CSC2001F
Assignment 2
AVL Trees
Done by: KHLPAL002

Table of Contents

1. Introduction

In this experiment the AVL tree will be investigated, the AVL tree is a Binary Search tree that always has to be balanced on every right and left subtree, the difference in height can only be less than or equal to 1, this is achieved through constant rotation of the tree. This experiment aims to investigate the different time complexities of the insert() function and the find() function.

2. Object Orientation Design and how they interact

I created 3 subclasses namely

1. BinaryTreeNode
2. BinaryTree
3. AVLTee

BinaryTreeNode is used to get the right and left node to be compared to the next key being entered, because this is an AVL tree, Binary Tree prints out the data once it has been sorted and AVL Tree which sorted out the data, balances and does the comparisons and operations to ensure the AVL properties are correct.

All of these were a part of AccessAVLApp class which I used to encapsulate all the methods and classes, so Encapsulation was used.

Furthermore In this program I used inheritance where the similar methods used for printallstudents() method are used for printstudent(ID) method. A super class BinaryTree is extended by AVLTree class and used by printallstudents() and printstudent(ID) methods.

Lastly abstraction was used in find() to hide the Name and Surname from each student as we were only searching for the student ID.

3. Goal of the Experiment and how it was executed

This experiment aims to investigate the different time complexities of the insert() function and the find() function. The time complexity is calculated by inserting a number of n data samples and counting the amount of operations in each case, this will then give an approximation of the time complexity.

The goal was executed by counting the number of operation done for find() and insert() respectively, this was done by initializing a counter in the AVLTree class and counting every time an operation was executed and finally.

Firstly I took names, surnames and IDs from the 5000 that were given in the oklist.txt, I took the first 100 and stored them in the data directory and named it 100names.txt and this would be my first list, I did the same with 200, 300, 400, 500, 600, 700, 800, 900, 1000 and had 10 lists to always refer to when conducting my experiment.

I then read from each list and chose to print out the counter every time an operation was preformed, I stored this information using unix output redirection in the corresponding resultsInsert and resultsSearch directories.

For Insert(), I ran the program by using java AccessAVLApp.java, as this would allow the entire tree to be inserted without adding additional steps to find the an ID and corresponding name. I did this for all 10 of my lists. The last opCount was then found when the last operation was performed and a counter printed. A graph was then derived from the number of the last entry in each of the list found in resultsInsert.

For Find(), I commented out the insert operations opCount, and only printed out operations on the find function. I then ran the program with parameters ie java AccessAVLApp “KHLPAL002”, I separated the experiment to

- **Worst Case**: A student ID I knew was not in the list, this would go through every ID before terminating
- **Average Case**: A student ID on the list but more or less in the middle, to show how many operation would be performed.

- **Best Case**: A student ID right on top, in the best case the least amount of operations are performed

I did this for all 10 lists and stored the results in the bestCase, worstCase and averageCase directories, I then used the last entry of the counter to draw a graph for each case. Best, average and worse.

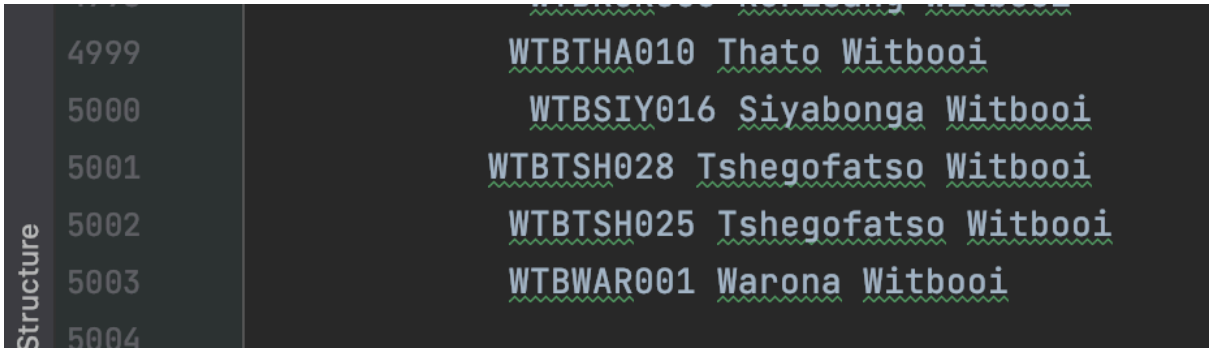
4. Part 1 values used

All results are stored in the resultsPart1 directory, inOrder traversal was used. The results consist of 1 print all students, 3 invalid inputs and 3 valid inputs. The screenshots of the results are as follows:

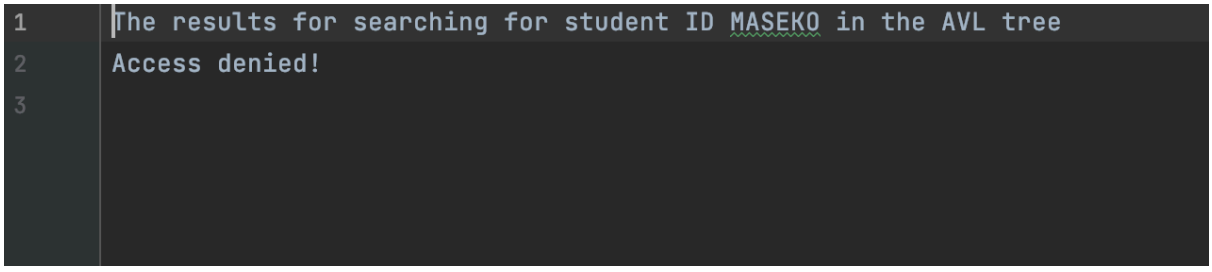
1. For all printing all students

   First 5 names

   Last 5 names



2. Invalid output 1



3. Invalid output 2

```
The results for searching for student ID nhllut014 in the AVL tree
Access denied!
```

4. Invalid output 3

```
The results for searching for student ID KHLPAL002 in the AVL tree
Access denied!
```

5. Valid output 1

```
The results for searching for student ID NHLLUT014 in the AVL tree
Student name: Luthando Nhlapo
```

6. Valid output 2

```
The results for searching for student ID DLMASE008 in the AVL tree
Student name: Asemahle Dlamini
```

7. Valid output 3

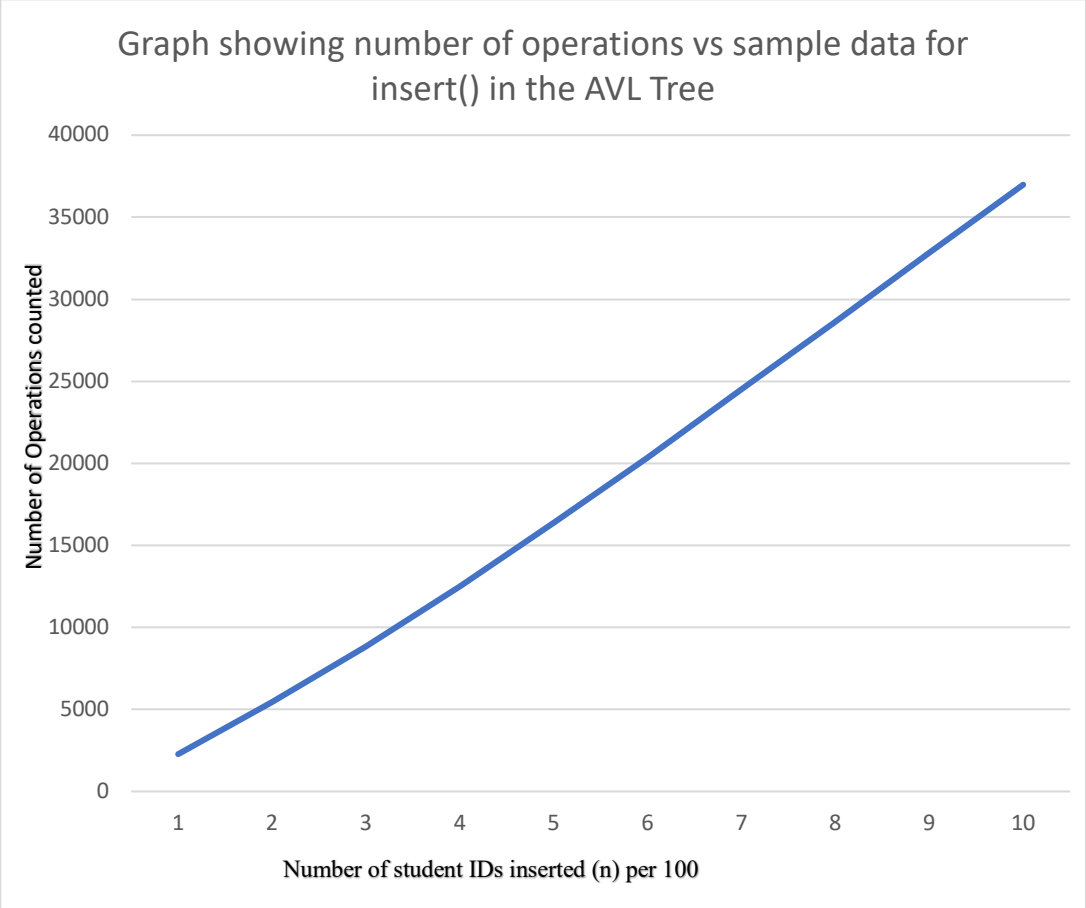
```
The results for searching for student ID WLLKUN002 in the AVL tree
Student name: Kungawo Williams
```

5. Part 2 Graphs and discussions

5.1. Insert()

Table showing the number of names vs the number of operations counted in the insert() graph

Number of names	Number of operations
100	2274
200	5425
300	8824
400	12513
500	16358
600	20363
700	24509
800	28604
900	32828
1000	36982



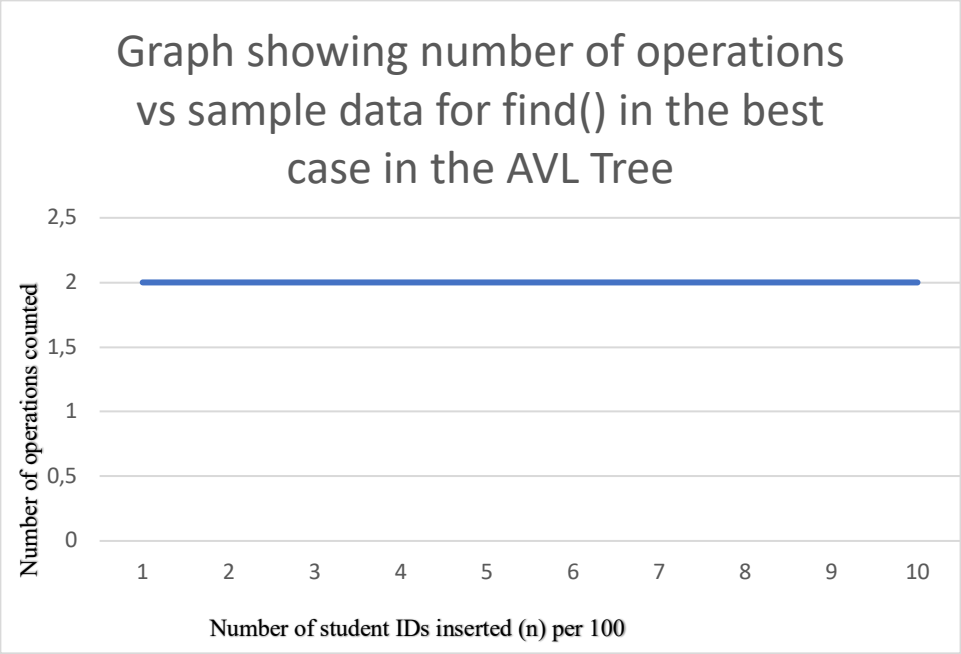
Discussion: The Insert function shoes an operation graph shows a logarithmic time complexity meaning the Big O notation is O(log(n)), this means that inserting the data will generally take O(log(n)) time for any n number of data.

5.2. Find()

Best case

Table showing the number of names vs the number of operations counted in the find() graph in the best case

Number of names	Student ID used	Number of operations
100	MLLNOA014	2
200	MLLNOA014	2
300	MLLNOA014	2
400	MLLNOA014	2
500	MLLNOA014	2
600	MLLNOA014	2
700	MLLNOA014	2
800	MLLNOA014	2
900	MLLNOA014	2
1000	MLLNOA014	2

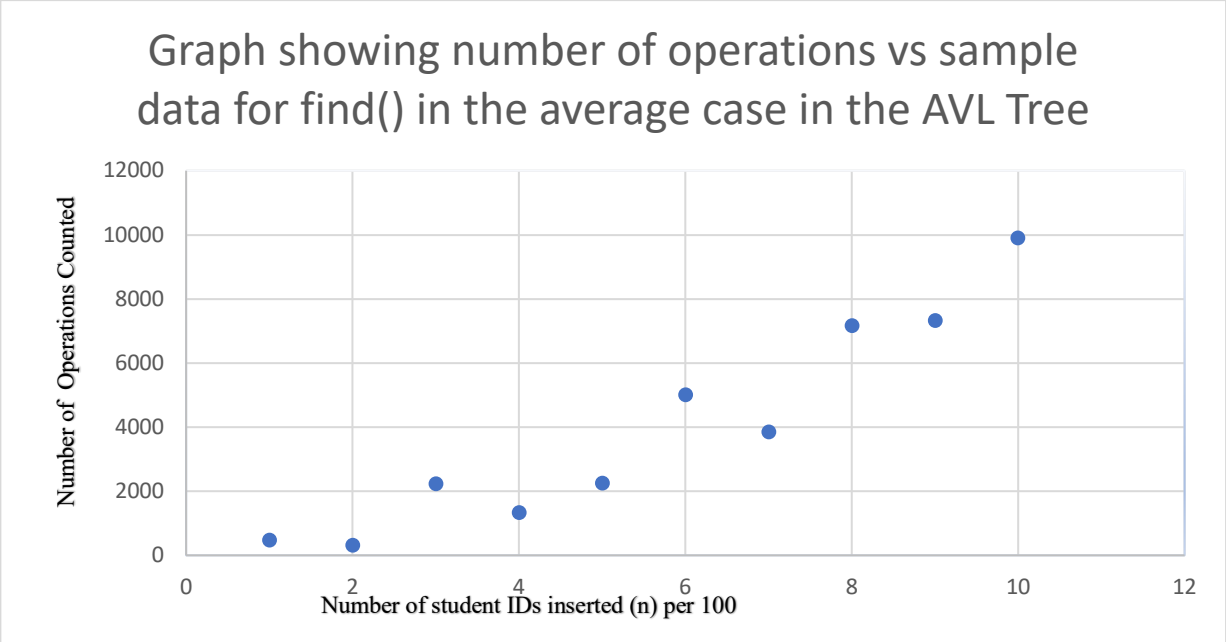


Discussion: In the best case the number of operation graph shows a constant time complexity meaning the Big O notation is O(1), this means that finding the data on top will always take O(1) time.

Average case

Table showing the number of names vs the number of operations counted in the find() graph in the average case

Number of names	Student ID used	Number of operations
100	MLTLUP014	480
200	TSTKAM038	310
300	LWXBOK037	2228
400	DNLOLE003	1342
500	MSXLET040	2254
600	STHPHE004	5006
700	MSKJOS006	3854
800	KHZLIK013	7163
900	LWXMIN010	7316
1000	JCBJOS004	9899



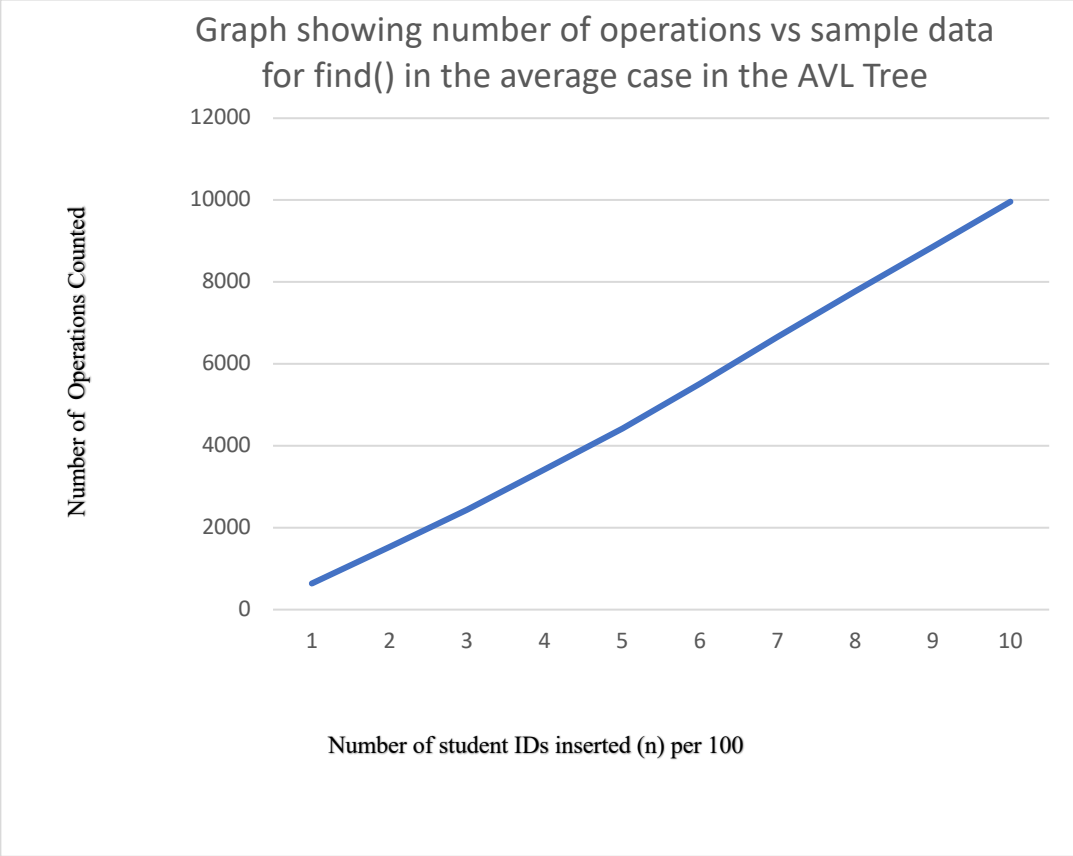
Discussion: If a general trendline were to be drawn, the resulting line would be a straight line, we can then say in the average case the number of operation graph shows a logarithmic time complexity meaning the Big O notation is O(log(n)), this means that finding the data will generally take O(log(n)) time

Worst case

Table showing the number of names vs the number of operations counted in the find() graph in the best case

Number of names	Student ID used	Number of operations
100	KHLPAL002	638
200	KHLPAL002	1533
300	KHLPAL002	2432
400	KHLPAL002	3426
500	KHLPAL002	4426
600	KHLPAL002	5525
700	KHLPAL002	6660
800	KHLPAL002	7762

900	KHLPAL002	8862
1000	KHLPAL002	9962



Discussion: In the best case the number of operation graph shows a logarithmic time complexity meaning the Big O notation is O(log(n)), this means that finding the data will generally take O(log(n)) time, this is the same as the average case, this means that AVL Tree is quite fast when searching for keys.

6. Creativity

My creativity came from the fact that I used unix redirection store all my results including Javadoc and every single other output in different directories.

7. Summary for Git Log

```
0: commit a0c6aa15911b4b5a463d7576410563a80b9266d4
1: Author: Palesa Khoali <palesakhoali@Setups-MBP.home>
2: Date:   Tue Apr 27 19:38:54 2021 +0200
3:
4: added gitLog.txt
5:
6: commit 6e4f7d228cdb167599752b2b5563166581a4573b
7: Author: Palesa Khoali <palesakhoali@Setups-MBP.home>
8: Date:   Tue Apr 27 19:33:21 2021 +0200
9:
...
67: Author: Palesa Khoali <palesakhoali@Setups-MBP.home>
68: Date:   Sat Apr 24 15:39:00 2021 +0200
69:
70: added data and the AccessAVLAPP
71:
72: commit 0f185a8726a360f862e6d5be7c64b155f41cbd66
73: Author: Palesa Khoali <palesakhoali@Setups-MBP.home>
74: Date:   Fri Apr 23 21:44:18 2021 +0200
75:
76: First commit
palesakhoali@Setups-MBP AVL %
```

8. Conclusion

The time complexity of the AVL Tree is generally O(Log n) for both inserting and finding nodes