CSC2001F Assignment 2

Edson Shivuri SHVNKA005

Class Structure, Interaction and OOP design:

Driver Classes:

LSAVLapp – Class that runs the AVL solution. It stores the AVL tree to be used and calls the UserMenu class to execute the finding and printing methods for the AVL tree.

LSBSTApp – Class that runs the BST solution. Like the LSAVLapp, it stores the BST tree and calls the UserMenu class to execute the finding and printing methods for BST.

UserMenu – Main driver class. It provides a text-based interface in the form of menu to allow the user to select the functionality they want to use. It also contains the printAreas() and printAllAreas() methods.

Data Structure Classes:

BTNode – Node class for a binary tree that stores a ScheduleItem object as well as a reference to the child nodes and keeps track of its height.

BT – Super Class for Binary trees. It therefore contains methods to be used by different types of binary trees. In this case, both the BST and AVL classes inherit from it.

AVL – Class that extends the BT class and includes find and insert methods that recursively add new data or search for an item in the AVL tree. Also contains methods to balance the tree.

BST – Class that extends the BT class and includes find and insert methods that recursively perform their tasks of respectively adding new data to the BST or searching for a specific element

Miscellaneous Classes:

CommonMethods - Static Class containing key generation and key splitting methods to be used by other classes.

ReadFile – Class To handle file reading in both the LSBSTApp and LSAVLapp classes. Adds data items to the correct data structure based on which class called this class's constructor and allows the LSAVLapp and LSBSTApp classes to populate their data structures.

ScheduleItem - Class to hold the data for a load shedding item, I.e., the key and the areas. Contains methods for accessing data within and instances of this class are placed into the data structures.

SearchItem– Class that stores two variables: one for the area to search for and one for a stage to search for. This is used to be able to find specific loadshedding data for an area.

NOTE: See Annex 1 for complete UML class description showing class interactions as well

Experiment Description

Aim:

The aim of this experiment is to compare the number of operations performed by a BST when compared to an AVL tree by making use of loadshedding data. The performance of each data

structure is measured by counting the number of comparisons made when searching for a specific item. Instrumentation was used in the code to count comparisons. Subsets of the data have been generated using a python script (see runScript.py). For each line of data in each subset, the LSAVLapp and the LSBSTApp are called to find that data item in their respective data structures. The comparison counts for finding each data item is then captured. From the captured counts, the best, worse and average case is determined for each subset (see testScript.py). Graphs are then generated from the data for comparison (see graphgeneration.py).

Trial Test Values and Outputs (Part 2: BST):	Trial Test Values and Outputs (Part 4: AVL):
Input: 1 1 00 Stage: 1, Day: 1, Start Time: 00:00 Areas Affected: 1 Number of insert operations: 357104 Number of find operations: 1 Tree Height: 260 **Input: 6 29 04** Stage: 6, Day: 29, Start Time: 04:00 Areas Affected: 6, 14, 2, 10, 3, 11 Number of insert operations: 357104 Number of find operations: 174 Tree Height: 260 **Input: 8 25 22** Stage: 8, Day: 25, Start Time: 22:00 Areas Affected: 14, 6, 10, 2, 15, 7, 11, 3 Number of insert operations: 357104 Number of find operations: 255 Tree Height: 260 **First 10 Lines of PrintAllAreas():** Stage: 1, Day: 1, Start Time: 00:00 Areas Affected: 1 Stage: 1, Day: 1, Start Time: 02:00 Areas Affected: 2 Stage: 1, Day: 1, Start Time: 04:00 Areas Affected: 3 Stage: 1, Day: 1, Start Time: 06:00 Areas Affected: 4 Stage: 1, Day: 1, Start Time: 08:00 Areas Affected: 5 Stage: 1, Day: 1, Start Time: 10:00 Areas Affected: 6 Stage: 1, Day: 1, Start Time: 12:00 Areas Affected: 7 Stage: 1, Day: 1, Start Time: 14:00 Areas Affected: 8 Stage: 1, Day: 1, Start Time: 16:00 Areas Affected: 9 Stage: 1, Day: 1, Start Time: 18:00 Areas Affected: 10 **Last 10 Lines of PrintAllAreas()** Stage: 8, Day: 31, Start Time: 04:00 Areas Affected: 6, 14, 2, 10, 3, 11, 15, 7 Stage: 8, Day: 31, Start Time: 06:00 Areas Affected: 7, 15, 3, 11, 4, 12, 16, 8 Stage: 8, Day: 31, Start Time: 08:00 Areas Affected: 8, 16, 4, 12, 5, 13, 1, 9 Stage: 8, Day: 31, Start Time: 10:00 Areas Affected: 9, 1, 5, 13, 6, 14, 2, 10 Stage: 8, Day: 31, Start Time: 12:00 Areas Affected: 10, 2, 6, 14, 7, 15, 3, 11 Stage: 8, Day: 31, Start Time: 14:00 Areas Affected: 11, 3, 7, 15, 8, 16, 4, 12 Stage: 8, Day: 31, Start Time: 16:00 Areas Affected: 12, 4, 8, 16, 9, 1, 5, 13 Stage: 8, Day: 31, Start Time: 18:00 Areas Affected: 13, 5, 9, 1, 10, 2, 6, 14 Stage: 8, Day: 31, Start Time: 20:00 Areas Affected: 14, 6, 10, 2, 11, 3, 7, 15 Stage: 8, Day: 31, Start Time: 22:00 Areas Affected: 15, 7, 11, 3, 12, 4, 8, 16	**Input: 1 10 00** Stage: 1, Day: 10, Start Time: 00:00 Areas Affected: 15 Number of insert operations: 72668 Number of find operations: 9 Tree Height: 17 **Input: 6 25 18** Stage: 6, Day: 25, Start Time: 18:00 Areas Affected: 12, 4, 8, 16, 13, 5 Number of insert operations: 72668 Number of find operations: 10 Tree Height: 17 **Input: 8 9 22** Stage: 8, Day: 9, Start Time: 22:00 Areas Affected: 14, 6, 10, 2, 15, 7, 11, 3 Number of insert operations: 72668 Number of find operations: 13 Tree Height: 17 **First 10 Lines of printAllAreas():** Stage: 1, Day: 1, Start Time: 00:00 Areas Affected: 1 Stage: 1, Day: 1, Start Time: 02:00 Areas Affected: 2 Stage: 1, Day: 1, Start Time: 04:00 Areas Affected: 3 Stage: 1, Day: 1, Start Time: 06:00 Areas Affected: 4 Stage: 1, Day: 1, Start Time: 08:00 Areas Affected: 5 Stage: 1, Day: 1, Start Time: 10:00 Areas Affected: 6 Stage: 1, Day: 1, Start Time: 12:00 Areas Affected: 7 Stage: 1, Day: 1, Start Time: 14:00 Areas Affected: 8 Stage: 1, Day: 1, Start Time: 16:00 Areas Affected: 9 Stage: 1, Day: 1, Start Time: 18:00 Areas Affected: 10 **Last 10 lines of printAllAreas():** Stage: 8, Day: 31, Start Time: 04:00 Areas Affected: 6, 14, 2, 10, 3, 11, 15, 7 Stage: 8, Day: 31, Start Time: 06:00 Areas Affected: 7, 15, 3, 11, 4, 12, 16, 8 Stage: 8, Day: 31, Start Time: 08:00 Areas Affected: 8, 16, 4, 12, 5, 13, 1, 9 Stage: 8, Day: 31, Start Time: 10:00 Areas Affected: 9, 1, 5, 13, 6, 14, 2, 10 Stage: 8, Day: 31, Start Time: 12:00 Areas Affected: 10, 2, 6, 14, 7, 15, 3, 11 Stage: 8, Day: 31, Start Time: 14:00 Areas Affected: 11, 3, 7, 15, 8, 16, 4, 12 Stage: 8, Day: 31, Start Time: 16:00 Areas Affected: 12, 4, 8, 16, 9, 1, 5, 13 Stage: 8, Day: 31, Start Time: 18:00 Areas Affected: 13, 5, 9, 1, 10, 2, 6, 14 Stage: 8, Day: 31, Start Time: 20:00 Areas Affected: 14, 6, 10, 2, 11, 3, 7, 15 Stage: 8, Day: 31, Start Time: 22:00 Areas Affected: 15, 7, 11, 3, 12, 4, 8, 16

Table 1: Successful Trials

Invalid BST Inputs	**Invalid AVL inputs**
input: 12 4 24 Areas not found Number of insert operations: 357104 Number of find operations: 260 Tree Height: 260 **Input: 4 24** Invalid Call, please try again **Input: 1 15 28** Areas not found Number of insert operations: 357104 Number of find operations: 28 Tree Height: 260	**Input: 2 56 4** Areas not found Number of insert operations: 72668 Number of find operations: 15 Tree Height: 17 **Input: 2 4** Invalid Call, try again **Input: 2 4 91** Areas not found Number of insert operations: 72668 Number of find operations: 13 Tree Height: 17

Table 2: Unsuccessful trials

Creativity in Application

Creativity comes in with the ReadFile class. Instead of including file reading functionality into the LSBSTApp and LSAVLapp classes, ReadFile checks the stack to determine which class called it, triggering a flag that signals which data structure the data should be added to. Since multiple classed require the making and breaking of keys from the args passed into the main methods or user input, a helper class (CommonMethods) was created to avoid rewriting code. Python scripts were used to create subsets of data from the load shedding schedule as well as to automate their run. Furthermore, UML class diagrams were used to show relationships between classes.

Furthermore, the UserMenu class was added into the program. It provides an interface for the user to be able to interact with the program and can run by being called by either the LSBSTApp or LSAVLapp in a similar fashion to the ReadFile class. In addition to that, the UserMenu contains the printAreas() and printAllAreas() methods to facilitate code re-use as well. In addition, the functionality of being able to search for all the scheduled loadshedding times for a specific area and specific time has been added. This was achieved by doing the following: 1) Adding the SearchItem class to hold the area and stage to look for, 2) Modifying the ScheduleItem class to be able to see what must be searched for so that each ScheduleItem object can check itself to see if it meets the search criteria. If there is an item to be searched for, then calling the toString only returns the data the object has met the search criteria. Note that the values stored in the ScheduleItem class are only changed in the UserMenu class and must be reset after the search.

Summary of git usage (empty log lines omitted from first and last 10)

From Terminal:

0: commit 655e84f2bc2ba67bcbd34ec0b13a20f6f104c477

1: Author: VoidMaster23 <edsonshivuri@gmail.com>

2: Date: Mon Apr 20 23:05:42 2020 +0200

3:

4: Most Done, report and review

5:

6: commit 930ec596484439fc709c3fe0f99129d532022cac

7: Author: VoidMaster23 <edsonshivuri@gmail.com>

8: Date: Tue Apr 14 19:48:30 2020 +0200

9:

...

91: Author: VoidMaster23 <edsonshivuri@gmail.com>

92: Date: Wed Mar 11 16:02:47 2020 +0200

93:

94: Set up complete

95:

96: commit ae691752cec9a94599d693b6bad5289df94a86ab

97: Author: VoidMaster23 <edsonshivuri@gmail.com>

98: Date: Tue Mar 10 20:41:29 2020 +0200

99:

100: Yeh

Graphs and Discussion

Insertion Count

Though the experiment was only counting the number of find operations, we can still compare the insertion counts of the two data structures. Looking at Table 1 above we can see that for each of the trials done the BST insert count was 357104, while for the AVL tree it was 72668. The data used to obtain those numbers was, however, ordered, so the BST was operating at its worst case (linear tree). However, in the AVL, despite the data being ordered the insertion count is a lot lower simply because the AVL will is still balanced in the worst case, so there are not as many comparisons to make when inserting a new node in the correct position.

Experiment Results

The following graphs and results were obtained from the experiment using ten unordered subsets of the loadshedding schedule data of varying sizes and finding the best, worst and average counts for each one.

Best Case Results

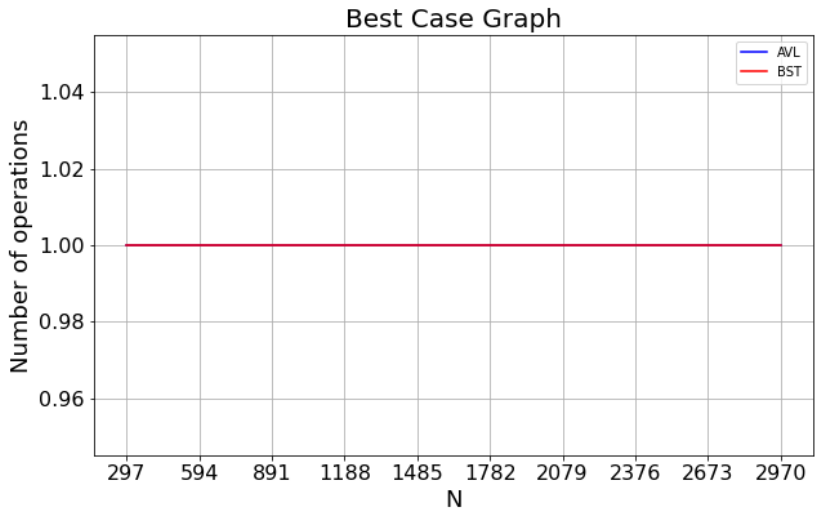

Figure 1: Best Case Graph

For both the AVL and BST the best case would be searching for the root node. Since the only comparison being made in that case is checking if the root node is the item that is being searched for, the operation count will always be 1. So, at the best case both AVL and BST is O (1).

Worst Case Results

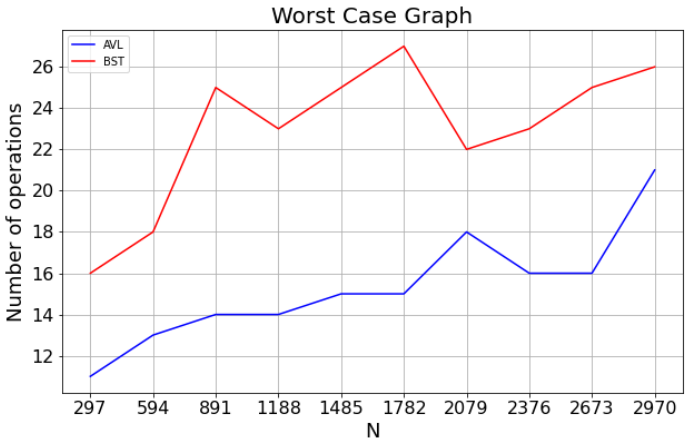

Figure 2: Worst Case Graph

N	**297**	**594**	**891**	**1188**	**1485**	**1782**	**2079**	**2376**	**2673**	**2970**
AVL	11	13	14	14	15	15	18	16	16	21
BST	16	18	25	23	25	27	22	23	25	26

Table 3: Worst case results for the experiment

For each value of N, both the AVL and BST were supplied with the same data sets. So, what we see is a comparison between searching for data in a balanced tree vs an unbalanced tree. Notice that that at every point the AVL performs less operations for find data items than the BST. Since the data is unordered, the BST is not operating at its worst case of O(N), i.e. it is not a linear tree. The fluctuations in operation counts for the BST stems from the fact that the BST is unbalanced, so the increase in counts is attributed to the fact that the unordered data causes a tree to have varying heights. However, for the AVL tree we see a much smoother curve which comes from the fact that at the worst case, a leaf node will be the one that is being searched for,

and the balancing operating ensures the minimum tree height. The time complexity seen in the AVL in practice here is in the range O (log N) and O(1.44*log N) . Note that the operation count is related to the height of the tree. For a tree of height h , at the worst case we make h + 1 comparisons. So since the AVL is a balanced BST using the same data would result in a tree that has a smaller height thus the number of operations is always going to be lower in the AVL than BST.

Average Case Results

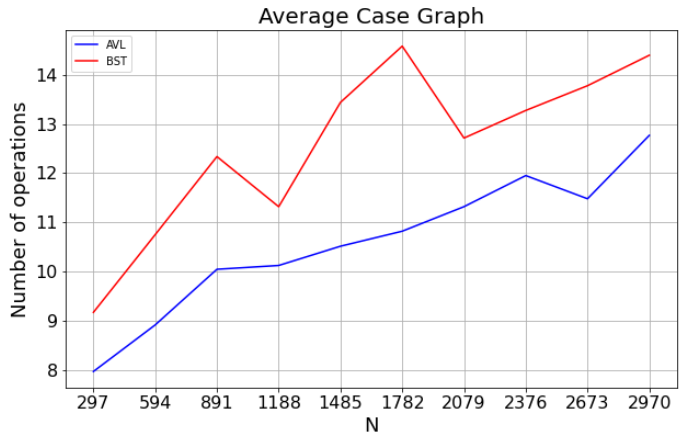

Figure 3: Average Case Graph

N	297	594	891	1188	1485	1782	2079	2376	2673	2970
AVL	7.97	8.92	10.04	10.12	10.51	10.82	11.32	11.95	11.48	12.77
BST	9.17	10.75	12.34	11.32	13.44	14.58	12.71	13.28	13.78	14.40

Table 4: Worst case results for the experiment

The average case was obtained by determining the average of all operation counts across the different data sets. From the above, we can see that the average case plots have the same relationship and the worst case. From the above it can be seen that on average the AVL is closer to O (log N) than the BST.

Conclusions

From the above discussions it can be seen that the AVL tree performs better than the BST. From the lower insertion counts, to better searching performance when it comes to finding data in the worst and average cases. This is because the AVL is always self-balancing so the number of comparisons that must be made for each operation is lower than in a BST. If the data were ordered, however the BST would have an even worse performance since it would degenerate into a linear structure.

Annex 1: UML Class Structure