# CSC2001F 2024 Data Structures Assignment 2

MSHPRI017

This report is based on a program I created to evaluate the performance of the AVL Tree to establish whether it effectively balances nodes and delivers strong performance regardless of the dataset's size.

Prince Mashava
Mshpri017@myuct.ac.za

# Overall OOP design:

The data read in from the GenericsKB textfile was formatted in the following way, (term, sentence, confidence score) all separated by tabs. So, I created a class called, KB which would have these three fields as instance variables and can be used to create an object of type, KB. The datatypes of the term and sentence are String, and the confidence is of type double. The class has get and set methods for its instance variables.

I read the statements in the GenericsKB.txt and created a KB object using each statement. From there, I inserted each KB object into the AVL Tree. Then I read all the queries in the GenericsKB-queries.txt into an array. To check if the queries are in the array, I used a for-loop to iterate through the array and check if each element (a query) is in the tree using the tree's findTerm method, displaying a message that says if the term is found or not.

To verify that the application accurately processes queries containing 10 terms - 5 present in the dataset and 5 terms absent from the dataset. I created an array of the size 10 and read these terms into that array. I used a for-loop to iterate through the array and check if each element (a query) is in the tree using the tree's findTerm method, displaying a message that says if the term is found or not.

## AVL Tree implementation

I used Hussein Suleman's Binary Tree Node, Binary Tree and AVL Tree classes for this assignment. These classes are all generics and I parameterized them to work with my KB class. The AVL Tree is a data structure that is a self-balancing binary search tree, where the heights of the two child subtrees of any node differ by at most one. It contains methods such as balance, rotateLeft, rotateRight which ensure that it is balanced as nodes are inserted. In the AVLTree class, I created a method called, findTerm that compares objects based on their terms and I used that method when searching for statements in the AVL Tree. I also added counters in the insert methods after every comparison as part of the experiment and counters in the findTerm method to count the searches.

# Experimental tests

## Experiment description:

To conduct the experiments, I varied the size (n) of the input dataset, and measure the number of comparison operations performed by the algorithm in the best-case, average-case, and worst-case scenarios for different values of n.
I created a method called instrumentation that takes a size(n) as a parameter. This is size specifies the size of the AVL Tree. Then I read the GenericsKB.txt and created KB objects using each statement and putting those KB objects into an array. From there, I chose n random elements of the array and inserted them into an AVL Tree. The AVL Tree's insert counter was incremented each time an insertion was made.
Secondly, I used the array with the 10 queries to check if queries are in the AVL Tree. The AVL Tree's search counter was incremented each time a search was performed. This

instrumentation method returns the number of insertions and searches made for each AVL Tree of size,n.

Thirdly, I performed multiple tests for each size. So I created an array with 10 different n-values - [10,50,100,500,1000,5000,10000,20000,35000,50000]. For each n-value, I used a for-loop for perform 10 iterations. I used the max and min functions to get the best case and worst case insert and search values. For the insert average case, I summed all the values and divided them by 10. For the search average case, I summed all the values and divided them by 10.

## Trial test values and outputs (Part 1):

- The query file I manually constructed

```
mshpri017@nightmare:~/DS_Assignment_02/data$ cat Query-Test.txt
soft drink
maser
onion ring
alcoholic cirrhosis
nemertean worm
concentration
motor
JGrasp
Tower
Skyscraper
```

- Test that the application can handle queries with both terms in the dataset and terms not in the dataset correctly

```
Query-Test.txt loaded successfully.

soft drink: Soft drinks are beverages. (1.0)
maser: A maser is an amplifier (1.0)
onion ring: Onion rings are finger food. (1.0)
alcoholic cirrhosis: Alcoholic cirrhosis is the destruction of normal liver tissue, leaving non-functioning scar tissue. (0.8427623510360718)
nemertean worm: Nemertean worms are long, thin, animals without segments. (0.8558743000030518)
Term not found: concentration
Term not found: motor
Term not found: JGrasp
Term not found: Tower
Term not found: Skyscraper
```

- Loading the GenericsKB and the GenericsKB-queries files

```
Choose an action from the menu:
1. Load the knowledge bases
2. Search items in the query
3. Test application using query file
4. Experiment
5. Quit

Enter your choice: 1

GenericsKB.txt loaded successfully.
Number of insertions into AVL Tree: 721884
GenericsKB-queries.txt loaded successfully.
```

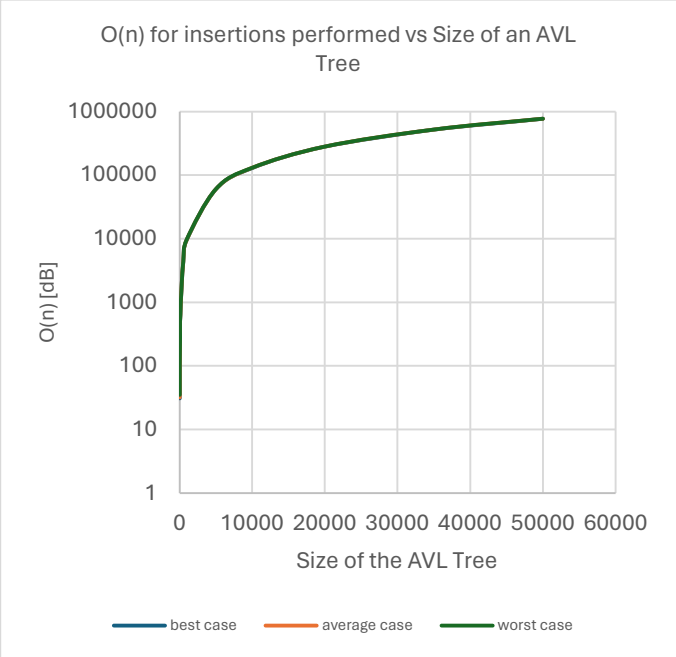
- Last ten searches for terms in the GenericsKB-queries file

```
religious activity: Religious activities are cultural activities. (1.0)
rightist: A rightist is a conservative (1.0)
different gas: Different gases comprise particles with different masses. (0.7551524043083191)
ethnic study: Ethnic studies are fields of study. (1.0)
splayed leg: Splayed legs are legs that extend outward from the hips. (0.8099573254585266)
amphidiploid: Amphidiploids are tetraploids with a complete diploid genome complement from each species parent. (0.8168812990188599)
sponger: A sponger is a follower (1.0)
moon jelly: Moon jellies catch small plankton with tentacles, covered with stinging cells, called nematocysts. (0.8074753880500793)
follicular cast: Follicular casts are tightly adherent scales around the hair shaft. (0.7716030478477478)
riptide: Riptides are current. (1.0)

Number of searches performed: 140498
```

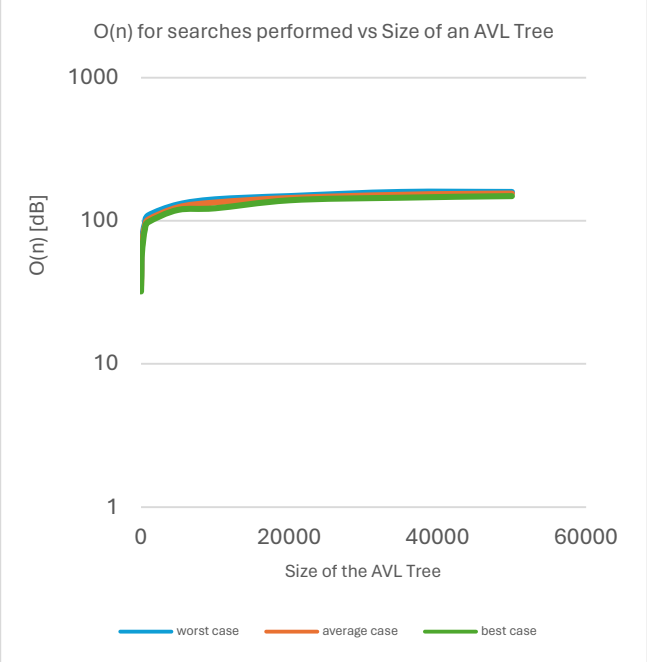
# Results:

- Insertions:




| Insertions: | | | |
|---|---|---|---|
| n | best case | average case | worst case |
| 10 | 31 | 32 | 35 |
| 50 | 265 | 270 | 276 |
| 100 | 632 | 642 | 656 |
| 500 | 4321 | 4358 | 4385 |
| 1000 | 9671 | 9717 | 9775 |
| 5000 | 60270 | 60475 | 60695 |
| 10000 | 130829 | 131065 | 131469 |
| 20000 | 281369 | 282422 | 283320 |
| 35000 | 522492 | 523060 | 524573 |
| 50000 | 772146 | 773636 | 775635 |

- Searches




| Searches: | | | |
|---|---|---|---|
| n | best case | average case | worst case |
| 10 | 32 | 35 | 39 |
| 50 | 51 | 57 | 61 |
| 100 | 60 | 67 | 74 |
| 500 | 87 | 91 | 95 |
| 1000 | 98 | 102 | 109 |
| 5000 | 119 | 124 | 130 |
| 10000 | 122 | 134 | 141 |
| 20000 | 139 | 144 | 149 |
| 35000 | 145 | 152 | 159 |
| 50000 | 149 | 155 | 159 |

## Discussion of results:

From the results, we can see conclude that the best, average and worst case for insertions in an AVL Tree are O(log n). We see that for searches, the best, average and worst case in an AVL Tree are also O(log n). This is due to the self-balancing nature of AVL trees, which ensures that the tree remains approximately balanced at all times.

In the best case scenario, the tree is perfectly balanced, meaning that the difference between the heights of the left and right subtrees of any node is at most 1. In a perfectly balanced AVL Tree, the height of the tree is approximately log n. Therefore, the time complexity for insertions and searches is O(log n) in the best case.

For the average case, the AVL Tree may not be perfectly balanced all the time, the self-balancing property ensures that the tree remains approximately balanced after insertions and deletions. This means that the height of the tree is still proportional to log n. As a result, the time complexity for insertions and searches remains O(log n) on average.

In the worst-case scenario, the AVL Tree may become slightly unbalanced due to a series of insertions or deletions on one side of the tree. However, the AVL property guarantees that the height of the tree never exceeds O(log n). This is because whenever the tree becomes unbalanced, rotations are performed to restore the balance, ensuring that the height remains bounded by O(log n). Therefore, the time complexity for insertions and searches is still O(log n) in the worst case.

In conclusion we see that AVL Trees provide efficient performance for searches and insertions, with a time complexity of O(log n) in the best, average, and worst cases, making them suitable for applications that require guaranteed logarithmic-time performance. Thus, they do effectively balance nodes and deliver strong performance regardless of the dataset's size. The time complexity doesn't degrade as the size of the tree increases.

# Git usage log

```
0: commit b5faa95a41b36ec67605bd1df1c3d6b3453eefe0
1: Author: Prince Mashava <mshpri017@myuct.ac.za>
2: Date: Thu Mar 21 12:24:34 2024 +0000
3:
4: To get the average case, I created a averageCaseInsert and averageCaseSearch as well as a for-loop to call the instrumentation method 10 times and sum th
e results each time the method is called. When the for-loop has terminated, I divided each averageCase variable by 10 to get the average
5:
6: commit 240feefa058b691a84d3fac3cde689af00534fc1
7: Author: Prince Mashava <mshpri017@myuct.ac.za>
8: Date: Tue Mar 19 21:15:01 2024 +0000
9:
...
31: Author: Prince Mashava <mshpri017@myuct.ac.za>
32: Date: Fri Mar 15 15:28:46 2024 +0000
33:
34: I created a feature to load the GenericsKB textfile into an AVLTree. Then load the GenericsKB-queriesinto an array. Then another feature to loop through
 the arry of queries and check if there are any statements whos terms match the queries in the array
35:
36: commit ffa2d1b7bf851f1cd238cd8a758c02905912ba7a
37: Author: Prince Mashava <mshpri017@myuct.ac.za>
38: Date: Fri Mar 15 14:38:34 2024 +0000
39:
40: I used Hussein's AVLTree and modified to work my KB class. Then I created the GenericsKbAVLApp to allow users to load the two knowledge base textfiles.
I added methods in the AVLTree to find terms
```