Using an AVL Tree data structure to test the time complexities of AVL Search and Insert algorithms

CSC2001F Assignment 2

Zuhayr Halday (HLDZUH001)

3/22/24

Overall OOP Design

GenericsKbAVLApp

In the program written for this assignment, Object-Oriented Programming (OOP) principles were extensively utilized to implement an AVL tree data structure. The program two classes: *AVLNode* and *AVLTree*, along with a main class *GenericsKbAVLApp*.

The *AVLNode* class was written to represent a single node in the AVL tree and encapsulates the data stored in the node, along with references to its left and right child nodes. This class demonstrates encapsulation, as it encapsulates the node's data and provides methods to access and manipulate it.

The *AVLTree* class was written to represent the AVL tree itself and encapsulates the tree's root node. It provides methods for inserting data into the tree, searching for data, and performing tree balancing operations and rotations. This class showcases encapsulation, abstraction, and modularity, as it encapsulates the tree's implementation details, abstracts away complex tree operations, and organizes functionality into separate methods.

The *GenericsKbAVLApp* main class written for the program serves as the main entry point for the program, orchestrating interactions with the AVL tree. It prompts the user for the names of input files containing the knowledge base which must be stored and queries to search the knowledge base for, reads data from files, constructs the AVL tree, performs searches, and displays search statistics. This class illustrates encapsulation, abstraction, and inheritance, as it encapsulates the program's main functionality and inherits functionality from the AVL tree class.

As seen above in each of the classes written for this program, the GenericsKbAVLApp program effectively demonstrates the use of a variety of OOP principles. Encapsulation, abstraction, modularity, and inheritance were all implemented in order to create a program with an efficient and maintainable AVL tree data structure.

Experimentation Program

A separate Experimentation Program was written purely to test the time complexity of the AVL tree’s insert and search algorithms by varying the size of the datasets stored in the AVL tree and searched through. All of the aforementioned OOP principles related to the AVL tree classes were all similarly implemented in this program.

While the *AVLNode* and *AVLTree* classes were implemented exactly the same, the *Experimentation* main class was significantly different in comparison to the main class of the previous program. This main class contains methods written to generate random subsets of data (of specific sizes) from a larger knowledge base, and iteratively read these into the AVL tree, search for specific queries within each subset, and calculate and measure the best, worst, and average cases for the time complexities of both the AVL Insert and AVL Search algorithms.

AVL Tree Experimentation

Goals of the experiment

The goal of this experiment was to accurately measure and calculate the Big O time complexity of inserting data into an AVL tree and then searching for data in said AVL tree. In order to conduct a fair experiment, the dataset for which the AVL tree will be storing cannot be the same during every test. This experiment was conducted in order to prove that the time complexities of both searching and inserting into the AVL tree matches what is expected (as shown below):

Insert:

- Best case: O(1)
- Average case: O(log n)
- Worst case: O(log n)

Search:

- Best case: O(1)
- Average case: O(log n)
- Worst case: O(log n)

Method of experimentation:

Instead of repeatedly running the *GenericsKbAVLApp* with different datasets manually, an *Experimentation* program was written in order to automate the process. This program made use of the exact same AVL tree data structure, as well as the same search and insert algorithms as the original program, but the main functionality of the program was changed in order to conduct the experiment.

Manipulation of the dataset:

To calculate the time complexities of varying dataset sizes, the program creates 10 datasets of different sizes to be read into the AVL tree. The size of these datasets is equally spaced on a logarithmic scale, in order to better illustrate whether the time complexities correspond to that of log(n) or not. These sizes were: 1, 5, 10, 50, 100, 500, 1000, 5000, 10 000, and 50 000.

In order to maintain a fair experiment, each different dataset contains different data, all selected randomly from the GenericsKB knowledge base. An ArrayList of all data from the knowledge base was made, and using randomly generated index integers, randomized data was added to a new AVL tree for each dataset size. This was done in order to truly determine the best, worst and average time complexities of any dataset.

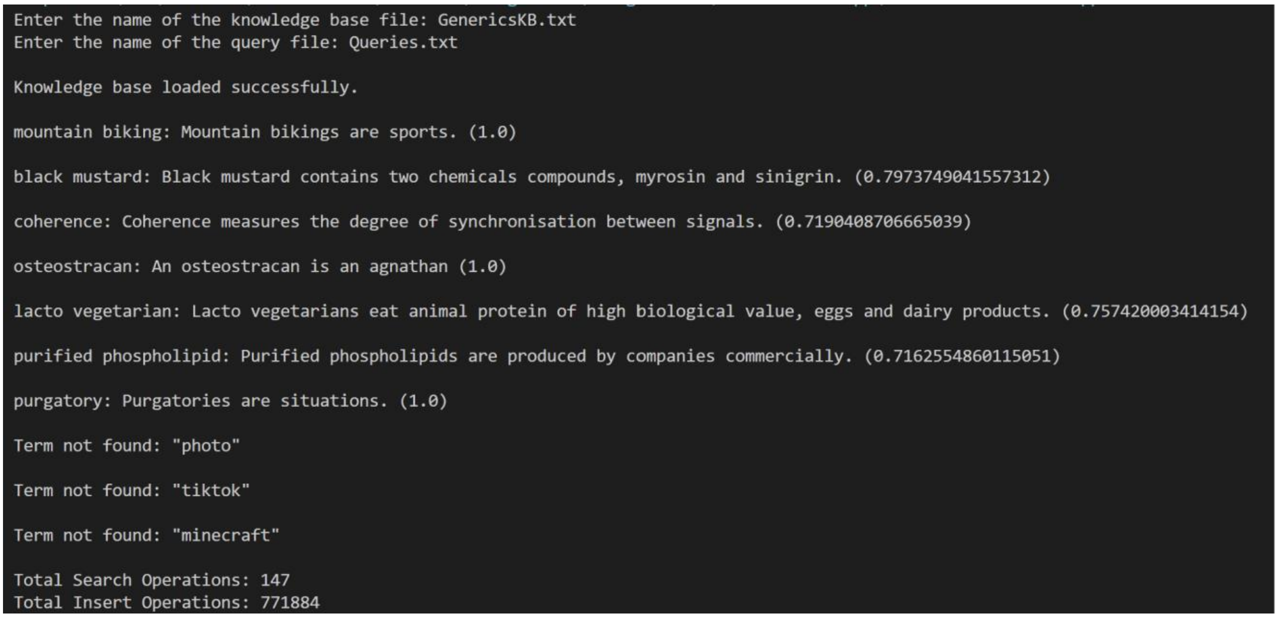
Calculating Best, Worst and Average cases:

To apply the instrumentation to the AVL tree, two separate counter variables were incremented for every comparison made in order to count the number of comparison operations that were run when inserting a term into the AVL tree and searching for a term in the AVL tree. This is because comparisons can be considered to be a computationally expensive process, thus notable when determining time complexity.

The counter variables for both inserting and searching were then stored in separate ArrayLists after each insert and search iteration. To calculate the theoretical best, worst and average cases for each algorithm, the smallest value in each ArrayList corresponded to the smallest number of comparisons made, thus the best case. Similarly, the largest value corresponds to the most comparisons made, thus the worst case. And finally, the mean average value of each ArrayList was calculated and thus, the average case for each dataset was calculated.

Test query values and outputs

To test the *GenericsKbAVLApp* program, 10 query terms were chosen: **mountain biking**, **black mustard**, **coherence**, **osteostracan**, **lacto vegetarian**, **purified phospholipid**, **purgatory**, **photo**, **tiktok**, and **minecraft**. The output of the program for each query term was as follows:

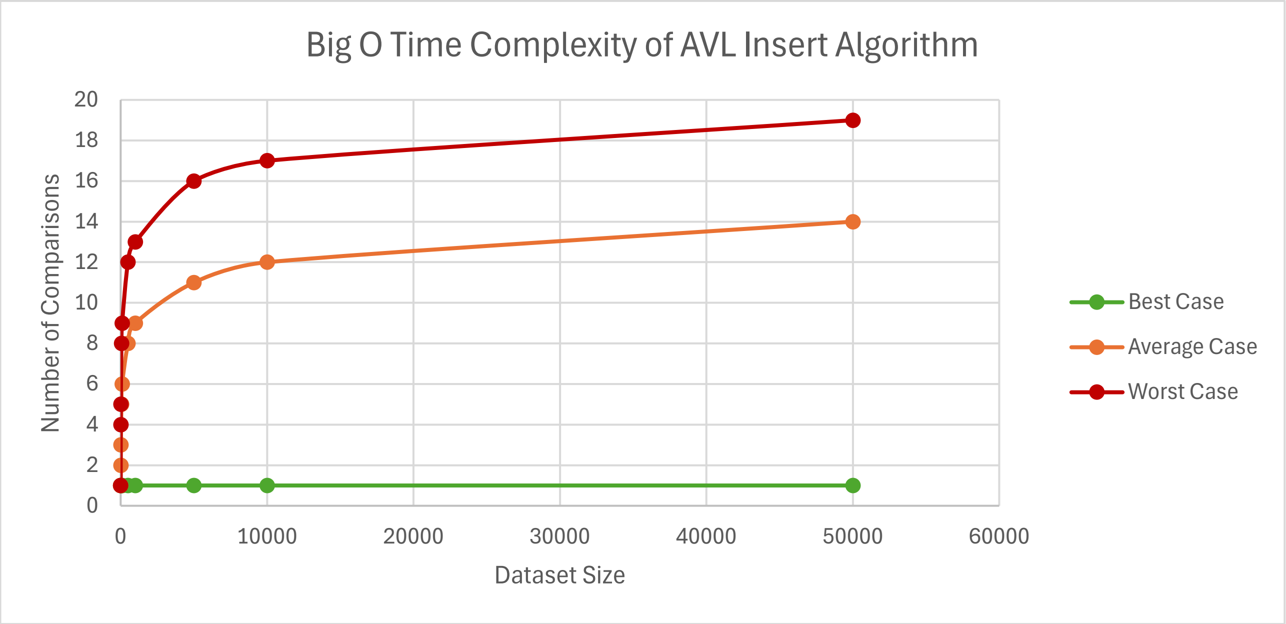


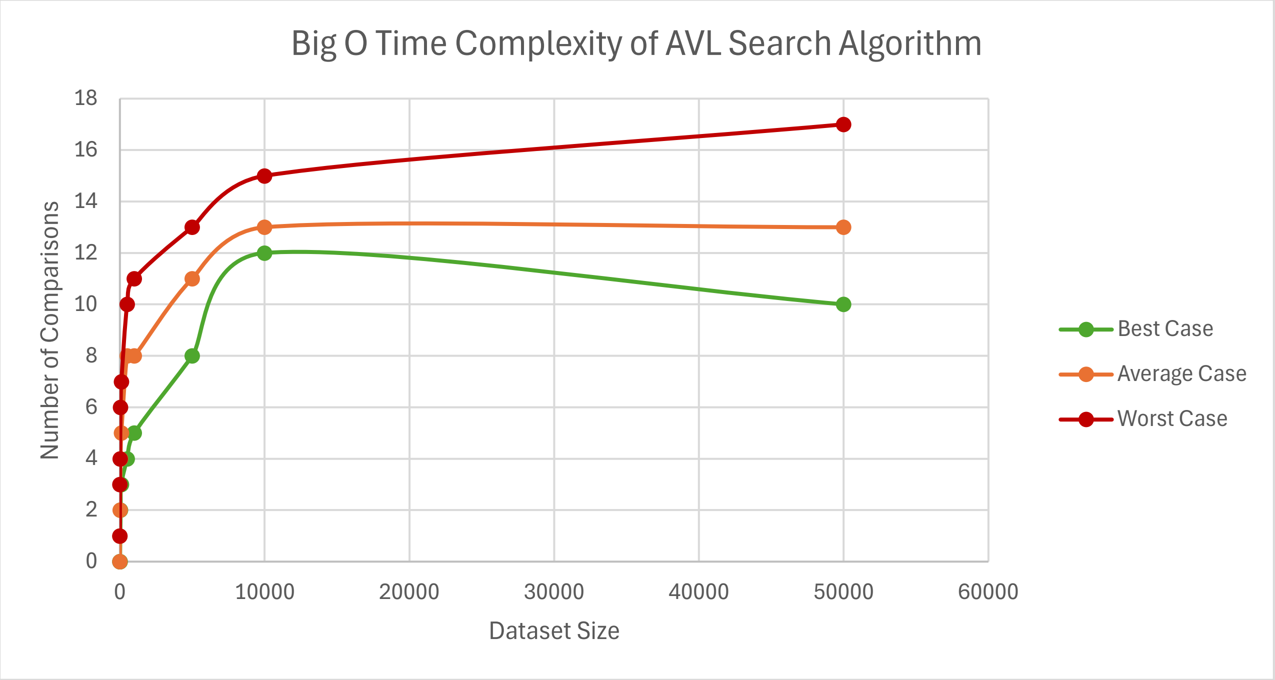
```
Enter the name of the knowledge base file: GenericsKB.txt
Enter the name of the query file: Queries.txt

Knowledge base loaded successfully.

mountain biking: Mountain bikings are sports. (1.0)

black mustard: Black mustard contains two chemicals compounds, myrosin and sinigrin. (0.7973749041557312)

coherence: Coherence measures the degree of synchronisation between signals. (0.7190408706665039)

osteostracan: An osteostracan is an agnathan (1.0)

lacto vegetarian: Lacto vegetarians eat animal protein of high biological value, eggs and dairy products. (0.757420003414154)

purified phospholipid: Purified phospholipids are produced by companies commercially. (0.7162554860115051)

purgatory: Purgatories are situations. (1.0)

Term not found: "photo"

Term not found: "tiktok"

Term not found: "minecraft"

Total Search Operations: 147
Total Insert Operations: 771884
```

Instrumentation Experiment Results:

The results for the experiment were printed to an *experimentation.txt* text file. These values were then copied into Excel, and plots of the Big O time complexities were generated:

Size of Dataset	Best Case for Insert	Average Case for Insert	Worst Case for Insert	Best Case for Search	Average Case for Search	Worst Case for Search
1	1	1	1	0	0	1
5	1	2	4	0	1	3
10	1	3	5	0	2	4
50	1	5	8	2	4	6
100	1	6	9	3	5	7
500	1	8	12	4	8	10
1000	1	9	13	5	8	11
5000	1	11	16	8	11	13
10000	1	12	17	12	13	15
50000	1	14	19	10	13	17





Discussion of Experiment Results

As shown in the above graphs, both the Insert and Search algorithms of the AVL tree have a worst case and average case time complexity similar to O(log n), as the log curves in each graph can easily be seen. While the best case time complexity for the AVL Insert does in fact correspond to O(1), the same cannot be said for that of the AVL Search.

While O(1) is theoretically the actual best case time complexity for an AVL Search, in practice there are many occasions where a randomized dataset will not have a search term as the first node of an AVL tree, especially for extremely large datasets. This can be seen in the graphs above, as the experiments show the best case of an AVL Search to also be roughly O(log n).

Despite this minor difference, this experiment can be considered a success. All fair-testing practices were maintained, and the practical results of these calculated time complexities almost all agree with the theoretical Big O time complexities for each algorithm.

Implementation of Creativity

While conducting the experiment by manually changing the dataset for each test was always an option, creativity was used when writing a completely new Java program to assist in automating the entire process in order to create a fair and efficient experiment. Using ArrayLists in order to calculate the minimum, maximum and average number of comparisons greatly improved the accuracy of results and writing these results neatly to an *experimentation.txt* file was also going above and beyond what was required. The implementation of these creative additions not only added more content to the assignment, but also served a functional role in obtaining extremely accurate results from the experiment.

Git Logs

First 10 Commits

```
1.  commit 9ff7ac01ddd70a8cea121bc0b4ef949c5d81708c
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date:   Tue Mar 19 12:19:18 2024 +0200

        First Commit
2.  commit 9fddc563c1c52c425f8d68de70faa29cabbb1c4e
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date:   Fri Mar 22 01:47:42 2024 +0200

        Added .txt files
3.  commit 7e07c544720b1161ef93dcf5af5202f898b3460a
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date:   Fri Mar 22 01:52:08 2024 +0200

        Added AVL Node Structure
4.  commit 46ffaed9d4917f5e1b87e70422d78b1d6257efcb
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date:   Fri Mar 22 01:54:24 2024 +0200

        Added a AVL Tree structure and constructor for a new AVL data structure
5.  commit fe2ffa4b2f372650a67ee88aeac95e5cdcc7f8bf
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date:   Fri Mar 22 02:06:53 2024 +0200

        Added rotation functions for the AVL tree structure from lecture notes.
6.  commit 56babcd4f5351701ffccd8b256aa57f3b337c952
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date:   Fri Mar 22 02:21:44 2024 +0200

        Added utility functions and balance function to balance the AVL tree.
7.  commit 459930a225ca65ba83a4eb0bee6939462007c3eb
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date:   Fri Mar 22 02:40:52 2024 +0200

        Added insert functionality to the AVL tree class, with instrumentation to count comparisons.
8.  commit 22996dbf2e131bcbee73b0ac159bd5a40d01ab54
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date:   Fri Mar 22 02:53:32 2024 +0200

        Implemented AVL search algorithm
9.  commit c8eca7d737913d1dba85035cc06c54dc8af8542a
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date:   Fri Mar 22 02:55:39 2024 +0200

        Added getter functions to AVL tree data structure.
10. commit 7cf48951724171d11b8b17b635ba0d3a84ea9f69
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date:   Fri Mar 22 03:03:49 2024 +0200

        Search function now prints data to terminal output.
```

Last 10 Commits

16. commit 564e640510825922f20e3a283e02d83f04f5964e
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date: Fri Mar 22 03:54:32 2024 +0200

    Bug fixes with Experimentation. AVL Tree class in Experimentation was being confused with that of the main program.
17. commit 7c66f8b9489a456ff970f1dea9ac6e04a7f06107
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date: Fri Mar 22 03:56:03 2024 +0200

    Altered dataset sizes for experimentation
18. commit 8418dcf346668eefbbffad36be0f92361d122f24
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date: Fri Mar 22 05:29:30 2024 +0200

    Minor formatting changes, changed how experimentation results were output to textfile.
19. commit 7b4488d7aea6f51a799b8577523a43b419d045dc
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date: Fri Mar 22 06:46:19 2024 +0200

    Fixed error in calculating Insert time complexity for Experimentation.
20. commit 40c75da7d67548f52733c6b33028cb7d77dc7d5e
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date: Fri Mar 22 14:00:06 2024 +0200

    Formatting + comments.
21. commit 84c2bd96a55680f7d1d7989ec91ea575138e6723
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date: Fri Mar 22 14:32:51 2024 +0200

    Added javadoc comments.
22. commit 59f22ece007b07d0ef89eb0f598d79f737a119ee
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date: Fri Mar 22 14:42:15 2024 +0200

    Added more comments + javadoc comments for Experimentation program.
23. commit 8479bac7018ae418926198922a46b60ed0e74ecb
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date: Fri Mar 22 14:45:40 2024 +0200

    Formatting changes.
24. commit c342019df212075b1d8a207114c568f79428ab1b
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date: Fri Mar 22 14:48:39 2024 +0200

    Added Student Number to top of each program.
25. commit 516913596ae8ab05f14fc7da21620722a2cf0381
    Author: ZuhayrHalday <halday.zuhayr@gmail.com>
    Date: Fri Mar 22 15:15:46 2024 +0200

    Added javadocs and Makefile