CSC ASSIGNMENT 2 REPORT
STUDENT NUMBER: JDGHOL001

NB: PySimpleGUI needs to be installed on the tutor's computer to be able to run my code

MY OOP DESIGN

Students.java

Implements comparable and has a compareTo method which is used in AVLTree.java to compare the keys.

AccessAVLApp.java
Classes it uses: AVLTree.java, Students.java

Instantiates an AVLTree object using the Students class. The method initializeAVL() calls the insert() method from AVLTree.java to add each student one by one to the tree (reading from a text file). A variable called "counterInsert" in the AVLTree.java class is counting the number of key insertions needed to insert each student to the tree. This variable is retrieved by calling the AVLTree class. The integer is then tested to see if it is the minimum or maximum, and so that it can be added to the total count of insertion operations which will later be divided by the number of insertions to find the average.

The method treeOrder from AVLTree is called to print out all the students that were inserted into the tree. This method requires the Students class compareTo method.

The method find() from AVLTree.java is called to find the node in the AVL tree that contains the student ID that is passed through as a parameter. The method printStudent(student ID) in AccessAVLApp then performs string manipulation to return the student name associated with the given ID. A variable called "counterFind" in the AVLTree.java class is counting the number of key insertions needed to find the node containing the student ID given. This variable is retrieved by calling the AVLTree class. The integer is then tested to see if it is the minimum or maximum, and so that it can be added to the total count of insertion operations (which will later be divided by the number of insertions to find the average).

Experiment.java
Classes it uses: AccessAVLApp.java

This class takes in student ID's as arguments. It then calls the printStudent() method from AccessAVLApp on each of the arguments. The variable that is counting the number of find key comparisons (for the student passed in as a parameter) is accessed. This variable is compared to the maximum and minimum variables (that were originally both set to the number of find comparisons of the first argument) and added to the total number of find key comparisons that have been done. This total is divided by the total number of student ID's (arguments) to get the average number of key comparisons needed to call the find method on the arguments given.

The class returns a print statement stating the best, worst and average number of key comparisons required to insert and find student objects in an AVL tree the size of the selected subset.

GUI.py
Classes/scripts it uses: AccessAVLApp.java, testTrials.py

Implements the basic GUI that can be used to run the experiment on specific subsets of the oklist and can be used to test AccessAVLApp with and without parameters. If user selects to test the AccessAVLApp then AccessAVLApp is called when the button is clicked. If user selects to test the experiment, testTrials is called on the subset inputted by the user. More detail about this class is explained under creativity.

testTrials.py
Classes/scripts it uses: Experiment.java, GUI.py

Randomises data in oklist. Takes input from the user in the GUI class- this input is the size of the subset to be experimented on. Creates a new text file to write the subset data to and then copies this file into oklist so that the file used in AccessArrayApp is changed dynamically in this way. Student ID's are extracted from the subset and concatenated into a string which is passed in as an argument to the Experiment class. In the Experiment class, the printStudent(studentID) method is called on each of the student ID's (the arguments). When this is called, counterFind in AVLTree is incremented, then accessed in AccessAVLApp, and then accessed by the Experiment class to determine the minimum, maximum and average number of key comparisons required to find a student in an AVL tree of this specific subset. When testTrials calls Experiment.java, all the student names are printed out as well as the best, worst and average number of key comparisons required to insert and find a student object in an AVL tree. More detail can be found under creativity.

AVLTree.java
Classes it uses: Students.java, BinaryTree.java

This class contains the find(), insert() and treeOrder() methods that are called in AccessAVLApp.java. The code was written by Hussein Suleman but I adjusted it to include two variables (counterFind and counterInsert) which are used to count the number of key comparisons to find a student object in the tree and insert a student object into a tree respectfully. I also included a resetInsert() and resetFind() method to set the variables counterFind and counterInsert back to zero after each time the method find() or insert() is called (respectfully). These variables and methods are called in AccessAVLApp and are used to determine the worst, best and average number of key comparisons required to insert or find a student object within a specific subset.

BinaryTree.java uses BinaryTreeNode.java to make a node object in the AVL tree.

DESCRIPTION OF THE EXPERIMENT

The aim of the experiment was to test the theoretical time complexity of an AVL tree using a real-life implementation.

The experiment executed compared the number of key comparisons required to run AccessAVLApp on each parameter in a dataset to determine the minimum, maximum and average number of key comparisons. The best case being the minimum operations needed to

execute AccessAVLApp on a specific parameter passed (student ID) and the worst case being the maximum operations needed to be performed. Insert operations and find operations were counted separately so that they could also be compared. This experiment was repeated 10 times on 10 different sized data sets. This was done to see the relationship between an increasing dataset size and the effect that has on the best, average and worst cases of insert and find operations separately. Splitting the insert and find operations meant that the effect of an increasing dataset and the number of insert vs find key comparisons required could be compared and the overall trend of the results plotted onto a graph.

The trend of the number of insert and find operations across the 10 dataset sizes can then be compared to the expected, theoretical, trend that an AVL tree should show as the dataset increases. This 'trend' is called, in theory, the time complexity of an AVL tree (as the more operations required, the longer the tree takes to perform).

This meant the results of the experiment needed to then be compared to the theoretically expected worst case for insert and find operations in an AVL tree of a specific size (n)- that is O(logn).

TEST VALUES PART 1

Test values input:

1. CHKONT018
2. BTHAMO046
3. DMSMEL001
4. DLMORA019
5. MGLLET011

Test values output:

1. Onthatile Chauke
2. Amogelang Buthelezi
3. Melokuhle Adams
4. Oratilwe Dlamini
5. Lethabo Mogale

First 5 lines of print all:

1. MLLNOA014 Noah Maluleke
2. KHZOMA010 Omaatla Khoza
3. DMSMEL001 Melokuhle Adams
4. BXXBON021 Bonolo Booi
5. BRHKAT012 Katleho Abrahams

Last 5 lines of print all:

1. WTBTHA010 Thato Witbooi
2. WTBSIY016 Siyabonga Witbooi
3. WTBTSH028 Tshegofatso Witbooi
4. WTBTSH025 Tshegofatso Witbooi
5. WTBWAR001 Warona Witbooi

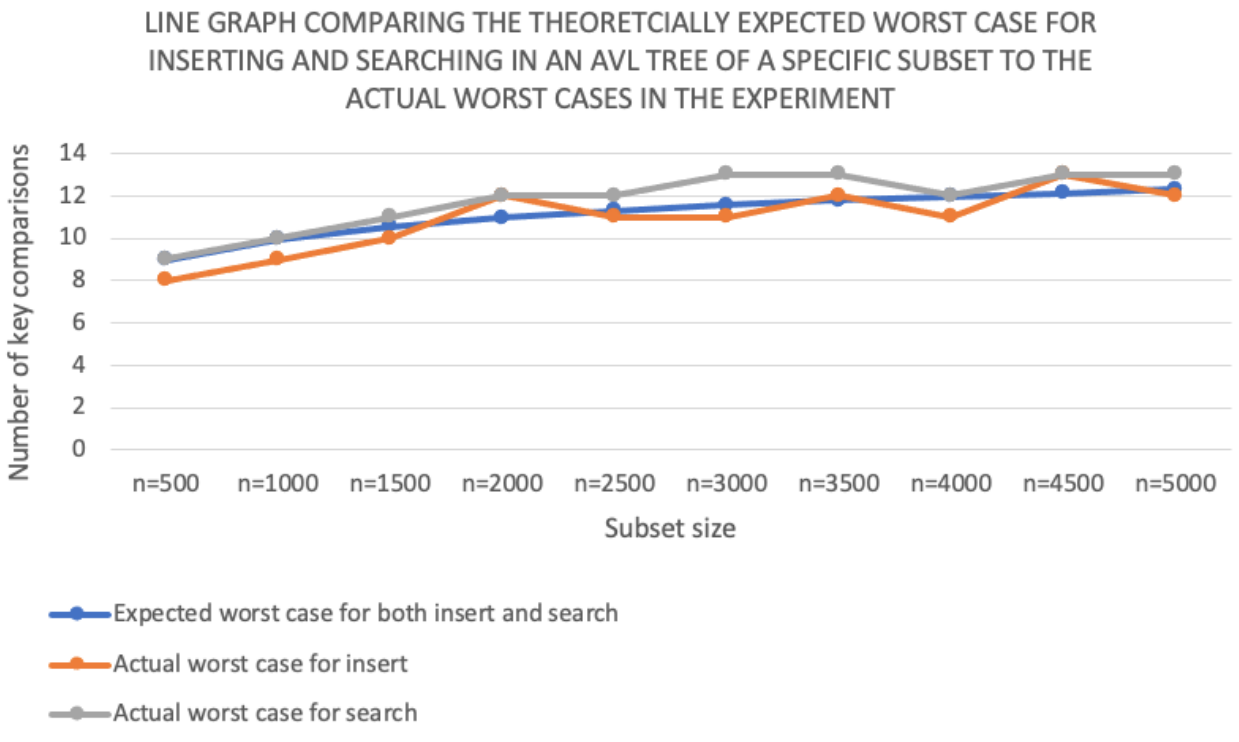
RESULTS

BEST, WORST AND AVERAGE NUMBER OF KEY COMPARISONS REQUIRED PER SUBSET FOR INSERTING INTO AN AVL TREE

	BEST	WORST	AVERAGE
N=500	0	8	2.81
N=1000	0	9	3.29
N=1500	0	10	3.68
N=2000	0	12	3.96
N=2500	0	11	4.03
N=3000	0	11	4.28
N=3500	0	12	4.48
N=4000	0	11	4.47
N=4500	0	13	4.40
N=5000	0	12	4.57

BEST, WORST AND AVERAGE NUMBER OF KEY COMPARISONS REQUIRED PER SUBSET FOR SEARCHING IN AN AVL TREE

	BEST	WORST	AVERAGE
N=500	1	9	4.45
N=1000	1	10	5.1
N=1500	1	11	5.48
N=2000	1	12	5.8
N=2500	1	12	6.08
N=3000	1	13	6.04
N=3500	1	13	5.84
N=4000	1	12	6.18
N=4500	1	13	6.21
N=5000	1	13	6.33



**Insert and search are both 0(logn) in an AVL tree hence only one line is used to represent both in the line graph*

DISCUSSION OF RESULTS

(Note: When re-running the program, numbers will vary slightly per subset due to the randomisation of oklist each time testTrials is run)

Theoretically the worst case (or highest number of key operations required) for searching and inserting into an AVL tree is represented by O(logn) for each dataset of size n. O(logn) is also theoretically the average case for inserting and searching in an AVL tree.

This is plotted on the line graph above. The results from my experiment are all within a very small range around the expected values. The greatest range being between the expected worst search case in subset n=3000 and the actual worst case. The expected number being 11.5 and the actual number being 13; therefore, the greatest range is 1.5. The smallest range is 0.03, this is between the expected worst case for search and the actual worst case for search in n=1000 and between the expected worst case for search and the actual worst case for search in subset n=4000. This proves that O(logn) is the correct formula to determine the worst case for inserting and deleted.

From a dataset of 500 to 5000, the worst case for insert and find both only increase by 4. This small increase in operations compared to the large increase in amount of data shows the effectiveness of an AVL tree in handling large amounts of data.

The worst numbers for both insert and find are very similar, the worst case across the 10 datasets both being 13. This means that this data structure is equally effective in searching and inserting. This can be placed in comparison with an array and a binary search tree- an array being very efficient for insertion but inefficient at searching and a binary search tree being slow at insertion but quicker with searching. The AVL tree, however, is equally effective at both finding and inserting. It is the balancing factor of an AVL tree that makes it more efficient than a binary search tree.

CREATIVITY

My experiment part of the assignment is implemented dynamically using the python scripts testTrials.py and GUI.py and the java class Experiment which ran from these scripts.

I didn't need to create separate text files for each subset and pass these into my program. Instead, I manipulated the oklist from within testTrials.py so that AccessAVLApp could be left untouched (i.e., I did not have manually to change the file used in AccessAVLApp).

In GUI.py, the user is asked to select AccessAVLApp or Test Trials. If AccessAVLApp is selected, a new window opens asking the user whether they want to print all students or input a specific student ID and return that students name. If they select the button to print all students, all the students are printed from the subset 5000 which is the default subset. If the user selects to input a student ID, a new window opens asking the user to input a student ID. When 'OK' is clicked the program returns the student name associated with the student ID. Thus, this GUI can be used to test my AccessAVLApp class.

However, if Test Trials is selected, a new window opens prompting the user to choose a subset (n) to run the experiment on. The user enters a number between 1 and 5000 and this number is passed into testTrials.py as a parameter. It should be noted that this means that the subset can be of any size (whereas in my previous assignment there was only the options of n=500, n=1000 etc …)

The number given by the user is used in testTrials.py to create a subset of n from oklist (which has, at this point, been randomised). This subset, which is saved to a text file, is then copied into oklist. Now, when oklist is used in AccessAVLApp, it is not the full list of 5000 used but a specific subset.

Still in testTrials.py, the student numbers extracted from the text file are concatenated into a string which is passed into Experiment.java, each string being an argument. The student numbers need to be passed as arguments so that the method printStudent from AccessAVLApp (which is called in Experiment) receives a student ID. testTrials then invokes the os function so the command 'java Experiment' plus the arguments from concatenated string is executed, returning the results of the experiment when performed on this specific subset (that is, the best, worst and average case for find and insert methods).

Finally, oklist is reset to its original form as a copy was made prior to manipulating it.

GIT USAGE LOG

0: commit 67842daf29ad809c830e823d5634d8fd22248312
1: Author: Holly Judge <jdghol001@myuct.ac.za>
2: Date: Mon Apr 26 20:13:47 2021 +0200
3:
4: final project- adjustment made was additional commenting for javadocs
5:
6: commit cd6e2f7e3635160413464e75386962aa908c3100
7: Author: Holly Judge <jdghol001@myuct.ac.za>
8: Date: Mon Apr 26 18:44:14 2021 +0200
9:
...
43: Author: Holly Judge <jdghol001@myuct.ac.za>
44: Date: Thu Apr 22 21:49:45 2021 +0200
45:
46: git repo was corrupted to had to reset it to a previous git version saved remotely. updates include dynamically producing subsets allowing for worst,best and average case to be calculated without outputting all the numbers for each student ID. since last commit which was corrupted so now cannot be seen i have fixed my insert method as i was not reseting the counter to 0 before adding a new number
47:
48: commit c91331e5e17ca825e3ce5cc20be0aa3f7ffac6c1
49: Author: Holly Judge <jdghol001@myuct.ac.za>
50: Date: Mon Apr 19 19:30:14 2021 +0200
51:
52: first version- contains conversion from assignment 1 to 2 and an upgrading of the automation process