This experiment aims to determine if AVL trees really do balance nodes and provide good performance irrespective of the order of the data.

The data is randomized in such a way that during each randomization more elements are manipulated to ensure the efficiency of randomizing because many elements of the data have the same country name. To achieve this the project uses several classes, which are **Vaccination.java, VaccineArray.java** and **RandomizeVaccines.java**.  **Vaccination.java** has a constructor which can easily split the lines in the **vaccinations.csv** file into country, date, and number of vaccines. **VaccineArray.java** creates type Vaccination array object and also has an **add ()** method**. RandomizeVaccines.java** creates VaccineArray object and uses the **add ()** method to add objects *(country+date+number of vaccines)* from the **vaccinations.csv** file into the array data structure. Finally, from here the array data structure can easily be manipulated to randomize the data. **RandomizeVaccines.java** has **randomize (Vaccination [] v)** method which takes in the built array data structure as a parameter. The array is sliced into sections starting from the first element. The sliced portion is added at the end of the array. This is controlled by the **randomLevel (amount of randomization)** while it increases the more portions are sliced and added at the end of the array randomizing the array more.This is an efficient algorithm because at each iteration it randomizes multiple elements at once.

After the data is randomized each and every array is stored in the **vaccinations\_random.csv** file. Objects in the file are then inserted in the **AVLTree** using the **insert ()** method,counting the number of insertion-related comparisons for each item added. This is done by incrementing the **insertOp** instancevariable when comparing the node and the data item being inserted.

To implement the **find** operation the project has a random query list **(testinput.txt)** that includes a date and list of countries to search for items in the **AVLTree** data structure. The **userInterface ()** methodimplemented in AVLExperiment class is used to achieve this, it uses the **find ()** methodfrom the **AVLTree** class**.**  While searching for items in the **AVLTree** search operations are calculated for every item searched. This is accomplished by incrementing the **searchOp i**nstance variable when comparisons are made.

To automate the whole process of inserting different permutations of the data (1 to 20) and searching for the query list in the data, the project uses a bash script **runApp.sh**. It has a for loop which changes the amount of randomization and uses input redirection to query each and every randomized array using **testinput.txt** file. It also uses output redirection to save different insert and search operations of different permutations of the data in 20 different files for later processing.

The table and the graph below show the number of comparisons operations in terms of the minimum, average and maximum for different permutations of the data.

**Table

Description automatically generated**

**Chart

Description automatically generated**

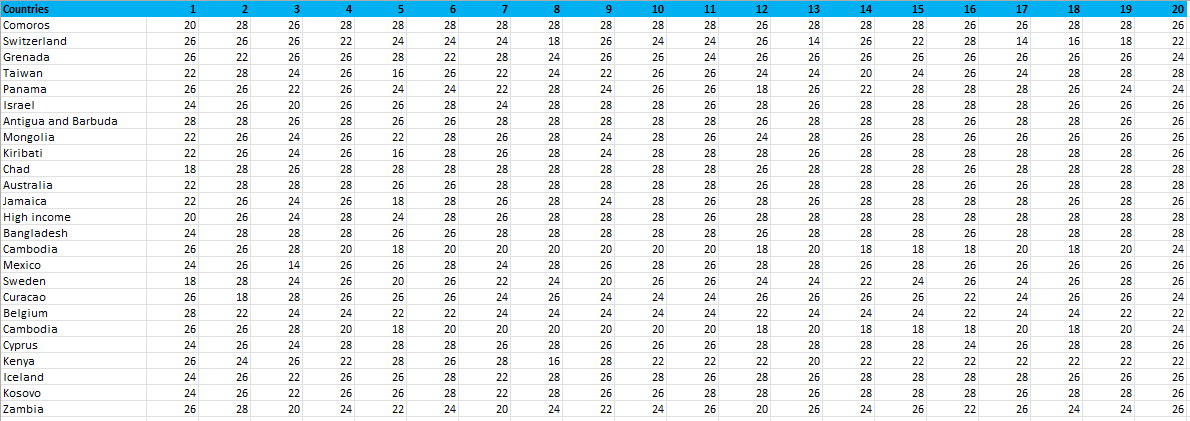
From the above evidence, the best case and worst case for different permutations of the data are the same no matter the order of the data. On the graph, they show a constant time complexity, execution time remains the same no matter the order of the input in the AVLTree. The average case data for different degrees of randomization shows a slight distinction, the average graph is not a straight line because of different insert operations. The average of insert operations increases as the degree of randomization increases.

The table below shows this distinction clearly, it shows the sum of all insert operations of all 9919 items for each and every different degree of randomization. As the amount of randomization(X) increases the number of insert operations also increases. This distinction makes sense because when the amount of randomization increases the data in the vaccinations.csv file is randomized more therefore the AVLTree algorithm will execute more operations to balance the nodes when inserting items in the tree. This ensures that the tree is balanced in the worst case scenario and enables faster search operations.

**Table

Description automatically generated**

The table below shows the search operations for different countries on different permutations of the data. The minimum (best case) search operation is 14, average (average case) is 25.24 and maximum (worst case) is 28. From the data below it is observed that when the amount of randomization increases the search operations for each item do not change by a greater factor, they rather stay the same or decrease. Check countries like *Switzerland* when X=1 the number of search operations was 26 and when X=20 they decreased to 24, *Zambia* when X=1 the number of search operations was 26 and when X=20 they remained the same. Most of the operations repeat themselves for different items. This shows that no matter the order of the data search operations will be same/almost the same for different degrees of randomization. This proves that AVLTree does balance nodes and provide faster search operations.

****

**Git usage:**

**Text

Description automatically generated**