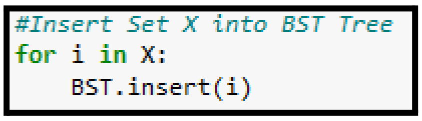
The exact same operation occurs when the previous value is more than the current value except the nodes affected are on the right instead of the left.



```python
def insert(self, value):
    #Check if root node is filled yet
    if self.root is None:
        self.root = node(value)
        #print(f"Inserted {value} as root")

    #If root already defined move onto next insert function
    else:
        self._insert(value, self.root)

def _insert(self, value, temp_node):
    #Check if past value is less than the value inside the current node
    if value < temp_node.value:
        self.comparisons += 1
        #If current node doesn't have a left child, insert it
        if temp_node.left is None:
            temp_node.left = node(value)
            temp_node.left.parent = temp_node
            #print(f"Inserted {value} as left child of {temp_node.value}")

        #Otherwise pass the child into the function recursively
        else:
            self.comparisons += 1
            self._insert(value, temp_node.left)

    #Some operation as above bit when the past value is more than the value inside the current node
    if value > temp_node.value:
        if temp_node.right is None:
            temp_node.right = node(value)
            temp_node.right.parent = temp_node
            #print(f"Inserted {value} as right child of {temp_node.value}")

        #Otherwise pass the child into the function recursively
        else:
            self._insert(value, temp_node.right)
```

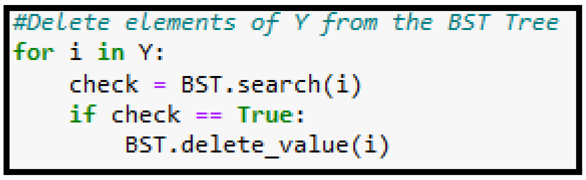
Insert functions described above

The insertion function is called from outside the class as follows until the entire tree is filled.



```python
#Insert Set X into BST Tree
for i in X:
    BST.insert(i)
```

Deleting operates very differently as it starts by calling the “search” function outside the class to make sure the value to be deleted even exists in the tree. It is only when the result is true that the delete is called.



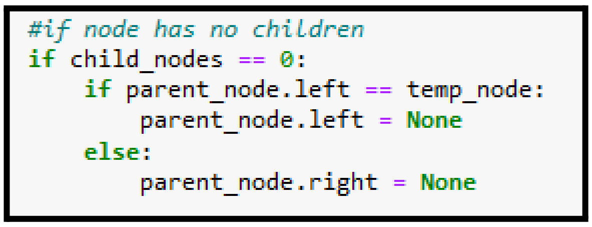
```python
#Delete elements of Y from the BST Tree
for i in Y:
    check = BST.search(i)
    if check == True:
        BST.delete_value(i)
```

The search function traverses the tree recursively, returning true if the value it is searching for is found in the tree.

After the search returns true and the delete is called, another function called “find” is immediately called. While this operates similarly to “search”, instead of finding if a value is in the tree or not, it finds the node containing that value since we’ve already established that the value exists.

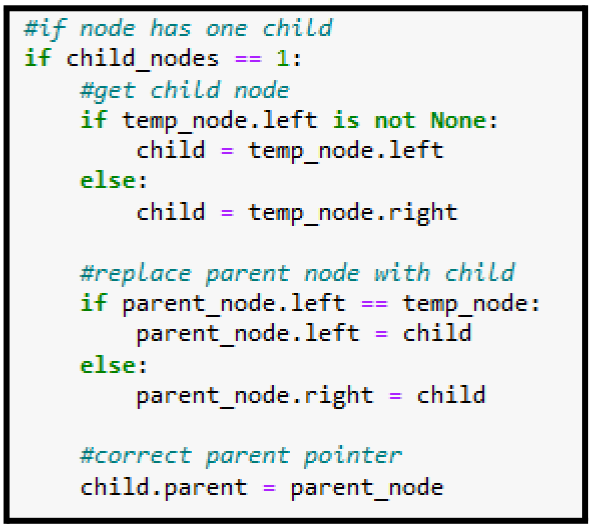
Once reaching the “delete_node” function, three different scenarios can take place depending on the number of children the node to be deleted has.

If it has no children, then the respective side of the parent node is cleared.



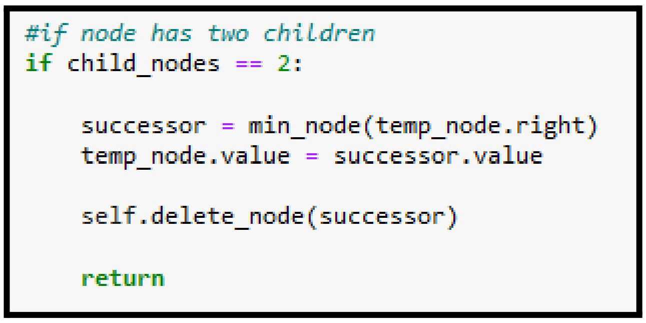
```python
#if node has no children
if child_nodes == 0:
    if parent_node.left == temp_node:
        parent_node.left = None
    else:
        parent_node.right = None
```

If there is a child then the child replaces the parent node and the parent pointer is updated.



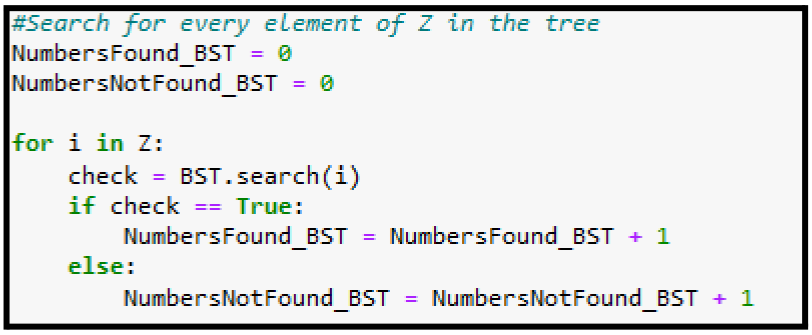
```python
#if node has one child
if child_nodes == 1:
    #get child node
    if temp_node.left is not None:
        child = temp_node.left
    else:
        child = temp_node.right

    #replace parent node with child
    if parent_node.left == temp_node:
        parent_node.left = child
    else:
        parent_node.right = child

    #correct parent pointer
    child.parent = parent_node
```

If there are two children nodes then the code starts by finding the successor of the node. The successor is the smallest value in the right subtree of the node to be deleted. The value of the node is then replaced with the successor's value. Lastly, it recursively calls the “delete_node” function to delete the successor node from its original position.



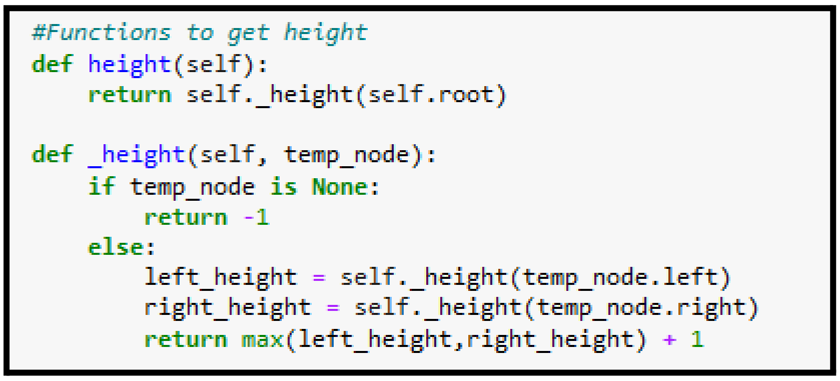
```python
#if node has two children
if child_nodes == 2:

    successor = min_node(temp_node.right)
    temp_node.value = successor.value

    self.delete_node(successor)

    return
```

To search for the values of Z remaining in the tree, the same search function described for deleting is used but when the results are true or false, respective counters are updated accordingly.



```
#Search for every element of Z in the tree
NumbersFound_BST = 0
NumbersNotFound_BST = 0

for i in Z:
    check = BST.search(i)
    if check == True:
        NumbersFound_BST = NumbersFound_BST + 1
    else:
        NumbersNotFound_BST = NumbersNotFound_BST + 1
```

To get the height of the tree, the height functions recursively traverse the left and right sides of the tree. The larger of the two sides is then returned.



```
#Functions to get height
def height(self):
    return self._height(self.root)

def _height(self, temp_node):
    if temp_node is None:
        return -1
    else:
        left_height = self._height(temp_node.left)
        right_height = self._height(temp_node.right)
        return max(left_height,right_height) + 1
```

The node count is done using another set of recursive functions. It gets the number of nodes on the left and the number of nodes on the right side of the tree and adds them together. A plus one is added after to represent the root node.

Every time a comparison is made, the self.comparisons value is incremented. The get_compарsions function can then be called to get this value. It is reset to 0 in between insert, delete and search to get accurate results.

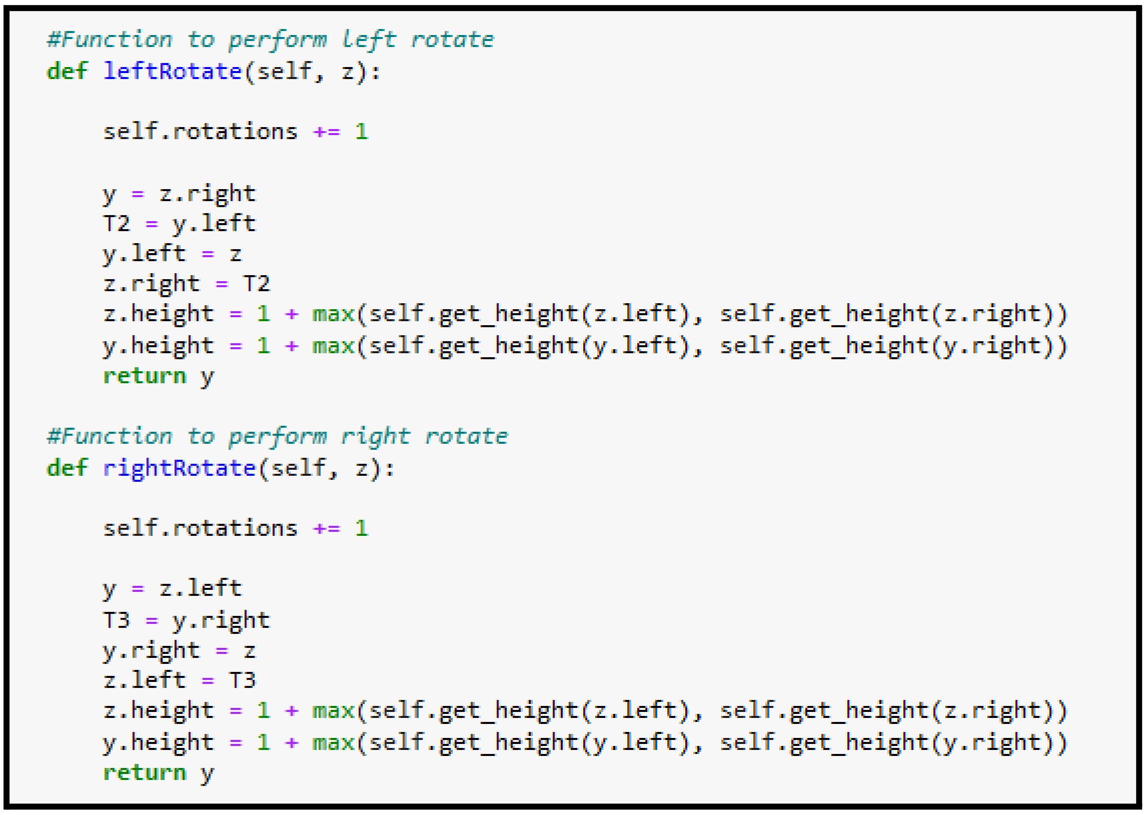
AVL Tree

All the functions in the AVL tree function differently to the Unbalanced Binary Search Tree (UBST) as the root is handled outside the "AVL_Tree" class. A height property is added to the node class whilst the parent is removed .

Despite this the search, height, node count and comparison count functions all work similarity, in most cases just requiring the root to be passed in the function call.

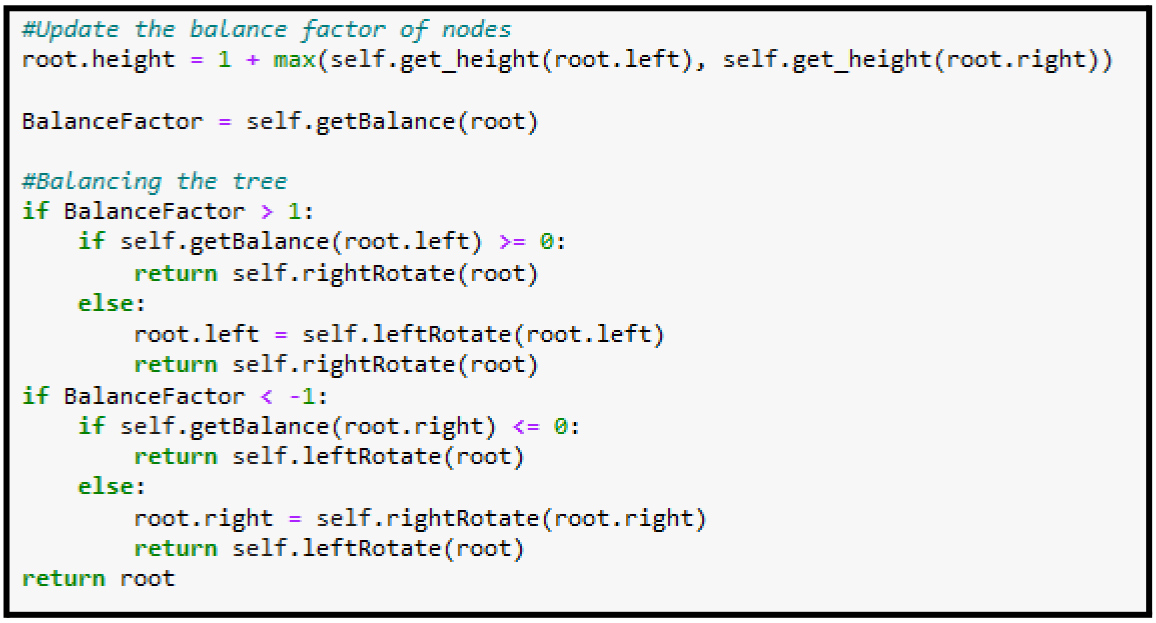
A new count is tracked using "self.rotations" and identically to "self.comparisons" is returned via a simple function call. The only difference being that rotations are incremented for rotations not comparisons.

The biggest change between the AVL and the UBST is the fact that the AVL is self balancing and therefore needs to rotate after inserting and deleting to stay balanced. These rotations are handled by the LeftRotate and RightRotate functions.



```
#Function to perform left rotate
def leftRotate(self, z):

    self.rotations += 1

    y = z.right
    T2 = y.left
    y.left = z
    z.right = T2
    z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
    y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
    return y

#Function to perform right rotate
def rightRotate(self, z):

    self.rotations += 1

    y = z.left
    T3 = y.right
    y.right = z
    z.left = T3
    z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
    y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
    return y
```

At the end of an insert, the BalanceFactor of the tree is calculated and based on the result a left or right rotate on the required nodes is called.

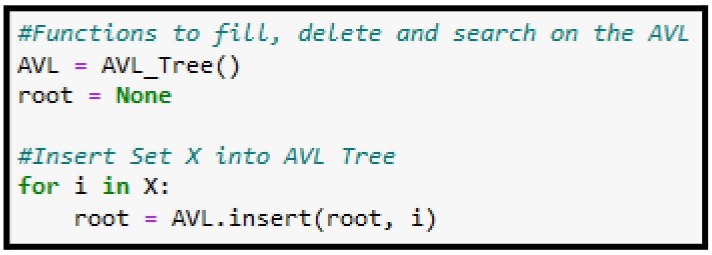


```python
#Update the balance factor of nodes
root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))

BalanceFactor = self.getBalance(root)

#Balancing the tree
if BalanceFactor > 1:
    if self.getBalance(root.left) >= 0:
        return self.rightRotate(root)
    else:
        root.left = self.leftRotate(root.left)
        return self.rightRotate(root)
if BalanceFactor < -1:
    if self.getBalance(root.right) <= 0:
        return self.leftRotate(root)
    else:
        root.right = self.rightRotate(root.right)
        return self.leftRotate(root)
return root
```

The exact same thing occurs when deleting where at the end of the delete the BalanceFactor is calculated and any respective rotates are called.

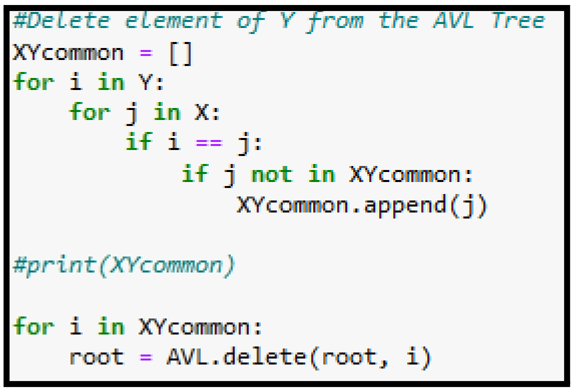
The getBalance function simply uses the height function described for the UBST but instead of adding the sides it subtracts one from the other to get a BalanceFactor.

Like with the UBST, the functions are then called from a separate cell. This time the root is being handled from within this cell rather than the class so the function calls operate a bit differently.



```python
#Functions to fill, delete and search on the AVL
AVL = AVL_Tree()
root = None

#Insert Set X into AVL Tree
for i in X:
    root = AVL.insert(root, i)
```

Because of the root being handled in this cell, the delete had to be called differently where instead of passing values that were returned as true, a list with the intersections is created beforehand and these values are passed immediately.



```python
#Delete element of Y from the AVL Tree
XYcommon = []
for i in Y:
    for j in X:
        if i == j:
            if j not in XYcommon:
                XYcommon.append(j)

#print(XYcommon)

for i in XYcommon:
    root = AVL.delete(root, i)
```

Different method to call delete in the AVL tree

Red Black Tree

In the Red Black Tree the root is once again handled by the tree class so the search, height and node count are all very similar to the UBST methods with slight adjustments to the variable names and return statements.

The comparison and rotation counts work identically to how they do in the AVL class with the positions of the increments slightly adjusted due to a different function structure.

There are a number of changes to the node class with each node now being assigned a colour and the parent pointer being reinstated.

Red Black Trees have to abide by four distinct properties in order to be correctly implemented. These are the following:
- The root must be black
- Every external node (leaf) must be black
- The children of a red node must be black
- All external nodes must have the same black depth

In order to abide by these properties, the insert and delete require additional functions called “insertFix” and “deleteFix” respectively as well as the initialisation of the class also creating NULL nodes.

The insert begins by creating a new node with the given value. At first, the node is coloured red to satisfy the Red-Black Tree property that the root must always be black.

Starting from the root, the tree is then traversed to find the appropriate position to insert the new node. The required comparisons are made with the existing nodes to see if the new node should be placed to the left or right side.

After the position is found, the parent-child connections are established. The parent of the new node is set to the node at the insertion position, and the children of the new node are set to NULL nodes.

If there is a grandparent node, it means the tree was not initially empty. In this case, the “insertFix” method is called to fix any imbalances in the tree.

The “insertFix” method fixes any violations of the Red-Black Tree properties that occurred during insertion. It handles cases such as the violation of the red-black colouring rule or the tree becoming unbalanced.

It does this by performing the following steps:

First the uncle node is determined based on the position of the parent node.

If the parent and uncle nodes are both red then the colour of the parent and uncle nodes are changed to black, while the colour of the grandparent node is changed to red.

If the parent and uncle are not both red then depending on if the new node and parents are right childs are left childs, a few different things can happen.

When the new node is a right child and the parent is a left child, a left rotation is performed on the parent node.

When the new node is a left child and the parent is a left child, or the new node is a right child and the parent is a right child , a right rotation is performed on the grandparent node, whilst the colours of the parent and grandparent nodes are swapped. This ensures that the parent node becomes the new subtree root, and the grandparent node becomes a child of the parent node.

Once the necessary rotations are done, the “insertFix” method is called recursively on the grandparent node to further check for any violations and fix them. This process repeats itself until all the violations are resolved or until the root node is reached. If the root node is reached, its colour is set to black since root nodes must always be black in RBTs.

To delete, a similar deleteFix is used to fix any RBT property violations. Deleting in the RBT is done as follows.

The node to be deleted is located in the tree based on its value. If the node is not found, the function simply returns as there is nothing to delete.

If the node to be deleted has no children it is simply removed from the tree.

If the node to be deleted has a single child, that child replaces it, becoming the new child of the node's parent.

If the node to be deleted has two children, it is replaced with its successor or predecessor node. Like in the UBST, the successor is the smallest node in the right subtree, whilst the predecessor is the largest node in the left subtree. The same delete algorithm is then called on the successor or predecessor to delete them from their original position.

After the deletion if there is any violation, the “deleteFix” method is called to fix them. A list of the potential violations and there fixes are as follows:

If the node to be deleted is red, it is simply removed

If the node to be deleted is black and its sibling is red, a colour change and rotation are performed.

If the node to be deleted is black and its sibling is black whilst both the sibling's children are black, the sibling is recolored to red. If the parent is red, it is changed to black. If the parent is black, the problem is moved up to the parent node recursively.

If the node to be deleted is black and its sibling is black whilst both sibling's left child is red and right child is black, a colour change and rotation are performed.

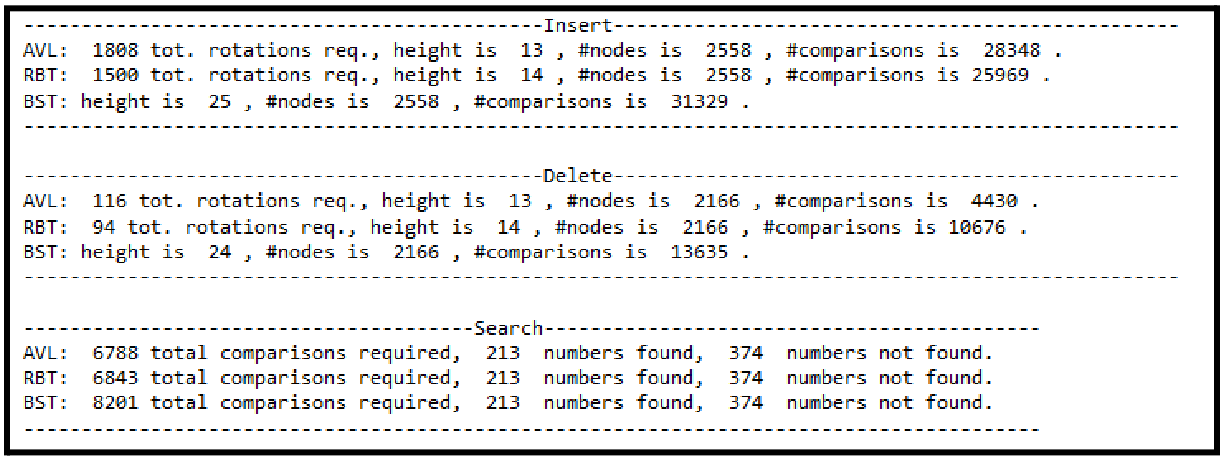
If the node to be deleted is black and its sibling is black whilst the sibling's right child is red, a rotation and colour change are performed.

Another function used by the delete operation is the “__RB_Transplant” which is responsible for transplanting one subtree with another.

Calling the Red Black Tree functions from a separate class is done identically to how it is done with the UBST with the addition of the rotation count method being called.

Evaluation

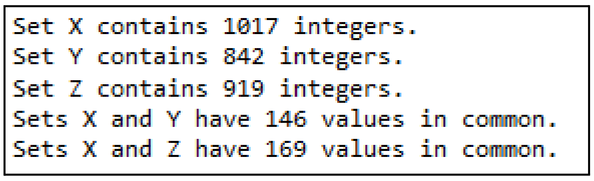
After all the operations are performed on the trees, the respective results are outputted in the following table.



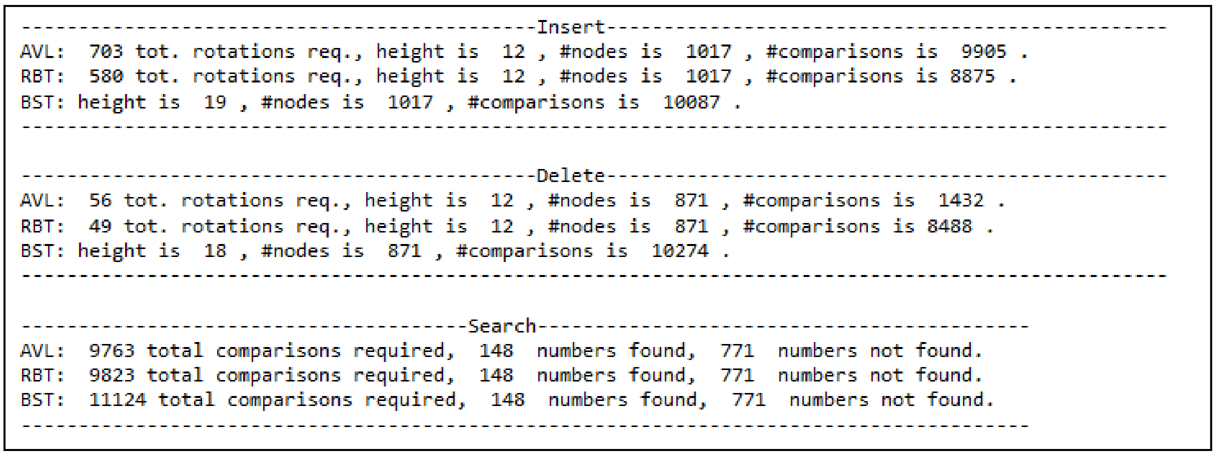
```
---------------------------------------------Insert------------------------------------------------
AVL:  1808 tot. rotations req., height is  13 , #nodes is  2558 , #comparisons is  28348 .
RBT:  1500 tot. rotations req., height is  14 , #nodes is  2558 , #comparisons is 25969 .
BST: height is  25 , #nodes is  2558 , #comparisons is  31329 .
----------------------------------------------------------------------------------------------------

---------------------------------------------Delete------------------------------------------------
AVL:  116 tot. rotations req., height is  13 , #nodes is  2166 , #comparisons is  4430 .
RBT:  94 tot. rotations req., height is  14 , #nodes is  2166 , #comparisons is 10676 .
BST: height is  24 , #nodes is  2166 , #comparisons is  13635 .
----------------------------------------------------------------------------------------------------

---------------------------------------Search------------------------------------------
AVL:  6788 total comparisons required,  213  numbers found,  374  numbers not found.
RBT:  6843 total comparisons required,  213  numbers found,  374  numbers not found.
BST:  8201 total comparisons required,  213  numbers found,  374  numbers not found.
----------------------------------------------------------------------------------------
```

To ensure that everything is working fine with any sets, three different runs of the program will be evaluated below followed by a comparison between the AVL and RBT trees.

Test Case #1



```
Set X contains 1017 integers.
Set Y contains 842 integers.
Set Z contains 919 integers.
Sets X and Y have 146 values in common.
Sets X and Z have 169 values in common.
```



```
--------------------------------------------Insert-------------------------------------------------
AVL:  703 tot. rotations req., height is  12 , #nodes is  1017 , #comparisons is  9905 .
RBT:  580 tot. rotations req., height is  12 , #nodes is  1017 , #comparisons is 8875 .
BST: height is  19 , #nodes is  1017 , #comparisons is  10087 .
---------------------------------------------------------------------------------------------------

--------------------------------------------Delete-------------------------------------------------
AVL:  56 tot. rotations req., height is  12 , #nodes is  871 , #comparisons is  1432 .
RBT:  49 tot. rotations req., height is  12 , #nodes is  871 , #comparisons is 8488 .
BST: height is  18 , #nodes is  871 , #comparisons is  10274 .
---------------------------------------------------------------------------------------------------

--------------------------------------Search-----------------------------------------
AVL:  9763 total comparisons required,  148  numbers found,  771  numbers not found.
RBT:  9823 total comparisons required,  148  numbers found,  771  numbers not found.
BST:  11124 total comparisons required,  148  numbers found,  771  numbers not found.
-------------------------------------------------------------------------------------
```

From the results above we can immediately deduce that the insert and delete functions are working properly as the number of nodes is consistent throughout all the trees.

This can be seen using the sets as well as set X contains 1017 unique values hence an initial node count with the same number. Once the intersecting numbers with Y are removed (1017 - 146) the result is 871 which matches the result above.

The search results also work with the same amount of numbers being found or not found in every tree. The result of numbers found is different from the intersection of X and Z as all the common elements between X and Y have been deleted by this point and there was overlap between the three lists.

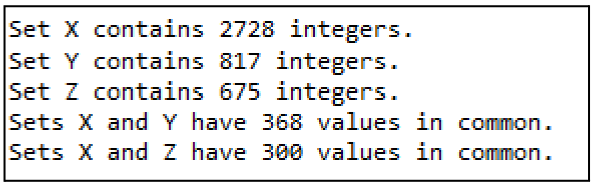
As expected, the height of the unbalanced tree is much larger than that of the balanced trees. The same applies to the number of comparisons where the unbalanced tree consistently has the most.

The height of the AVL and RBT trees are expected to be close to each other and in this case they were the same.

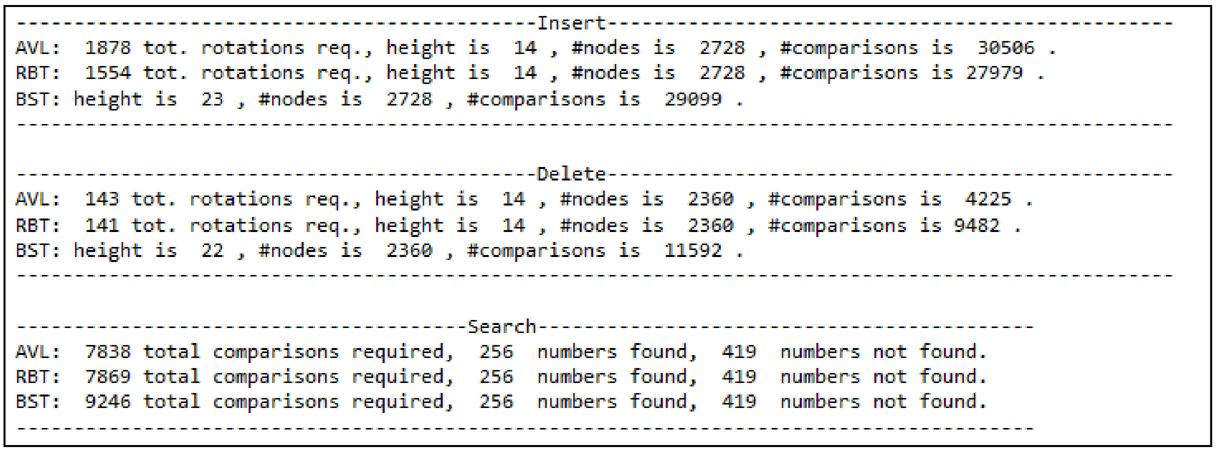
It is interesting to see that AVL had more comparisons than the RBT during insertion but much less during deletion and slightly less during the search.

The RBT consistently required less rotations than the AVL throughout all the operations.

Test Case #2



```
Set X contains 2728 integers.
Set Y contains 817 integers.
Set Z contains 675 integers.
Sets X and Y have 368 values in common.
Sets X and Z have 300 values in common.
```



```
--------------------------------------------Insert-------------------------------------------------
AVL:  1878 tot. rotations req., height is  14 , #nodes is  2728 , #comparisons is  30506 .
RBT:  1554 tot. rotations req., height is  14 , #nodes is  2728 , #comparisons is 27979 .
BST: height is  23 , #nodes is  2728 , #comparisons is  29099 .
---------------------------------------------------------------------------------------------------

--------------------------------------------Delete-------------------------------------------------
AVL:  143 tot. rotations req., height is  14 , #nodes is  2360 , #comparisons is  4225 .
RBT:  141 tot. rotations req., height is  14 , #nodes is  2360 , #comparisons is 9482 .
BST: height is  22 , #nodes is  2360 , #comparisons is  11592 .
---------------------------------------------------------------------------------------------------

--------------------------------------Search-------------------------------------------
AVL:  7838 total comparisons required,  256  numbers found,  419  numbers not found.
RBT:  7869 total comparisons required,  256  numbers found,  419  numbers not found.
BST:  9246 total comparisons required,  256  numbers found,  419  numbers not found.
---------------------------------------------------------------------------------------
```

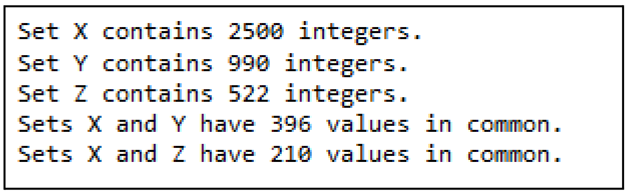
Once again the results above allow us to deduce that the insert and delete functions are working properly as the number of nodes is consistent throughout all the trees.

This can once again be seen using the sets as well as set X contains 2728 unique values hence the initial node count having the same number again and once the intersecting numbers with Y are removed (2728 - 368) the result is 2360 which matches the result above.

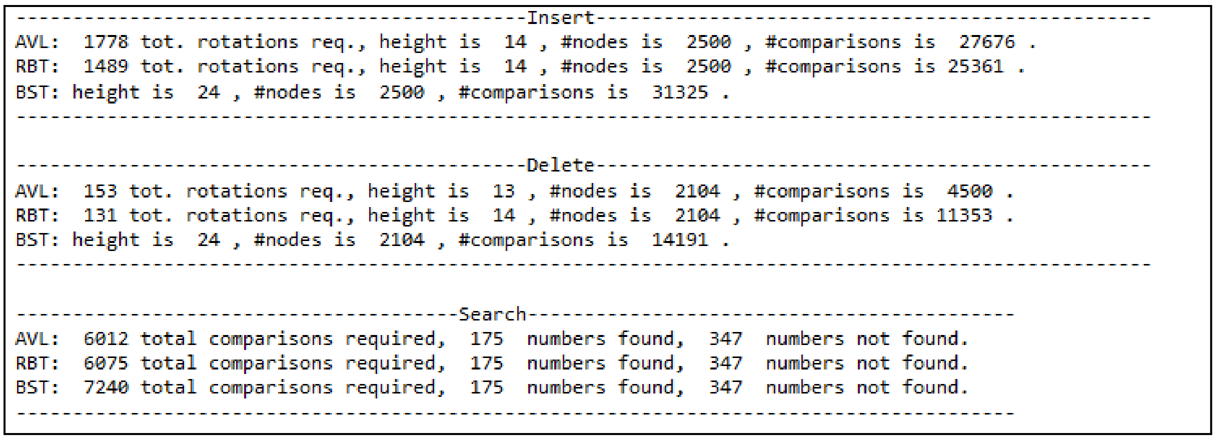
The search results also work with the same amount of numbers being found or not found in every tree. Once again the number from the intersection of X and Z differs from our final result due to the deletion operation.

Interestingly the number of comparisons when inserting where this time largest on the AVL tree however everything else remained consistent with the first test case.

Test Case #3



```
Set X contains 2500 integers.
Set Y contains 990 integers.
Set Z contains 522 integers.
Sets X and Y have 396 values in common.
Sets X and Z have 210 values in common.
```



```
---------------------------------------------Insert---------------------------------------------
AVL:  1778 tot. rotations req., height is  14 , #nodes is  2500 , #comparisons is  27676 .
RBT:  1489 tot. rotations req., height is  14 , #nodes is  2500 , #comparisons is 25361 .
BST: height is  24 , #nodes is  2500 , #comparisons is  31325 .
------------------------------------------------------------------------------------------------

---------------------------------------------Delete---------------------------------------------
AVL:  153 tot. rotations req., height is  13 , #nodes is  2104 , #comparisons is  4500 .
RBT:  131 tot. rotations req., height is  14 , #nodes is  2104 , #comparisons is 11353 .
BST: height is  24 , #nodes is  2104 , #comparisons is  14191 .
------------------------------------------------------------------------------------------------

---------------------------------------Search-------------------------------------------
AVL:  6012 total comparisons required,  175  numbers found,  347  numbers not found.
RBT:  6075 total comparisons required,  175  numbers found,  347  numbers not found.
BST:  7240 total comparisons required,  175  numbers found,  347  numbers not found.
----------------------------------------------------------------------------------------
```

Like the previous cases the results above allow us to deduce that the insert and delete functions are working properly as the number of nodes is consistent throughout all the trees.

Once again this can be shown for the delete as 2500 - 396 is equal to 2104 which is the result obtained above.

As expected the number from the intersection of X and Z once again differs from our final result due to the deletion operation.

The number of comparisons and rotations this time were consistent with the first case.

Comparing the trees

From the results above the Unbalanced Binary Search Tree was far and wide the least efficient tree (apart from rare instance in insertion of Case #2). It seems that the sole benefit of the UBST is its much simpler implementation than a Balanced Tree. The only instance where an unbalanced tree such as this might be used is where there is no time constraint or if you are working with constant values.

It is harder to differentiate between when to use an AVL Tree and an RBT Tree as they function so similarly. The main differences between the trees would be in the following areas:

- Insertion and Deletion: Is much more complex in the AVL tree as it requires more rotations to balance the tree than the RBT.

- Balance Factor: RBT does not make use of a balance factor whistl the AVL uses a balance factor to balance the tree.

- Strictly Balanced: AVL trees are strictly balanced as the height of the subtrees cannot differ by more than one. This is not the case for RBTs.

- Searching: Due to being strictly balanced, searching is much more efficient in AVL Trees than RBTs.

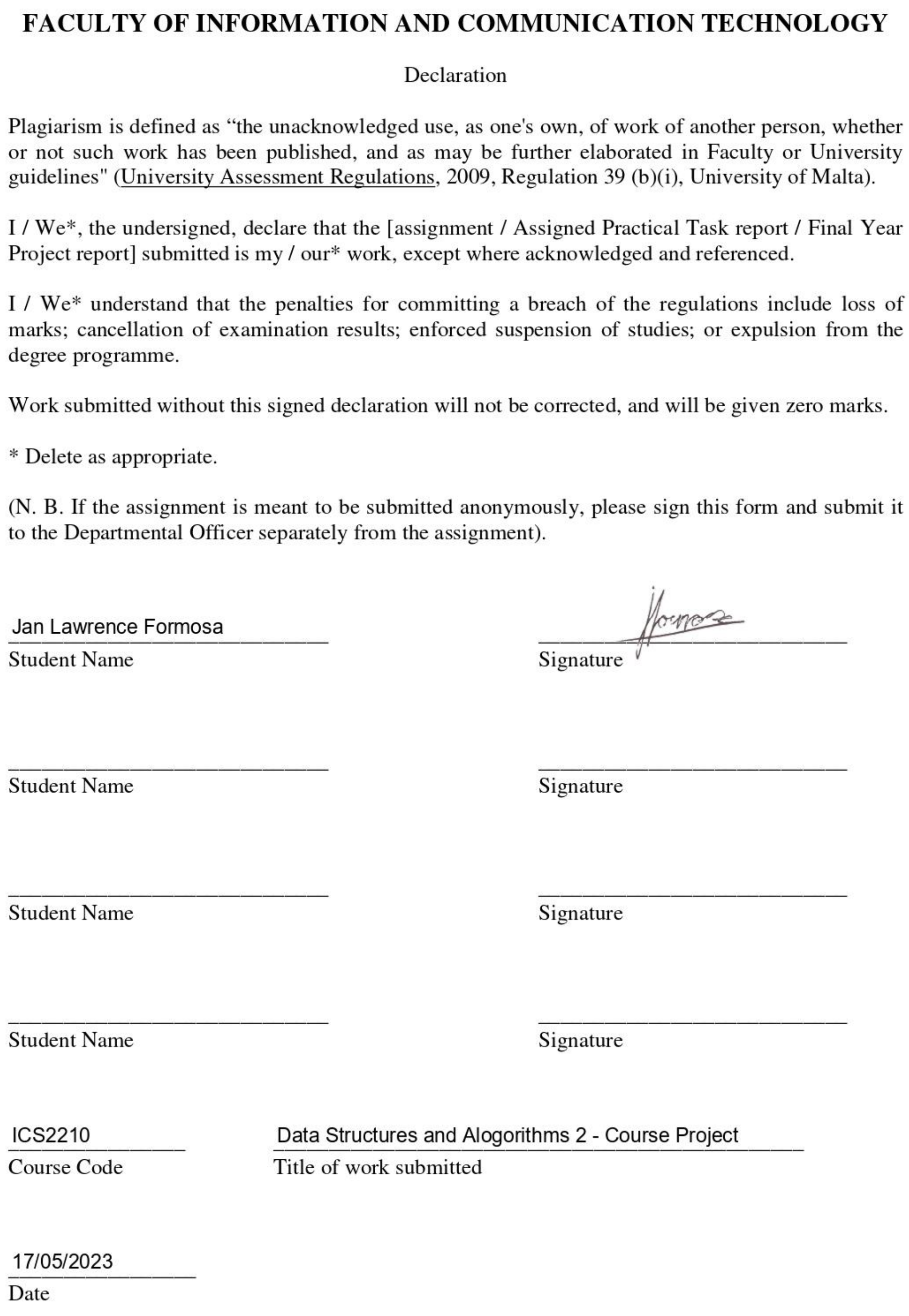
- Node Colour: In RBTs each node has a colour of either red or black whilst AVL Trees do not assign a colour to the node.

Thanks to being more efficient at searching, AVL Trees are commonly used database applications and other applications that require better searching[4].

As they are more efficient at insert and deletion, Red Black Trees are used in applications that require these operations more frequently. RBTs are also used in the Linux-Kernel[5].

Seeing the differences between AVL Trees and RBTs it is hard to totally recommend one over the other as it largely comes down to the operations the program will be performing.

Plagiarism Declaration Form



FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), 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 committing a breach of the regulations include 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).

Jan Lawrence Formosa	[illegible]
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

ICS2210	Data Structures and Alogorithms 2 - Course Project
Course Code	Title of work submitted

17/05/2023
Date

References

- [1] Binary Search Tree, https://en.wikipedia.org/wiki/Binary_search_tree.

- [2] AVL Tree, https://en.wikipedia.org/wiki/AVL_tree.

- [3] Red-black Tree, https://en.wikipedia.org/wiki/Red%E2%80%93black_tree.

- [4] AVL Tree Applications, https://www.javatpoint.com/avl-tree-applications.

- [5] Applications of Red-Black Trees, https://www.baeldung.com/cs/red-black-trees-applications.