**Thokozani Tshabalala Student number: TSHTHO018**

**CSC2001F**

**ASSIGNMENT 1 REPORT**

**OOP DESIGN STRUCTURE AND CLASSES**

**Vaccine** -The class stores three data values in the forms of strings, it stores the name of the country and date and number of vaccinations. The class has a constructor that takes in one string argument and splits it and stores the data respectively. The class has a compareTo method which compares two Vaccine objects lexographically using the concatenation of the vaccine object’s country and date.

**VaccineArray** – The class has two instance variables; one is the array of vaccine type and the second is an int variable called records. The class has a method called add which adds a vaccine object to the next available element in the array and updates the number of vaccine objects in the array by incrementing the records.

**AVL Tree classes**

**BinaryTreeNode –** The class has three instance variables. The first instance variable is called data which is of data type, it takes the data type of the data that is assigned to it. The second and the third instance variable is the left node and right node respectively. The class has a constructor which takes in three arguments, the first argument is the data of the data type, second argument is the left node and the third argument is the right node. The constructor assigns the argument values to the respective instance variables.

**BinaryTree** – This class has only one instance variable which is the root of the binary tree. The root is of binary tree node type. The class has a no argument constructor which sets the value of the root to null. The class has getHeight methods which returns the height of the tree and getSize methods which return the size of the tree.

**AVLTree-** The AVLTree extends the Binary Tree class, so it a child of the BinaryTree class. The AVLTree class has no instance variables of its own except the ones it has inherited from the BinaryTree class. The AVL tree has method find which searches the AVL tree for specific data. AVLTree class also has a method called insert which inserts data into specific nodes of the tree. The method insert works with the method called balance, it calls the method balance. Balance then will call the fixHeight and height methods to check the height of the tree then fix the height of the tree such that it is balance. The method balance then calls rotateRight or rotateLeft methods respectively to balance the tree if the height of the tree currently causes an imbalance, after this is done the height is fixed.

**AVLExperiment-** This class creates an object of the AVLTree class. It then also creates an instance of the VaccineArray class. Data from a csv file is then read and stored into a VaccineArray object. Then a user is asked for input, this input is the randomisation seed. Once the randomisation seed has been given the data in the VaccineArray object is then randomised according to the randomisation seed. After this data is randomised it is parsed into the AVLTree object.

**AVLExperimentation**- This class randomises data read in from a csv file. It has a method called readIn which is overloaded. The first readIn method reads in the entire file, this first readIn method takes no argument. The second readIn method takes a vaccine object as an argument, it only parses in that vaccine object into the AVL tree.

**Experiment**

**Aim:** To demonstrate the performance of AVL trees given different permutations of input data. This is done by using instrumentation, where we count the number of operations done to insert the data into an AVL tree and the number of operations done to find specific data in an AVL tree. We do this by creating randomisation seeds and then using the seed to randomise the data. This is done by a java script called Randomisation. This script does this randomisation of data before it is parsed into the AVL tree. It creates 30 different permutations of data in the randomisation process then parses each permutation into the AVL tree. For each permutation it will store the number of operations done to store all the data and find each line of data after it has been stored. It then outputs the maximum, minimum ad average for each permutation. We use opCount0i for counting the number of operations done when data is stored, we then use opCount0s to count the number of operations when we search through the data

**Why is the Randomisation script a good algorithm?**

It is a good algorithm because the randomisation seed is generated such that we use a pre- installed java class called Random. The randomisation seed tells us the extent to which we are randomising our data. We use the Random class to get a random integer from 0 to 9919 so as to avoid getting a number bigger than the number of data items we have. We do not parse any value into to Random class except for the upper limit of the seed. Now, for each permutation we create an instance of that Random class. Every time the random class is called a new seed is generated and that is where the argument of it being a good algorithm is. It is that every time we call the random class it will give us a totally different number from the last and probability of getting the same seed twice in a row is very low. The seed is then also used to calculate another random number by calling on the Random class again and getting another random number from it and multiplying this number with the seed, to get a number that is going to be used for randomisation. This number is the used to swap elements in an array from the first element to the element in the position corresponding to the seed value. A for loop is used to calculate this number every time we want to do a swap so meaning we do a double randomisation. This algorithm is good also because of this double randomisation such that you find that in the calculation of the swap number we get a randomised repetition of the same number. This means that we would have 1506 appearing as the swap number 5 times. These repetitions further randomise the data such that the element in the 1506 position in the array is swapped with other elements in different positions 5 times. So this means the double randomisation of this algorithm gives a high probability of moving data from one position to another more than once. This is what makes it good. Now because of the double randomisation data mostly moves thousands position away from it’s last position and considering that the index it will be placed is random and this moving of data is more than once means that the data moves very large and short spaces in the array with every iteration. This further complicates the randomisation and makes it a good algorithm

**Results**

For different permutation of the data the maximum number of operations done to find elements seems to be the same. In the graph above we see that number of operations done by the find method is mostly 16 for the 30 different permutations of the data, there is some data which are not at the 16 line. This data that is not at the 16 line itself is also associated with some stability, meaning some other data points are in the same line as itself. This means for the worst case where we are looking for data that is at the end of the tree, so a leaf node. We get to the leaf node fast in an AVL tree.

In the graph above we see that the minimum number of operations done by find is the same and is the value of 1. This is because the minimum case is where we are looking for the root of the AVL tree. So, one operation will be done to find it.

The average number of operations to find some data in the AVL tree also looks linear with some data being outliers and being found on the line 12. The average number of operations is dominantly at 13. The other operation counts are found at the line equal to 12. So there are only two options in which the number of operations could be it is either 13 or 12. The AVL tree find method is independent of the randomisation or permutation of the data.

In the graph the data exists between three lines 15,16 and 17. We see here that as you increase the extent of randomisation or permutation of the data the insert method operations slightly increase and stabilise. This means that a bit of work is done for insertion as we randomise the data more. This is because the original csv file or data file in sorted in such a way that all the countries vaccinations records are grouped together so as we randomise the data more operations will be done to insert the respective node to the tree.

The graph above depicts the minimum number of insert operations that are done for different permutations of the data. The minimum number of insertions would be just inserting one node to the tree, this parsing or creating only the root node for the AVL tree.

In the graph we see that the number of operations done as we increase the extent at which we randomised the data seems not to change. The operation counts stay at 13 as we randomise the data more, this is because when we insert the

**Conclusion**

for different permutations of the data the find method performance is the same but for insertion it is not the same. The AVL tree performs in the same way for find in different permutations of the data. This means that the order in which data is parsed into the AVL tree does not matter. The AVL tree has been optimised for finding and searching through it. The performance of the AVL tree is the same for different permutations of the data.