CSC2001F Assignment 2
AVL Tree
TWLCOM001 Comfort Twala

R E P O R T

OO Design:

- fileHandler class is the helper class that opens the dataset file with names, creates Student objects and populates the AVL tree using dataTree() method. It also has a dataList() method to populate ArrayList with Student objects to be when running the experiment. More on that later.
- Student class is the data class. It stores the student's ID, name and surname. It is used within the AVL tree as the data type and is accessed using the AccessAVLApp.
- AVLApp class uses the AVLTree of Student objects received from the fileHandler class and uses printAllStudents() to print out all Student objects and printStudent(studentID) to print out a specific Student objects depending on the studentID given.
- AccessAVLApp class is the program driver class that is used to invoke AVLApp methods and test the application.
- BinaryTree class is the data structure class that creates a BinaryTree of Objects and defines the methods to create the Tree.
- AVLTree class is an extension of the BinaryTree class as it is a special type of Binary Tree, a Balanced Binary Tree. It uses all the methods of BinaryTree with its own extra ones to ensure it's balanced. This is used to create the AVLTree in fileHandler and is used throughout.
- BinaryTreeNode class is used as the Node in BinaryTree to store the specific object data and pointers to the left and right Nodes.
- BTQueue class is a helper class in BinaryTree to help with order of the nodes in the BinaryTree.
- BTQueueNode class is used by BTQueue to be able to differentiate the different nodes to be able to go to the next node.
- Experiment class is used to insert random Student objects into AVLTree subset of length n and find them. It records the insert- and find counters and writes it in a simple format onto a text file ready for analysis.
- runExperiment class is the driver program for the Experiment class running it with n as 500 till 5000 with a step of 500.

Goal of the Experiment:

The goal of the experiment was to test the performance of the AVL Tree data structure when inserting and searching for objects in the tree. Performance was measured by how many comparison operations the methods used and whether it corresponded to the theoretical performance and time complexity of the the AVL Tree data structure.

Execution:

I executed the experiment by creating 10 subsets of AVL Trees starting from 500 Student Objects to 5000 Objects, incrementing with 500 Objects every time (n=500,1000,...,5000). For each subset, each Student Object was inserted into the subset, the insert counter for that object was recorded *(saved in data/experiment/insert)* and after the subset was full, each object was searched for in the Tree and the find counter was recorded *(saved in data/experiment/find).* From these results a minimum, maximum and average value were determined for each subset and captured *(saved in data/experiment/analysis).* From the captured data a data table and graph were created *(saved in data/experiment/results)* to show and visualise the performance of the Tree's insert and find methods. Data and graph were conducted using python scripts for accuracy and automation.

Instrumentation:
The insert- and find counters were added into the insert- and find methods of the BinaryTree class, respectively so, whenever a comparison operation was performed by the method. The counters were reset each time so that each Student Object inserted or searched for would return the counter for that specific object and not the whole collective.

Part 1 - Test results:

AccessAVLApp:
- known: *(saved in data/part1/known)*
    1. /usr/bin/java -cp bin AccessAVLApp BKSAVA009
       Ava Beukes
    2. /usr/bin/java -cp bin AccessAVLApp MLTLUK019
       Luke Malatji
    3. /usr/bin/java -cp bin AccessAVLApp TSTSIP016
       Siphesihle Tsotetsi
- unknown: *(saved in data/part1/unknown)*
    1. /usr/bin/java -cp bin AccessAVLApp TWLCOM001
       Access Denied!
    2. /usr/bin/java -cp bin AccessAVLApp MILWRD032
       Access Denied!
    3. /usr/bin/java -cp bin AccessAVLApp TRFKON002
       Access Denied!
-none: *(saved in data/part1)*
    /usr/bin/java -cp bin AccessAVLApp
    MLLNOA014 Noah Maluleke
    KHZOMA010 Omaatla Khoza
    DMSMEL001 Melokuhle Adams
    BXXBON021 Bonolo Booi
    BRHKAT012 Katleho Abrahams
    .....
        WTBTHA010 Thato Witbooi
        WTBSIY016 Siyabonga Witbooi
        WTBTSH028 Tshegofatso Witbooi
        WTBTSH025 Tshegofatso Witbooi
        WTBWAR001 Warona Witbooi

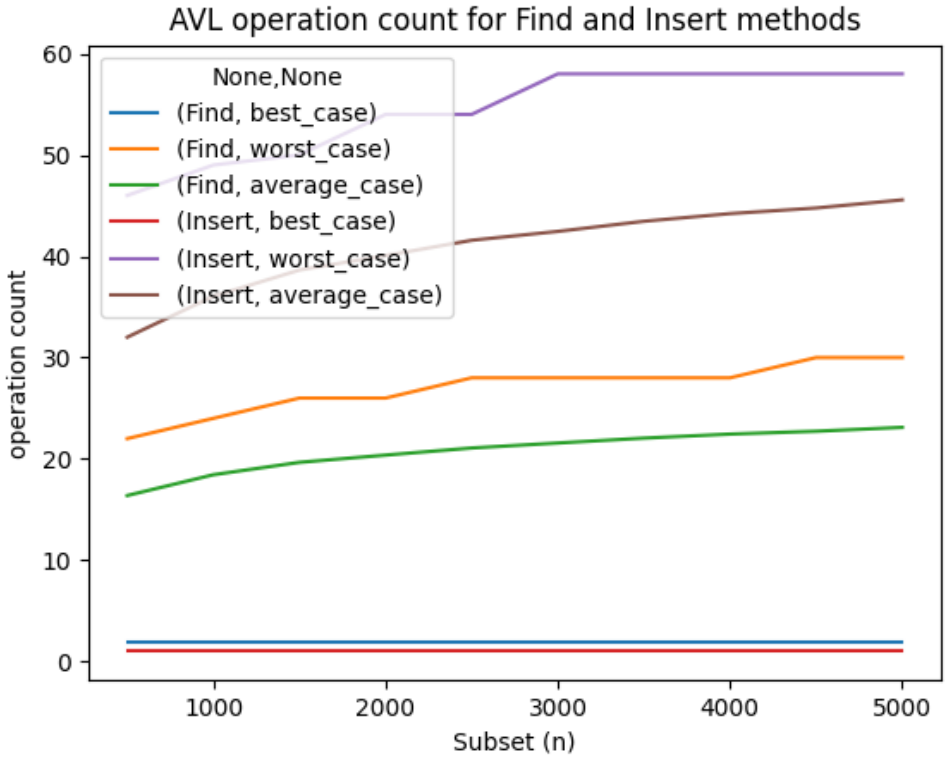
Results – Tables and Graph:

Table: Insert and Find (min, max, average)

	Find	Find	Find	Insert	Insert	Insert
	best_case	worst_case	average_case	best_case	worst_case	average_case
500	2.0	22.0	16.38	1.0	46.0	32.008
1000	2.0	24.0	18.434	1.0	49.0	36.048
1500	2.0	26.0	19.657333333333334	1.0	50.0	38.63733333333333
2000	2.0	26.0	20.372	1.0	54.0	40.068
2500	2.0	28.0	21.0728	1.0	54.0	41.5752
3000	2.0	28.0	21.565333333333335	1.0	58.0	42.451
3500	2.0	28.0	22.046857142857142	1.0	58.0	43.451142857142855
4000	2.0	28.0	22.446	1.0	58.0	44.207
4500	2.0	30.0	22.72711111111111	1.0	58.0	44.76111111111111
5000	2.0	30.0	23.1084	1.0	58.0	45.5628

*** *Generated using combinedStats.py*

Graph: Insert and Find (min, max, average)



Discussion of Results:

Theoretically the time complexity for a AVL tree for insert and search/find is:

$O(\log(n))$ meaning $h = \log_2 n$ and $n = 2^h$.

Therefore, the number of nodes is equal to 2 to the power of the height of the tree. This means that as the number of nodes increase exponentially, the height will slightly in increase. eg. when n is 8, h is 3 and when n = 16, h is only 4. n has doubled while h only increased by 1. Which is why the AVL is the better choice when compared to a BST with time complexity $O(n)$.

When comparing these theoretical values to the experiment results, it mirrors the same concept. The greater the subset, the more operations are performed as the height is more.

For the best case in insert when h = 0 and you're inserting the first node, the value is 1. $2^0 = 1$

For the worst case in insert and n = 5000, the insert only performed 58 comparisons which is very low considering there are 5000 nodes in the tree.

As expected, on average, for both find and insert the average value gradually increases as the number of nodes increase.

The find best case is constant at 2 and the worse case is 30 which is 0.6% of the Tree size.

Insert has higher comparisons performed because it has to balance the Tree every time for it to remain a AVL tree.

Creativity:

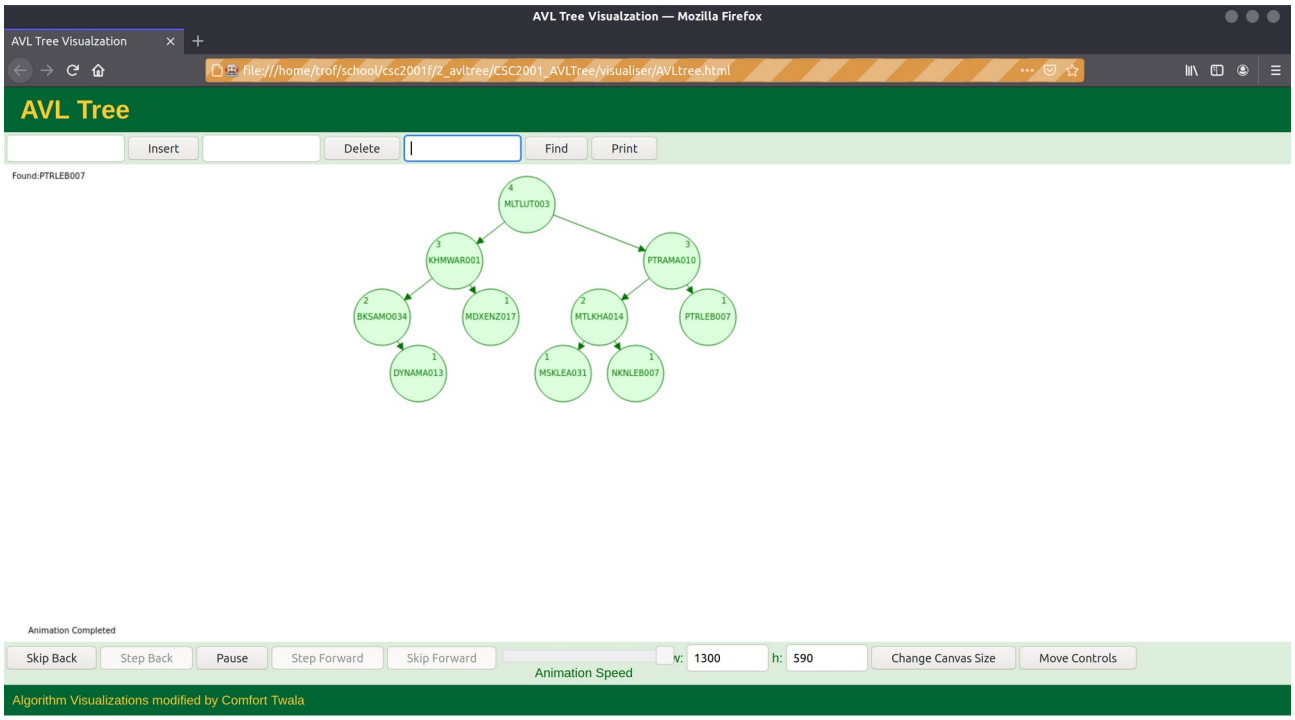
- Modified an online AVL visualiser to visualise how the AVLTree works when inserting values and when searching for them.
- Made the visualiser installation process user friendly and easy to install and use. *(READ README.md FOR USAGE)*
- Created multiple graphs and data tables for data using python scripts.
- Made README.md user friendly and very detailed.
- Everything can be run using Makefile only.

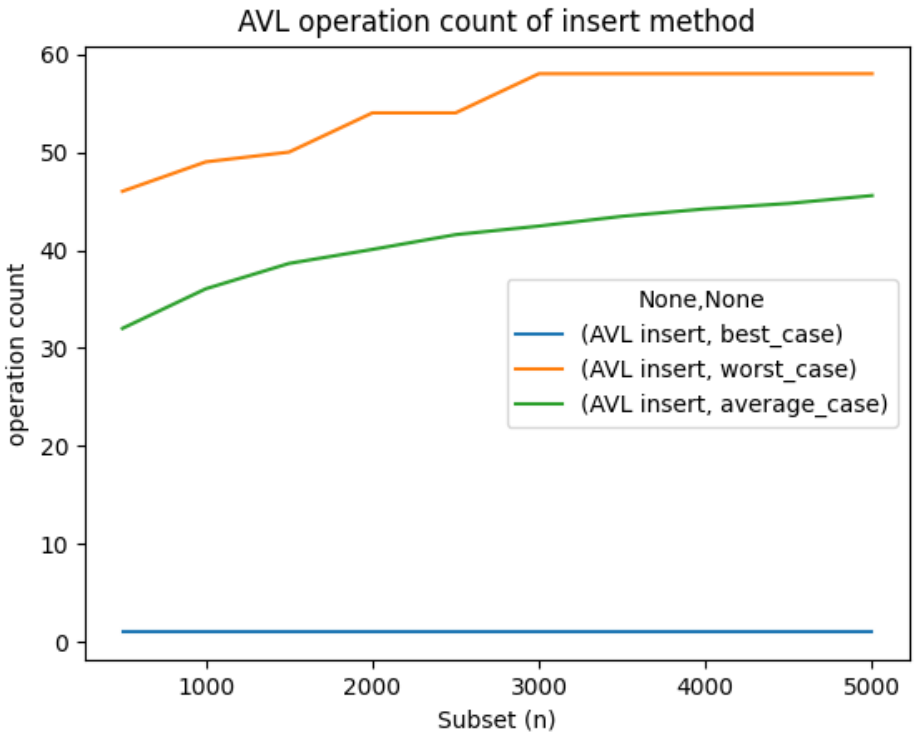
Git usage:

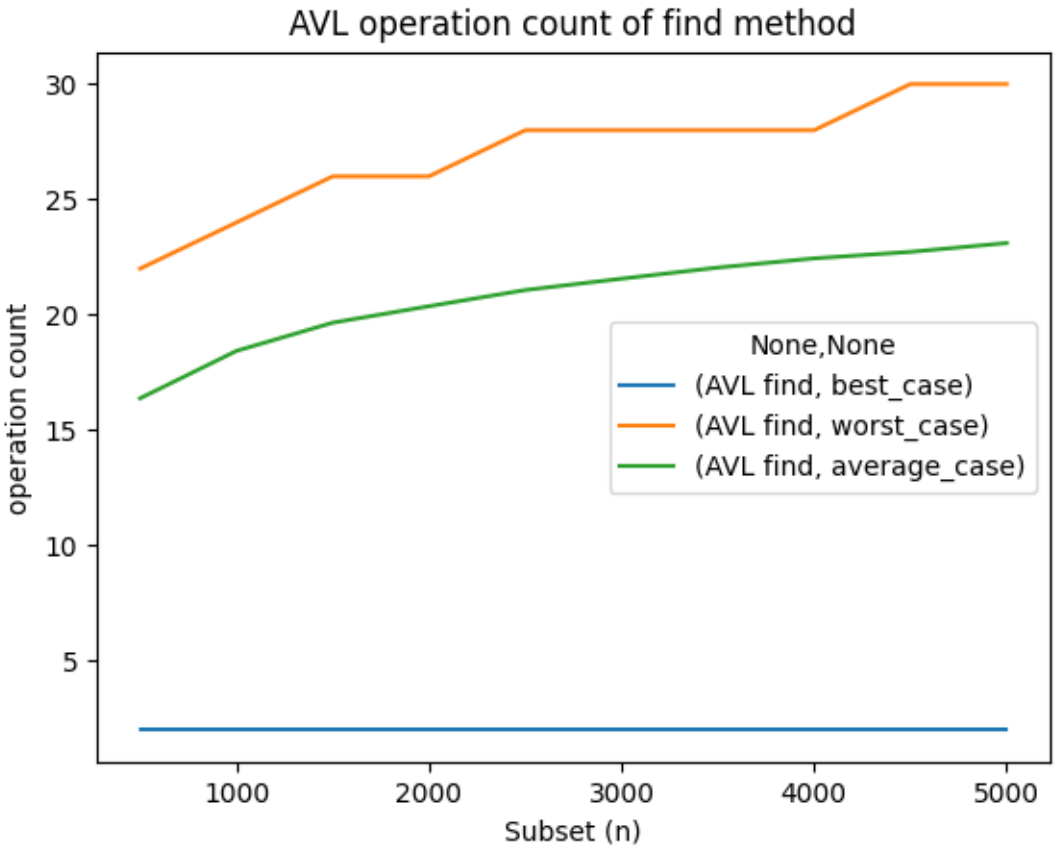
```
0: commit a2c09581c904f2b1fb441a196b53afc3b39bc515
1: Author: Comfort-Twala <kontreitroos@gmail.com>
2: Date: Mon Apr 26 13:06:09 2021 +0200
3:
4: finalised!
5:
6: commit d3db0527a8fb57923b0c995725251c10f0536dd2
7: Author: Comfort-Twala <kontreitroos@gmail.com>
8: Date: Sun Apr 25 23:48:01 2021 +0200
9:
10: Code refractoring and Makefile modifications
...
139: Author: Comfort-Twala <kontreitroos@gmail.com>
140: Date: Fri Apr 16 12:04:29 2021 +0200
141:
142: BinarySearchTree and AVLTree added
143:
144: commit f621972842d76dc4bc4ea3e5fbc1cb5831587b39
145: Author: Comfort-Twala <kontreitroos@gmail.com>
146: Date: Fri Apr 16 11:55:54 2021 +0200
147:
148: Project skeleton code
```

ADDITIONAL DATA GRAPHS:

Visualiser:
READ README.md for usage.






AVL operation count of find method
operation count
Subset (n)
30
25
20
15
10
5
1000
2000
3000
4000
5000
None,None
(AVL find, best_case)
(AVL find, worst_case)
(AVL find, average_case)