Maria Veltcheva VLTMAR001 18 March 2019

CSC2001F

ASSIGNMENT 2
STATISTICAL ANALYSIS

OVERVIEW

The aim of this report is to analyse the performance of the search and find methods in a Binary Search Tree (BST) and an AVL Tree. Outlined in this report is the design of the applications which implement the AVL and Binary Search Tree data structures, a description of the experiment undertaken on each application, as well as the results obtained during experimentation. Finally, an analysis of the results is included where a comparison between the performances of each data structure is drawn.

APPLICATION DESIGN

In order to analyse the performance of the AVL and Binary Search Tree data structures, a set of java applications were written that each populate either an AVL or a Binary Search Tree with data from a csv file and allow for either a search or a complete traversal to be performed on the dataset in question. An in-depth description on the design of the applications is provided below:

NOTE: The csv file contains records that each list 8 values pertaining to individual household power consumption. The elements that populate the respective data structures are a concatenation of only the Date/Time, Power and Voltage values of the individual records in the csv.

PowerBSTApp

PowerBSTApp, when run, loads data from a specified csv file into a Binary Search Tree and, as mentioned above, either searches for a record in the tree or performs a complete traversal of the tree depending on the command line arguments passed. If no arguments are passed, the printAllDateTimes times method is invoked which prints out the data of every record stored in the tree. If a Date/Time string of the form “DD/MM/YYYY/hh:mm:ss” is passed as the first parameter, the Date/Time string is passed to the printDateTime method where it is searched for in the tree. If the Date/Time string corresponds to a record in the tree then the record’s data is printed and if not, then “Date/Time not found” is printed. Instead of a Date/Time string being passed as the first parameter, the path to a text file storing a set of Date/Time strings can be passed. Thereafter, each Date/Time string in the textfile will be

passed to the printDateTime method one by one and searched for as above. In addition to the argument that instructs the application to search for one or more dateTime strings in the tree, PowerBSTApp also accepts the following as arguments:

args[1] (where args[0] is the dateTime string/queryfile) – A number (N) between 0 and 500 which denotes the number of records that will be loaded into the tree.

args[2] – A String that denotes the path to the file which will store the insertion counter values for a tree of a specific size (explained in detail later on in the report).

args[3] - A String that denotes the path to the file which will store the search counter values for a tree of a specific size (explained in detail later on in the report).

args[4]- A String that ,if present, instructs the program to load data from cleaned_data_sorted.csv as opposed to cleaned_data.csv.

NOTE: Arguments of type string must be placed within inverted commas when running from the command line.

When PowerBSTApp is run from the command line, it checks for the presence of these arguments. If they are present then their values are stored and used as necessary and if not then default values are set and used in the program. The purpose of the arguments is to aid in experimentation and will be explained in detail later on in the report.

The implementation of the AVL Tree data structure is done with the help of two other classes, namely the ***BinarySearchTree*** class and the ***BinaryTreeNode*** class. A brief description of each class and its interactions with PowerAVLApp is given below:

The ***BinaryTreeNode*** class will allow for the instantiation of objects with the following attributes: data in the form of a Date/Time record, as well as links to two BinaryTreeNodes (its left and right children). Along with the aforementioned attributes, the class also contains various get and set methods to allow for the retrieval and manipulation of the node's attributes. The linking of BinaryTreeNodes together constitutes a ***BinarySearchTree.*** The insert method of the BST class is what links objects of type BinaryTreeNode (BTN) together. It does this by traversing the tree and comparing the Date/Time record of the node to be inserted with that of the current node at each step. If a nodes data "comes before" that of the current node then the method traverses left, and if a nodes data "comes after" that of the current node then the method traverses right (lexical ordering is used). It does this traversal until a leaf node is reached, thereafter it creates a new BTN and links it to the leaf node in question as either a left or a right child. The BinarySearchTree is rooted at some starting node, the link to which is stored in the rootNode attribute of the BST class. When the PowerBSTApp application is loading data from the csv into a tree, it first creates an empty tree – which sets the counter attribute of the BST class (explained later) to zero and the rootNode to null. It then invokes the insert method as it traverses the csv, passing the Date/Time record to be stored in the node as an argument. Once the tree has been populated, the main method navigates to either the printDateTime method or the printAllDatesTimes

method according to the command line arguments passed when running PowerBSTApp from the command line, as explained earlier. If the printDateTime method is invoked, the BST class's find method is invoked, in order to search for the given Date/Time record in the tree. The find method traverses the tree in a similar fashion to the insert method, comparing the Date/Time record given with that of each node visited during the traversal. If the Date/Time records match, the method returns the node containing the record, and if the node is not found then null is returned. If the printAllDateTimes method is invoked, the BST class's inOrder method is invoked, in order to traverse the tree and print out the Date/Time, Power and Voltage values for all records in the tree.

PowerAVLApp

The second application, ***PowerAVLApp***, is the one which implements an AVL tree as its data structure. The application has the same methods that perform the same functions as those in PowerBSTApp. It also accepts the same command line arguments as those in PowerBSTApp and its main method navigates between attribute states and method invocations in exactly the same way that the PowerBSTApp application does. The only difference is that an AVL tree is used to store the Date/Time records as opposed to a BST. The implementation of the AVL tree data structure is done with the help of two other classes, namely the ***AVLTree*** class and the ***AVLNode*** class. A brief description of each class and its interactions with PowerAVLApp is given below:

The AVLNode class is almost exactly the same as the BinaryTreeNode class, the only difference is that it stores an additional attribute with each node – an integer to store the height of each node (necessary for maintaining the structure of an AVL tree). AVLNodes link together in a similar fashion as BinaryTreeNodes do to form a tree whereby the linking of AVLnodes constitutes an AVLTree. The insert method of the AVLTree class that links AVLNodes is very similar to the one in the BinarySearchTree class, the only difference is that after each insertion, the balance method is called to ensure that the AVLTree maintains its balance properties. The balance method works in conjunction with the height, balanceFactor, fixHeight, rotateLeft and rotateRight methods which ensure that if at any node in the tree, the height of its left and right subtrees differ by more than one, then the affected nodes are rotated as necessary and their heights updated. The balanceFactor method checks the difference in height between the subtrees, the rotateLeft method performs and anticlockwise rotation on the necessary nodes, whereas the rotateRight method performs a clockwise rotation on the necessary nodes and the fixHeight method is used within the rotation methods to ensure that the adjusted nodes store the correct height value after being shifted. More information on the operation of the methods involved can be found in the javadocs for the AVLTree class.

The classes as described above serve as the basis for the experiment which explores the performance of the AVL and Binary Search Trees' insert and search methods. The design of the experiment is described in the next section of the report.

EXPERIMENT DESCRIPTION

The aim of the experiment is to observe the behaviour of the BST and AVL Trees' insert and search methods as the amount of data loaded into the respective trees changes. This will demonstrate the performance of these data structures in relation to problem size.

In order to make the necessary observations, the insert and search methods of the BinarySearchTree and AVLTree classes are instrumented in such a way that whenever the values of dateTime strings are compared, counter values pertaining to either insertion or searching are incremented. The insertion counter attribute of each class (totalInsertCount) is set to zero when an empty tree is initialised and thereafter, when a node is inserted, the number of comparisons made when inserting the node increments the totalInsertCount attribute. The search counter attribute of each class (findCount) is set to zero whenever the find method is invoked, and as the find method traverses the tree the counter is incremented. The value stored by findCount therefore always refers to the last dateTime record searched for in the tree. When the printDateTime method of either the PowerAVLApp or the PowerBSTApp class is invoked the findCount value pertaining to that search is set and retrieved by invoking the getFindCount method of the class in question. The findCount value is then printed along with the result of the search and written to a text file. The totalInsertCount is printed before the program in question terminates regardless of whether or not printDateTime is invoked. The value is obtained by invoking the getTotalInsertCount method of the class in question (either AVLTree or BinarySearchTree). As is the case with findCount, totalInsertCount is also written to a text file.

The above functionality can be manipulated by passing the appropriate command line arguments to PowerAVLApp and PowerBSTApp so as to observe the behaviour of the insert and search methods in relation to the amount of data stored by the respective datasets. This manipulation constitutes the experiment undertaken. The procedure is as follows:

During each iteration of the experiment, a subset data file of size N is generated from *cleaned_data.csv*. The subset data file is created by running the ***BuildFile*** application, passing N as the first parameter, the path to the text file that will store the subset as the second parameter, and the path to the csv file to create subsets from as the third parameter. The subset file is loaded into an AVL tree and a BST by running PowerAVLApp and PowerBSTApp, each time passing N as the second argument to the respective applications in order to instruct each application to only load N items into the dataset in question. The portion of cleaned_data.csv loaded into each tree is identical to the subset file of size N. When PowerAVLApp and PowerBSTApp are run, the subset file created is passed as the first argument to each application in order to instruct each application to search for the Date/Time record associated with each item in the subset file. The third and fourth arguments passed denote the paths to the text files that will store the insertion and search count values for trees of size N. For a tree of size N, there is only one insertion count value as the tree in question is only built once, but there exist N findCound values for each Date/Time record searched for. If the above procedure is run for N=1 to N=500, a set of 500 text files

containing findCount values as well as one text file containing the insertion count values is produced for each data structure.

Once the text files have been created. Each text file containing findCount values is passed as a parameter to the ***Analysis*** class which traverses the text file and determines the minimum, maximum and average of the values in the file. There exist 3 Analysis files for each data structure – one containing maximum values, one containing minimum values and the other containing average values. Each value is stored alongside the size of dataset from which the value was calculated. When the minimum, maximum and average values are calculated in the Analysis class, the values are appended to the appropriate files along with the value of N corresponding to the file. These files are then graphed so that they can be analysed. The results can be seen in the next section of the report.

Note: A bash script - ***Part5*** **-** automates the process of calling the aforementioned classes with the appropriate arguments from N=1 to N=500**.** The script can be run from the bin directory by either calling it with no parameters or with the parameter "../data/cleaned_data_sorted.csv" which instructs the script to call all of the classes with the respective arguments that denote that cleaned_data_sorted.csv must be used as the sample data file as opposed to *cleaned_data.csv* in all cases where a sample data file is needed. The comments in the script document how this is done. The *cleaned_data_sorted.csv* file is a copy of *cleaned_data.csv* that is stored lexically according to the date/Time field. When the sorted csv file is used, the experiment is only run for N=500. The minimum, maximum and average values are extracted as before.

TRIAL TEST VALUES

PowerAVLApp

Known Dates:

```
16/12/2006/18:18:00, 4.472, 233.29
Search Count: 11
Insertion Count: 3831
```

```
17/12/2006/00:19:00, 2.396, 239.03
Search Count: 13
Insertion Count: 3831
```

```
16/12/2006/19:26:00, 3.62, 233.47
Search Count: 17
Insertion Count: 3831
```

Unknown Date:

```
Date/time not found
Search Count: 18
Insertion Count: 3831
```

Date used:
01/16/2006/18:14:00

No Parameters:

```
16/12/2006/17:24:00  4.216   234.84
16/12/2006/17:25:00  5.36    233.63
16/12/2006/17:26:00  5.374   233.29
16/12/2006/17:27:00  5.388   233.74
16/12/2006/17:28:00  3.666   235.68
16/12/2006/17:29:00  3.52    235.02
16/12/2006/17:30:00  3.702   235.09
16/12/2006/17:31:00  3.7     235.22
16/12/2006/17:32:00  3.668   233.99
16/12/2006/17:33:00  3.662   233.86
```

```
17/12/2006/01:34:00  2.358   241.54
17/12/2006/01:35:00  3.954   239.84
17/12/2006/01:36:00  3.746   240.36
17/12/2006/01:37:00  3.944   239.79
17/12/2006/01:38:00  3.68    239.55
17/12/2006/01:39:00  1.67    242.21
17/12/2006/01:40:00  3.214   241.92
17/12/2006/01:41:00  4.5     240.42
17/12/2006/01:42:00  3.8     241.78
17/12/2006/01:43:00  2.664   243.31
Insertion Count: 3831
```

PowerBSTApp

Known Dates:

```
16/12/2006/19:26:00, 3.62, 233.47
Search Count: 9
Insertion Count: 4546
```

```
17/12/2006/00:19:00, 2.396, 239.03
Search Count: 13
Insertion Count: 4546
```

```
16/12/2006/18:18:00, 4.472, 233.29
Search Count: 9
Insertion Count: 4546
```

Unknown Date:

```
Date/time not found
Search Count: 6
Insertion Count: 4546
```

Date used:
01/16/2006/18:14:00

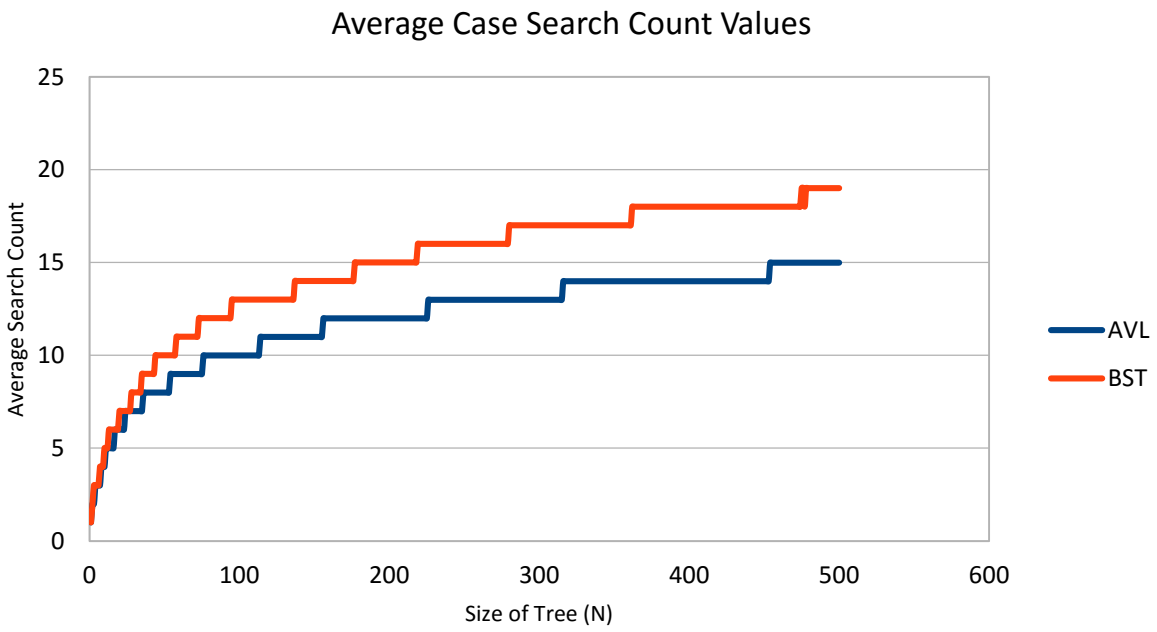
No Parameters:

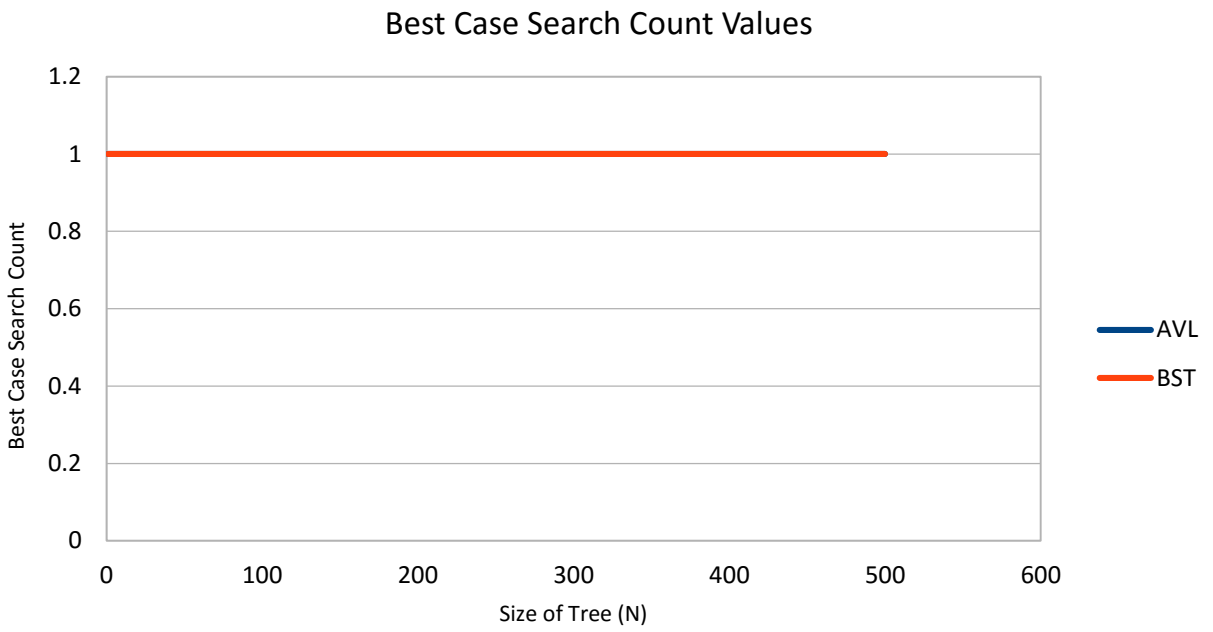
```
vltmar001@sl-dual-170:~/CSC2001F/Ass
Picked up _JAVA_OPTIONS: -Xms512m -X
16/12/2006/17:24:00  4.216   234.84
16/12/2006/17:25:00  5.36    233.63
16/12/2006/17:26:00  5.374   233.29
16/12/2006/17:27:00  5.388   233.74
16/12/2006/17:28:00  3.666   235.68
16/12/2006/17:29:00  3.52    235.02
16/12/2006/17:30:00  3.702   235.09
16/12/2006/17:31:00  3.7     235.22
16/12/2006/17:32:00  3.668   233.99
16/12/2006/17:33:00  3.662   233.86
```

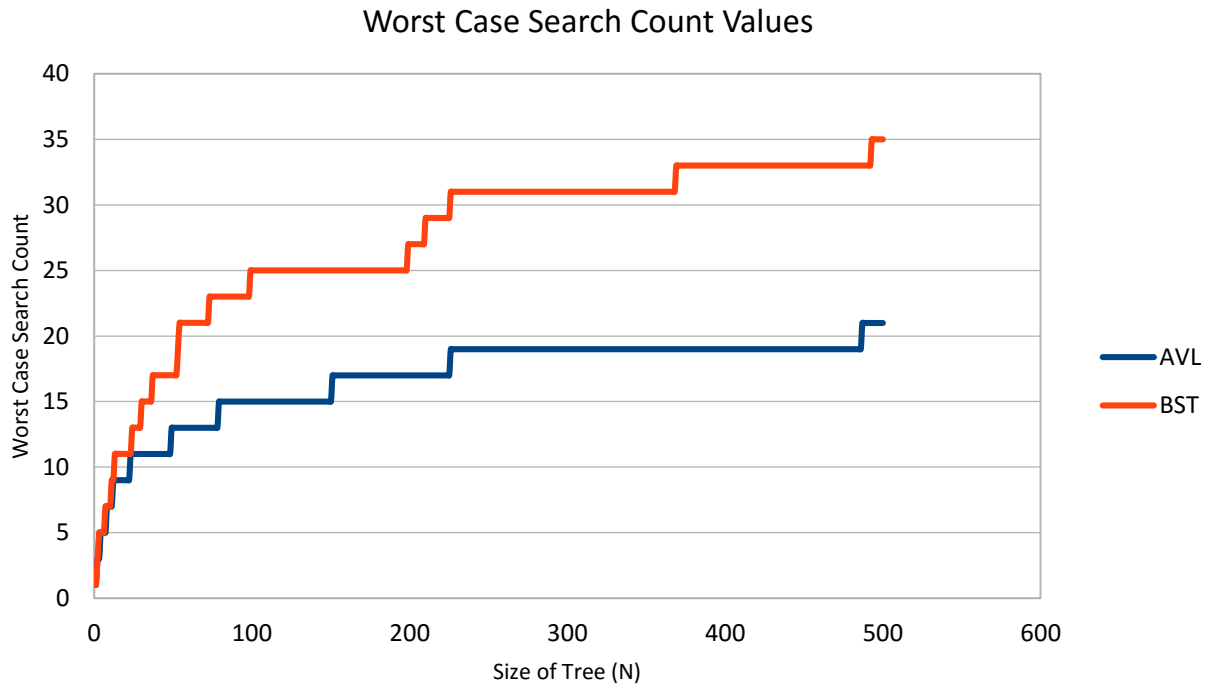
```
17/12/2006/01:34:00  2.358   241.54
17/12/2006/01:35:00  3.954   239.84
17/12/2006/01:36:00  3.746   240.36
17/12/2006/01:37:00  3.944   239.79
17/12/2006/01:38:00  3.68    239.55
17/12/2006/01:39:00  1.67    242.21
17/12/2006/01:40:00  3.214   241.92
17/12/2006/01:41:00  4.5     240.42
17/12/2006/01:42:00  3.8     241.78
17/12/2006/01:43:00  2.664   243.31
Insertion Count: 4546
```

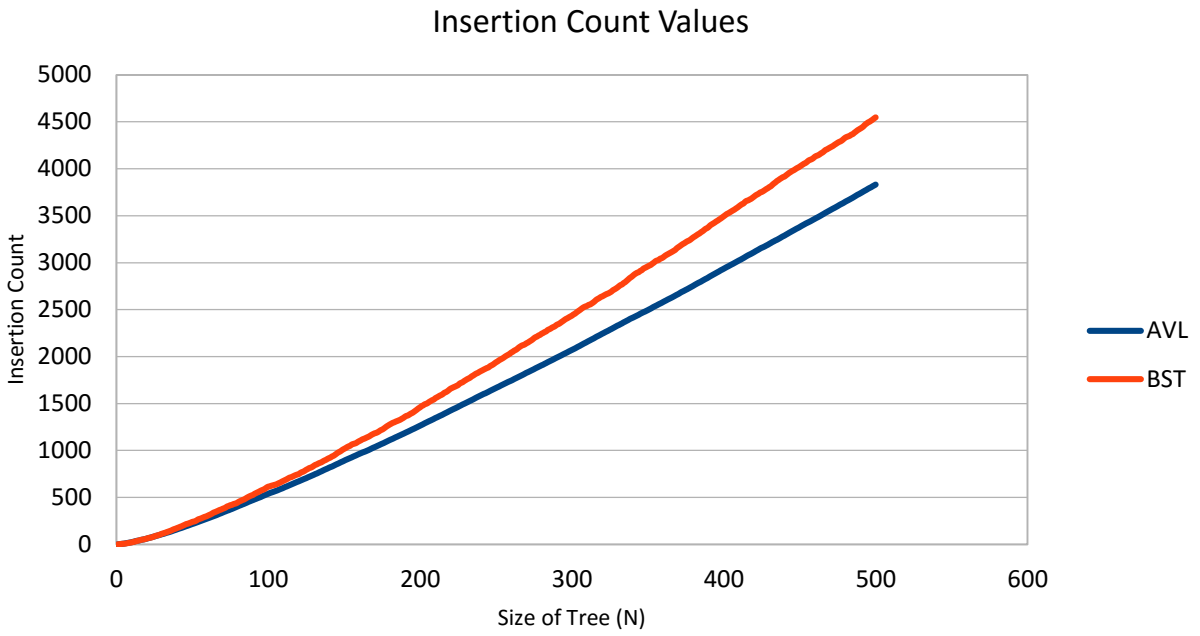
RESULTS OF EXPERIMENTION

Results for the experiment run with the unsorted csv file




Best Case Search Count Values
1.2
1
0.8
0.6
0.4
0.2
0
Best Case Search Count
0
100
200
300
400
500
600
Size of Tree (N)
AVL
BST


Worst Case Search Count Values
40
35
30
25
20
15
10
5
0
Worst Case Search Count
0
100
200
300
400
500
600
Size of Tree (N)
AVL
BST



Results for the experiment run with the sorted csv

SEARCH COUNT VALUES			
	Best Case	Average Case	Worst Case
AVL	1	14	17
BST	1	500	999

INSERTION COUNT VALUES	
AVL	**BST**
3989	124750

DISCUSSION OF RESULTS

Unsorted CSV

From the graphs above, it can be seen that the best case search count for both data structures is a constant value of 1 regardless of the size of the tree. This is expected because the best case is that the value to be found is at the root of the tree in question, and therefore will yield a search count of 1 in any sized tree.

It can also be seen that the graph of average search values for both data structures behaves similarly to the graph of $\log_2 N$ where N is the size of the tree. For example, $\log_2$ (500) is roughly 8.96 whereas the average case search count for the AVL tree is 15 and is 19 for the

Binary Search Tree. They therefore both yield values that are similar to $\log_2 N$ for N=1 to N=500 although they do differ slightly from $\log_2 N$. This behaviour is expected because the nodes to be searched for lie at a depths close to $\log_2 N$ in the tree, assuming the trees are complete, and therefore $\log_2 N$ comparisons will need to be made in order to reach the node to be found. This is due to the fact that a complete tree (with each node having 2 children) will have a height of $\log_2 N$ and since $\log_2 N$ is a very small value in this case, nodes at depths less than $\log_2 N$ will still be very close to $\log_2 N$.

The values yielded by the experiment differ slightly from $\log_2 N$ due to the setup of the experiment. The setup causes this difference because counters are incremented every time a comparison is made, not every time the find method moves to a new node. When at a node the method can make two comparisons – checking if the given Date/Time is equal to the data of the current node and if not equal then checking if the given Date/Time is greater or less than the data of the current node. This means that the values yielded will sometimes be double the value of $\log_2 N$.

The cause of the difference between the AVL tree search count values and the BST search count values is the balance property of the AVL tree which ensures that at any node in the tree, the height of its left and right sub trees differ by at most one. This means that the depth of the AVL tree's leaves are very close to $\log_2 N$. The BST on the other hand has no balance property and will often be structured in such a way that leaves are at depth greater than $\log_2 N$.

A similar pattern can be seen in the graph of the worst case search values where, for each data structure, the data plotted closely resembles $\log_2 N$. In this graph, however, the values are greater than the average case values. This is because the worst case search will visit the leaf node on the longest path from the root node to the leaves. The values yielded are greater than $\log_2 N$ both due to the setup of the experiment mentioned above and due to the randomised nature of the data that is loaded into the trees. Randomised data being loaded into the trees means that the trees will not be complete at each level, especially in the case of the BST, and will result in branches at depths greater than $\log_2 N$ being created. The AVL tree is still closer to $\log_2 N$ than the BST due to its balance property.

In the graph that represents the insertion count for each data structure, it can be seen that the data plotted for each data structure closely resembles $N\log_2 N$. This is expected because inserting a single node into the data structure means that the insert function will have to take $\log_2 N$ steps to get to the leaf nodes where nodes are inserted since $\log_2 N$ is the height of the respective trees. Since N nodes are inserted, the total insertion count is $N\log_2 N$. The insertion count values yielded for the BST are slightly greater than $N\log_2 N$ due to the fact that the tree has no balance properties and will therefore sometimes have a height greater than $\log_2 N$ whereas the balance property of the AVL tree ensures that the height of the tree is $\log_2 N$.

Sorted CSV

In the results pertaining to the experiment run using the sorted csv file, it can be seen that the best case search count is still a constant value of 1 for all values of N as it should be. The average search count value for the AVL Tree is $2\log_2 N$ as before due to the balance property

of the tree. However, the average search count value for the BST is now 500. This is expected because passing sorted data into a BST results in the tree degenerating into a linear linking of nodes. With the tree in this form it would take k nodes being visited to find a node located at position k. Summing the number of comparisons made to locate each node should yield 1 + 2 + 3 + …. + N which is equal to N(N+1)/2 (using the summation rule for summing values 1 to N). Then to obtain the average the value is divided by N which should yield (N+1)/2. The value yielded by the experiment is not (N+1)/2 due to the setup of the experiment mentioned earlier. Due to the fact that the number of comparisons is counted as opposed to the number of nodes visited, the value obtained is 500, which still conforms to what is expected given the setup.
The worst case value observed for the AVL tree is again very close to $\log_2 N$ as before due to its balance properties. For the BST, however, the value yielded is 999. This is because when the tree degenerates into a linear linked list, the worst case is that the node to be searched for is at the very end of the list, and so the number of nodes visited should be N. The value yielded is 2N-1 which is expected given the setup of the experiment because other than when the node to be found is the root node, 2 comparisons are made at each node visited.

From the results it can also be seen that the insertion count value for the AVL tree is similar to Nlog(N) as before. However, the insertion count for the BST is close to N^2 because when the tree degenerates into a linear linking of nodes, it takes k-1 nodes being visited in order to insert into position k. Inserting N nodes, the total number of nodes visited when building the tree should yield 1 + 2 + 3 + … + (N-1) which equals (N-1)*((N-1)+1)/2 (again using the summation rule). Since 500 nodes are inserted, the insertion count should be 124750. The results therefore conform to what is expected.

From the above it can be seen that due to the balancing property of the AVL tree, implementing an AVL tree is far more efficient than implementing a BST. Especially in the case where sorted data is used to populate the respective trees.

CREATIVE COMPONENT

The task of generating the necessary data in order to analyse the performance of the BST and AVL Tree data structures was made as efficient as possible as follows:

The process of running PowerAVLApp and PowerBSTApp using the appropriate command line arguments for N=1 to N=500 would have taken a very long time and been very tedious were it not for the bash script that automates this process. Time is also saved with the use of the BuildFile class which automates the process of creating subset data files of size N=1 to N=500. Lastly the use of the Analysis class to generate the minimum value, maximum value and average of all of the values in the textfiles created during the experiment saves a lot of time and makes the process of calculating the values more efficient. Detailed Information on these classes can be found in the javadocs for this project. Additionaly, information on the bash script, Analysis and BuildFile classes is given in the Experiment Description section of this report.

GIT USAGE

First ten lines:

```
commit 04230ec6074ea995c547d365a5b6a84f0086db24
Author: Maria Veltcheva <VLTMAR001@myuct.ac.za>
Date:   Mon Mar 11 18:55:33 2019 +0200

    A2: AVLNode and AVLTree classes created. The necessary attributes were added to A
VLNode to store the node's data, height, left and right children. The get and set met
hods needed to obtain and manipulate these attributes were also added. rootNode attri
bute added to AVLTree class to store the link to the root node of the tree. The const
ructor and insert methods were also added to the AVLTree class. The former creates th
e tree and the latter allows the tree to be traversed in search of the position of in
sertion and thereafter creates a new node in the position of insertion with the given
 data. Several methods such as balance and fixHeight still need to be added to the AV
LTree class.
```

Last ten lines:

```
commit 3fa85457b75f395d3b965ee39eec1d7ee8af58ab
Author: Maria Veltcheva <VLTMAR001@myuct.ac.za>
Date:   Mon Mar 18 18:19:18 2019 +0200

    A2: Javadoc comments were added to all classes. The way that PowerAVLApp and Powe
rBSTApp write counters to textfiles was changed. They are written with more detail wh
en only one date is passed as a paramater or only a queryfile is passed as a paramate
r. However, if the experiment is being run then the values are written in shorthand s
o as to be interpreted easier by a graphing program. A new attribute numArgs was intr
oduced in order to facilitate this - it stores the value which represents how many co
mmand line arguments were passed to the application when running from the command lin
e -- All tested and working.
```