Reabetsoe Mpapa MPPREA001
CSC2001F

Experiment Description

An AVL tree is a self-balancing binary search tree in which the difference between the heights of left and right subtrees of any node is less than or equal to one. The height of the AVL tree is always balanced and it gives a better search time complexity when compared to other binary trees due to its self-balancing capabilities.

This case study seeks to determine experimentally if AVL trees really do balance nodes and provide good performance irrespective of the order of the data. To investigate whether AVL trees do balance data and provide good performance irrespective of the order of data, an alphabetically sorted data file with 9919 entries of the number of vaccinations performed in a country on a given date will be randomized to 20 different levels. The 20 different permutations of this data are loaded into an AVL tree as vaccine objects and the number of comparisons performed by the are used to calculate the time complexity. The number of comparisons made are calculated as the vaccine objects (containing country date and number of vaccinations) are inserted into the AVL tree. Operation counts are then also calculated for every search performed for every element in the AVL tree.

Programs used in this Experiment

BinaryTreeNode Class

The BinaryTreeNode class is used to create BinaryTreeNode objects that that contain a pointer to the left and right node/child and some type of data item. These binary node objects are used to store data elements in the BinaryTree nodes.

BinaryTree Class

The Binary tree is implemented in the BinaryTree class. The BinaryTreeNode object is used a root. This class is used to make a binary tree where every node is greater or equal to the values in the left sub-tree and less than or equal to the node values in the right subtree.

AVLTree Class

The AVLTree is implemented in the AVLTree class. It extends the BinaryTree class explained above. It however has additional methods such as BalanceFactor, FixHeight, RotateLeft, and rotate RotateRight.

Vaccine Class

The Vaccine object is implemented in the Vaccine class. A vaccine objects has 4 attributes: country, date, vaccinations, and key which is a concatenation of the country and date. It has two constructors, one that takes in a string line and one that takes in two strings to create vaccine objects.

AVLExperiment Class

The AVLExperiment class reads in a data file and stores it into an array. The AVLExperiment class has a randomization method that takes in a string and an integer (n) and does n loops

Reabetsoe Mpapa MPPREA001
CSC2001F

swapping two random data items in the array at each iteration of the loop. The randomized array is then randomized using the randomization method and the randomized data is then inserted into an AVL tree whilst counting the number of comparison operations and writing them to a file. Each data in the array is also searched in the AVL tree whilst the number of comparison operations are counted and written to a File.

Randomization approach

The randomization approach used in this experiment is swap randomization. The level/amount of randomization measured by the number of swaps done. Each line in the data file is read into an array, and we do loop through the array for X number of times doing a swap between two objects in the array in each iteration of the loop. To determine which two data objects to swap, two random numbers between 0 and 9919 are generated and objects at the array index of these two randomly generated numbers are swapped given that the two randomly generated numbers are not equal. Should the randomly generated numbers be equal no swap would be do, to ensure that swaps are always made when the second randomly generated number is equal to the first randomly generated number another random number is generated until it's not equal to the first randomly generated number. The number of loops done determine how many swaps are done thus it is possible to control how randomized the data in the array is by specifying the number of loops to be made. So, to randomize the file a little bit you'd specify a small number of loops and vice versa.

This randomization algorithm used in this experiment is good because it is not only efficient and simple to implement but it also enables one to easily able to control the amount of randomization and you are guaranteed that the data will be thoroughly random because the index of items to be swapped is randomly generated.

Reabetsoe Mpapa MPPREA001
CSC2001F

Results

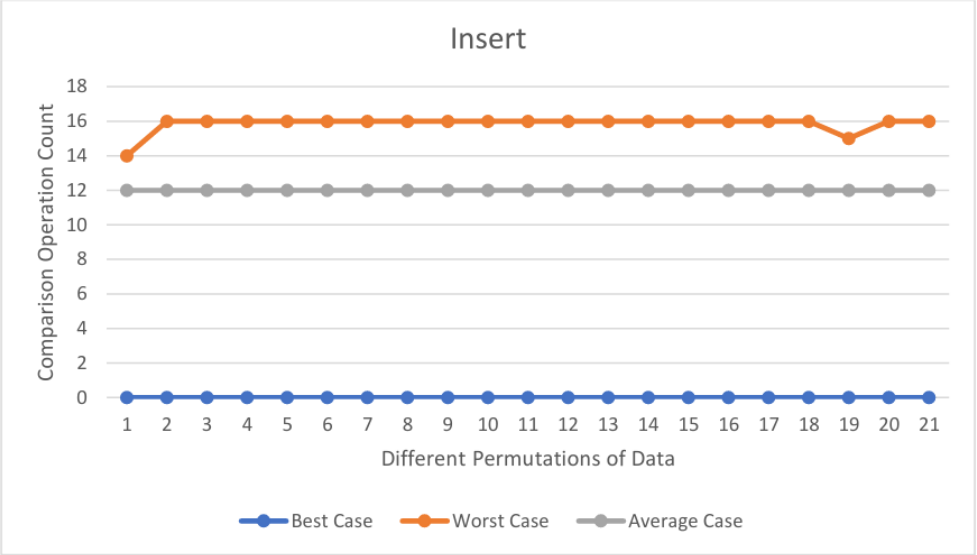

Figure 1

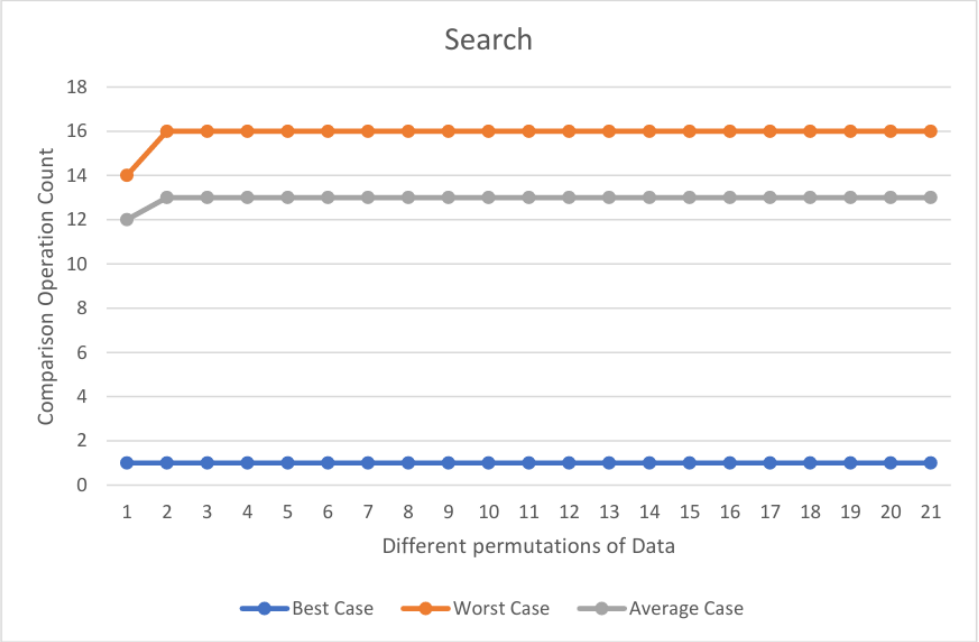

Figure 2

Reabetsoe Mpapa MPPREA001
CSC2001F

Figure 1 and Figure 2 shows the minimum, average and maximum comparison operations made when inserting the data into the AVL tree for 21 permutations of the data. The x-axis shows the different permutations, and the amount of randomisation increasing along the x-axis. This would mean that permutation 1 in the above figures is the least randomized (least number of swaps) and permutation 21 would be the most randomized (most swaps).

The minimum comparison operations made for an insertion is 1, the maximum is 16 whilst the average is 12 irrespective of the degree of randomness in the data.

The minimum comparison operations made for a search is 0, the maximum is 16 whilst the average is 13 irrespective of the degree of randomness in the data as well.

The results in the above table are consistent and support evidence from previous observations on AVL tree time complexity. An AVL tree best time complexity for insertion is 0, meaning that the data is inserted at the root node, the results in figure are consistent with this figure irrespective of how randomized the data is. An AVL tree best time complexity for search is 1, meaning that the data that is searched for is at the root node and only one comparison is made, the results in figure are consistent with this figure irrespective of how randomized the data is.

The worst time complexity of an AVL tree is $O(log_2 n)$ for both search and insert. Both worst cases in the above tables are 16 and the average number of comparisons made are 12 and 13 respectively. We insert 9919 data items into the AVL tree, and the expected worst time complexity should be $log_2 9919 = 13.27$. Though not exactly precise because of the rounding 12 and 13 are close to 13.27 and it can be concluded that the experiment is consistent with the proven time complexities of an AVL Tree.

The most interesting thing about the figures above is the horizontal lines which show that the number of comparisons made remain constant regardless of the permutations of the data set which in this case represent how randomized/ ordered the data is. These graphs prove to use that an AVL tree performance is not affected by how ordered/unordered data is.

This experiment was successful at proving that an AVL trees balance nodes and provide good performance irrespective of the order of the data because the different permutations of the data don't affect the performance of an AVL Tree, as the number of comparison operations made remains constant regardless of how ordered or unordered the data is.