# CSC2001F

# Assignment 2

## Experiment Description

This assignment is about testing the efficiency of AVL Trees, A CVS file with a list of 9919 entries of country names, dates and vaccination numbers. The general idea was to check if a country has had vaccines done on a particular date and how many vaccinations were completed on that date. The data is stored in an array first, randomized and then store in an AVL Tree after that. The data that is stored in these data structures is read from the provided CSV file and the file is broken down into 3 separate arrays of the country name, the date and the number of vaccinations. Various functions and methods are used to make use of the data provided. A combination of the country and the date is used to find vaccinations numbers of a particular day, that is however not necessarily the focus of this assignment, however this combination will still be utilized to control the data more efficiently.

The aim of this assignment is to check if AVL Trees really do balance the nodes and provided good results regardless of the order of the data. The best way to do this is to randomize the data and test the AVL Tree across different variations of the randomized data. This serves that purpose by randomizing the data using the Random class from Java as a sure way to make the data randomized, not only that but the randomization will be repeated x times to make sure it is consistent across all the x times.

## Randomization Approach & Application Design:

The randomization approach of this assignment revolves around using the Random class to randomize the values in the AVL Tree, across all the maximum of 20 different tests or permutations of the AVL Tree. Basically, the AVL Tree can

be randomized x times, each time the output is displayed and written to a File called RandomizedFile, the RandomizedFile will contain x amount of files, each containing the output for that specific randomization. So, all the x files have different data that has been randomized each time and then sent to these files. In each of these files, below the file is the minimum, average and maximum number of values for that specific permutation of the data.

## Class: AVLExperiment

This assignment only required only one class, the AVLExperiment has all the methods required to populate the AVL Tree and the methods required to randomize the data. It contains the main method, the createFiles method, the

## Methods in AVLExperiment:

1. `printOperations(String place, String day, BinaryTreeNode<String> rootNode, int SampleSize) -` This method is used to search for all the 'country + date' combinations that match with the supplied data (arguments) in the traditional array that they are stored in. The method will then be utilized by the RandomizeData method.

2. `randomizeData(String[] arrayCountry, int subset, String sDate, PrintWriter sampledata)`- This method is used to randomize the data in the AVL Tree once it has been populated. It makes use of the Random class to randomize the data. It then store the randomized data in x different files, depending on your specified x, which will be handled by the createFiles() method. The method also finds the minimum, maximum and average values of the insert and search operations, and outputs these on the files for that specific text file.

3. `reateFiles(String ram_no,String sDate)`– This method is used to create the x different text files that will store the output data for the x different times that the data set is randomized. The number of files created depend on the specified number (x), which is totally up to the user.

4. `main(String[] args)`– This is the main method where everything is basically initialized, this is where the array object and the AVL tree are initialized and then populated. The main method also reads in the CVS file dataset using the File and Scanner classes. The main method also calls the createFiles() method that calls the RandomizeData() method that calls the printOperations() method that outputs the data.

# AVL Tree test values and Outputs:

**This is a sample from the text file randomizedFile.txt from RandomizedData folder**

There are 6 search operations for Georgia ,and 15 insert operations.

There are 6 search operations for Saint Kitts and Nevis ,and 15 insert operations.

There are 5 search operations for Cook Islands, and 15 insert operations.

There are 9 search operations for Timor ,and 15 insert operations.

There are 8 search operations for Chad, and 15 insert operations.

There are 9 search operations for Kenya, and 15 insert operations.

There are 7 search operations for Portugal, and 15 insert operations.

There are 5 search operations for Guatemala ,and 15 insert operations.

There are 8 search operations for Kosovo ,and 15 insert operations.

There are 8 search operations for Asia ,and 15 insert operations.

There are 6 search operations for Isle of Man, and 15 insert operations.

There are 8 search operations for Peru ,and 15 insert operations.

There are 8 search operations for Uzbekistan ,and 15 insert operations.

There are 9 search operations for Uruguay ,and 15 insert operations.

There are 7 search operations for Italy ,and 15 insert operations.

There are 9 search operations for Malawi ,and 15 insert operations.

There are 9 search operations for North Macedonia ,and 15 insert operations.

There are 6 search operations for El Salvador ,and 15 insert operations.

There are 7 search operations for Iraq ,and 15 insert operations.

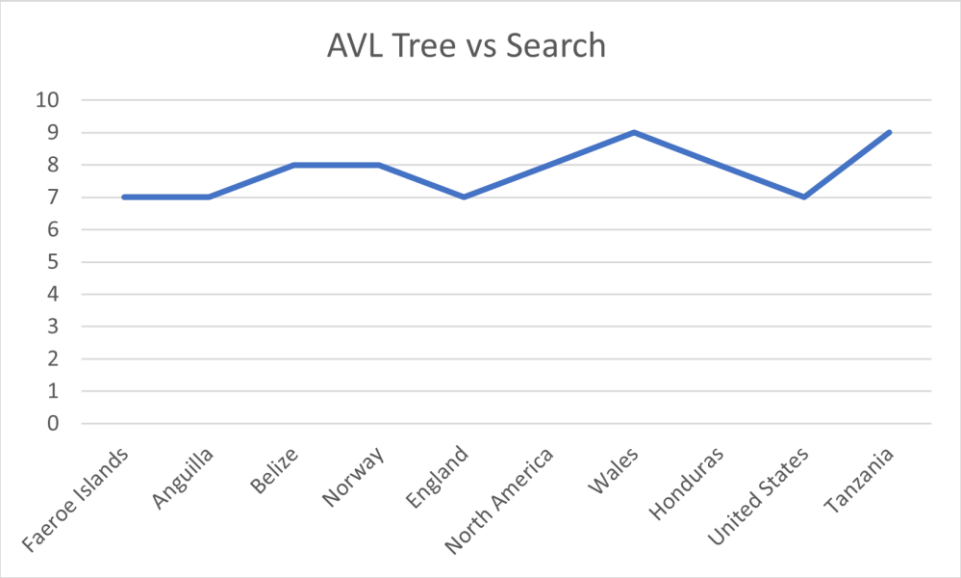
There are 8 search operations for Saudi Arabia ,and 15 insert operations.

There are 2 search operations for Romania ,and 15 insert operations.

# Results and Graphs:

**This is data that is extracted from different sample files but showcasing the same countries**

| Country | Search operations | Insert operations |
|---|---|---|
| Faeroe Islands | 7 | 15 |
| Anguilla | 7 | 15 |
| Belize | 8 | 15 |
| Norway | 8 | 15 |
| England | 7 | 15 |
| North America | 8 | 15 |
| Wales | 9 | 15 |
| Honduras | 1 | 15 |
| United States | 3 | 15 |
| Tanzania | 5 | |



# Discussion of results:

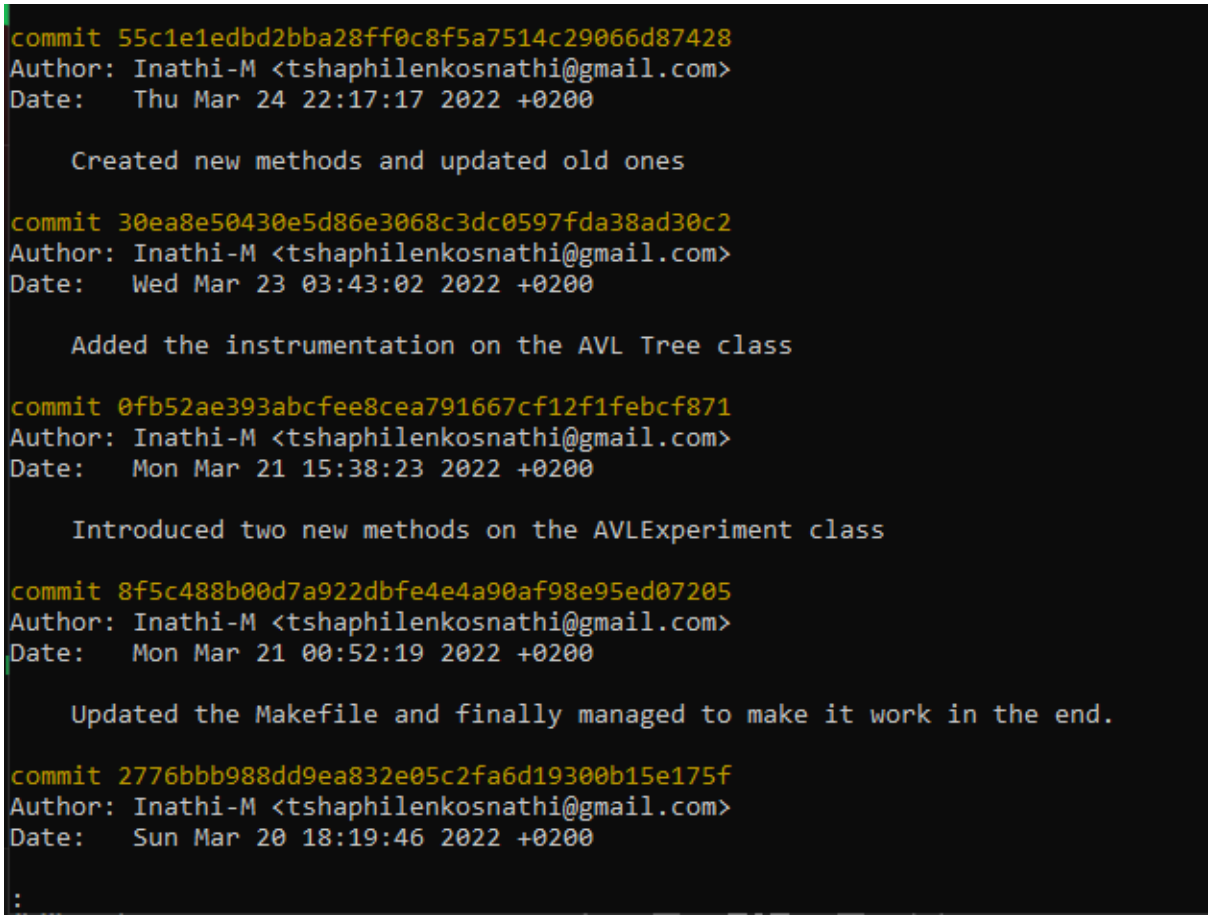
As we can see from the results above, AVL Trees are very consistent in the AVL across the different permutations of the data, regardless of how random the data is. The running time of the AVL Tree insertion is O(log n) and based on the results of the Binary Search Tree in the first assignment, the AVL Tree is much faster that a normal AVL Tree. This is of course since an AVL Tree is a height-balanced tree and therefore, the height of the tree never grows more than N where N is the number of nodes in the tree.

But the downside to AVL Trees is that fact that they are very difficult to implement and the constant factors of the AVL Trees for operations are very high, as you can see from the results above. This applies for both the insert and search operations, but especially for search operations.

.

# Git Usage:

```
0: commit 2e14c8ac64d558f0a183acb96c8cb5502a2d9326
1: Author: Inathi-M <tshaphilenkosnathi@gmail.com>
2: Date: Sat Mar 26 18:53:54 2022 +0200
3:
4: Added my report
5:
6: commit 1bd4236d7deaae62ccd71e9eb7cd618b1eca7876
7: Author: Inathi-M <tshaphilenkosnathi@gmail.com>
8: Date: Sat Mar 26 12:38:55 2022 +0200
9:
...
```




# Conclusion:

The conclusion is therefore that an AVL Tree is indeed very good in terms of performance irrespective of the order of the data that is provided, because this assignment proved AVL tree to be consistent over many iterations of randomized data (20 at most). AVL tree is a self-balancing binary search tree where the balance of the tree is checked by the balance factor and modified whenever required by performing a rotation process. Insertion and Deletion time complexity of AVL tree is O(log n) and the searching time complexity of the AVL tree is O(n) which makes it better than binary search tree and red-black tree. AVL tree is used for faster and efficient searching of data in a large dataset and therefore, it is always recommended to learn and understand the AVL tree in detail.