CSC2001F: Assignment 2
Mahmoodah Jaffer – JFFMAH001
21 March 2019

Contents

Object-Oriented Design

UML for AVL Tree

In Figure 1 below the UML Diagram shows which classes were created for Part 1 and Part 2 of the assignment, and how they interacted with one another:

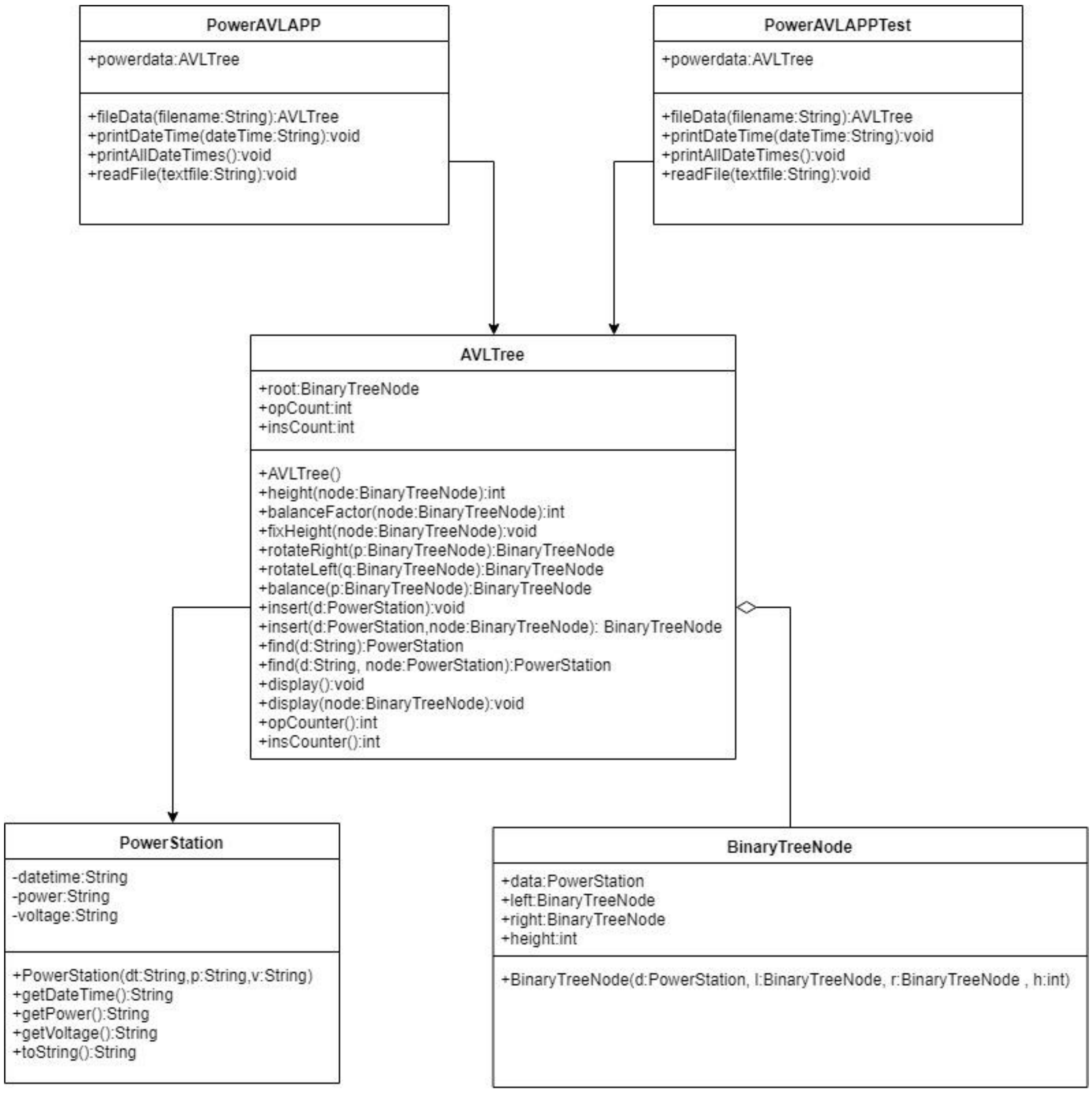

PowerAVLAPP
+powerdata:AVLTree
+fileData(filename:String):AVLTree
+printDateTime(dateTime:String):void
+printAllDateTimes():void
+readFile(textfile:String):void
PowerAVLAPPTest
+powerdata:AVLTree
+fileData(filename:String):AVLTree
+printDateTime(dateTime:String):void
+printAllDateTimes():void
+readFile(textfile:String):void
AVLTree
+root:BinaryTreeNode
+opCount:int
+insCount:int
+AVLTree()
+height(node:BinaryTreeNode):int
+balanceFactor(node:BinaryTreeNode):int
+fixHeight(node:BinaryTreeNode):void
+rotateRight(p:BinaryTreeNode):BinaryTreeNode
+rotateLeft(q:BinaryTreeNode):BinaryTreeNode
+balance(p:BinaryTreeNode):BinaryTreeNode
+insert(d:PowerStation):void
+insert(d:PowerStation,node:BinaryTreeNode): BinaryTreeNode
+find(d:String):PowerStation
+find(d:String, node:PowerStation):PowerStation
+display():void
+display(node:BinaryTreeNode):void
+opCounter():int
+insCounter():int
PowerStation
-datetime:String
-power:String
-voltage:String
+PowerStation(dt:String,p:String,v:String)
+getDateTime():String
+getPower():String
+getVoltage():String
+toString():String
BinaryTreeNode
+data:PowerStation
+left:BinaryTreeNode
+right:BinaryTreeNode
+height:int
+BinaryTreeNode(d:PowerStation, l:BinaryTreeNode, r:BinaryTreeNode , h:int)

Figure 1: UML Diagram for Part 1 and Part 2

The class **PowerStation** was used to store the date/time, voltage and power of the data taken from cleaned_data.csv, in an object called **PowerStation**. The value of each node in the AVLTree was of type **PowerStation**, thereby making it easier to store and retrieve the required data. **BinaryTreeNode** is an inner class of the **AVLTree** class. **BinaryTreeNode** was used to create each node in the AVL tree, the tree of which was created using **AVLTree**. The code for **BinaryTreeNode** and **BinarySearchTree** was obtained from the original authors – Patrick Marais and SJ [1]. I modified their original code to build the required BinarySearchTree. The main method in **PowerAVLApp** calls on the method *printAllDateTimes* if args is empty (meaning that there are no parameters). The method *readFile* is called when the args is a textfile called "DateTimeLists.txt". The *readFile* method will return the datetime, voltage and power of each datetime that is in the textfile. All datetimes are on a new line. If args contains any other string, then the method *printDateTime* is called. A java file **PowerAVLAppTest** was made when doing the testing for Part 5 and Part 6. It is the same as **PowerArrayApp**, besides the fact that it only returns the number of comparison operations count, and the insert comparisons count as opposed to also returning the PowerStation object.

UML for Binary Search Tree

In Figure 2 below, the UML Diagram shows which classes were created for Part 3 and Part 4 of the assignment, and how they interacted with one another:

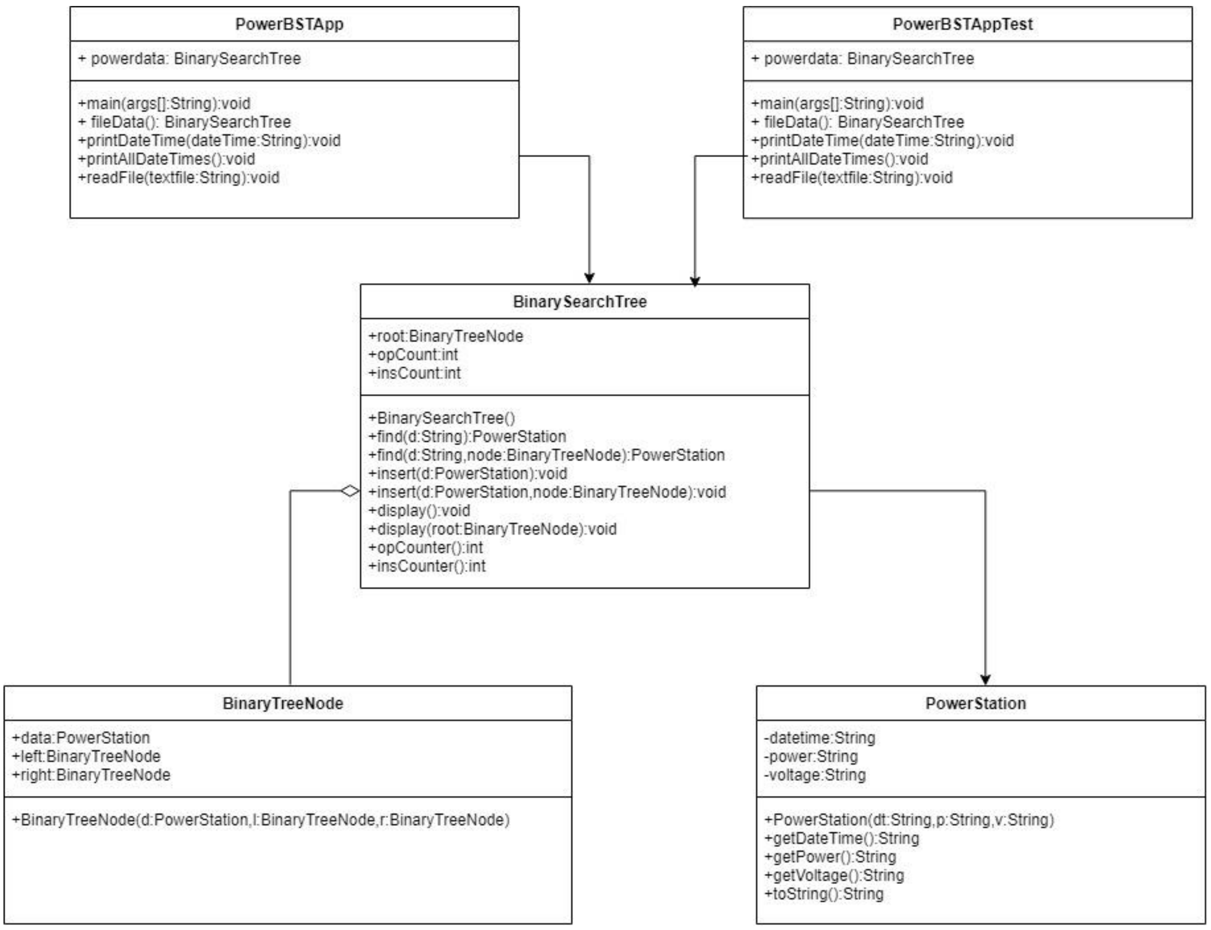

PowerBSTApp
+ powerdata: BinarySearchTree
+main(args[]:String):void
+ fileData(): BinarySearchTree
+printDateTime(dateTime:String):void
+printAllDateTimes():void
+readFile(textfile:String):void
PowerBSTAppTest
+ powerdata: BinarySearchTree
+main(args[]:String):void
+ fileData(): BinarySearchTree
+printDateTime(dateTime:String):void
+printAllDateTimes():void
+readFile(textfile:String):void
BinarySearchTree
+root:BinaryTreeNode
+opCount:int
+insCount:int
+BinarySearchTree()
+find(d:String):PowerStation
+find(d:String,node:BinaryTreeNode):PowerStation
+insert(d:PowerStation):void
+insert(d:PowerStation,node:BinaryTreeNode):void
+display():void
+display(root:BinaryTreeNode):void
+opCounter():int
+insCounter():int
BinaryTreeNode
+data:PowerStation
+left:BinaryTreeNode
+right:BinaryTreeNode
+BinaryTreeNode(d:PowerStation,l:BinaryTreeNode,r:BinaryTreeNode)
PowerStation
-datetime:String
-power:String
-voltage:String
+PowerStation(dt:String,p:String,v:String)
+getDateTime():String
+getPower():String
+getVoltage():String
+toString():String

Figure 2: UML Diagram for Part 3 and Part 4

PowerStation and **BinaryTreeNode** was used in the same way that it was used for **AVLTree**. **BinarySearchTree** was also very similar to **AVLTree** except for the fact that it didn't include any method that helped to balance the tree. The code for **BinaryTreeNode** and **BinarySearchTree** was obtained from the original authors – Patrick Marais and SJ [1]. I modified their original code to build the required **BinarySearchTree**. The main method in **PowerBSTApp** calls on the method *printAllDateTimes* if args is empty (meaning that there are no parameters). The method *readFile* is called when the args is a textfile called "DateTimeLists.txt". The *readFile* method will return the datetime, voltage and power of each datetime that is in the textfile. All datetimes are on a new line. If args contains any other string, then the method *printDateTime* is called. A java file **PowerBSTAppTest** was made when doing the testing for Part 5 and Part 6. It is the same as **PowerBSTApp**, besides the fact that it only returns the number of comparison operations count, and the insert comparisons count as opposed to also returning the PowerStation object.

Goal and Execution of Experiment

The goal of the experiment is to determine whether an AVL Tree, which is a data structure, is more efficient than a Binary Search Tree when trying to find a specific data item in a set of given data. We also researching how the number of insertion comparisons of a Binary Search Tree compares to that of an AVL Tree.

The experiment was executed by creating a java program called **AVLTree** that built a *balanced* Binary Tree. A program called **PowerAVLApp** was created to search for a specified data item in the *balanced* Binary Tree. Another java program was created called **BinarySearchTree** that built a Binary Tree. A program called **PowerBSTApp** was created to search for a specified data item in the Binary Tree. **PowerAVLApp** and **PowerBSTApp** were very similar and for both programs code was added for instrumentation – meaning that the number of comparison operations were counted. Different variables were used for the *find* method and the *insert* method.

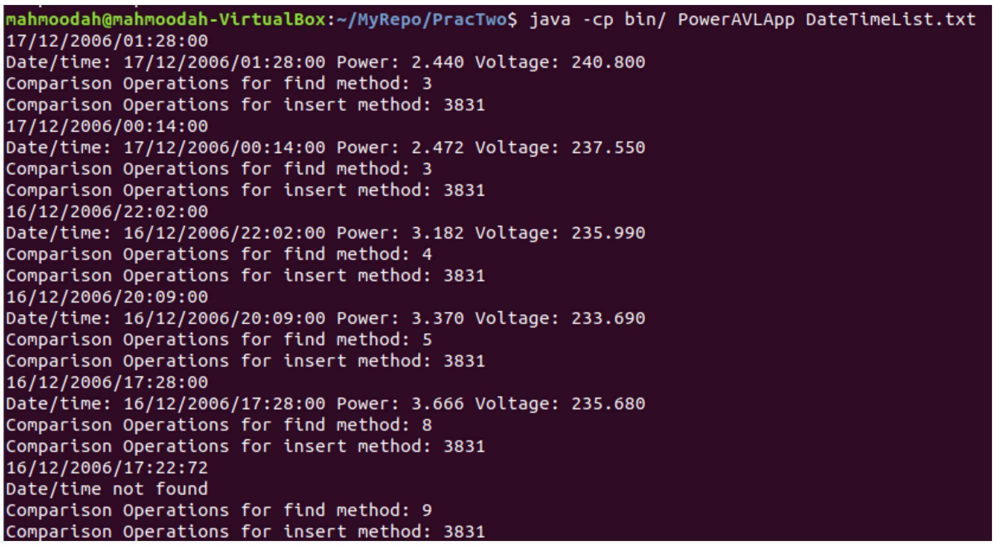
I then used the scripting language Bash to vary the size of the dataset and obtain the number of comparison operations, for each date/time string using both **PowerAVLApp** and **PowerBSTApp** by taking in the file "cleaned_data.csv"**.** The Bash script then output the comparison operations for each date/time to a different CSV file. Afterwards, I used python to determine the best, average and worst case for each date/time string. I used **matplotlib** to plot the graphs for each case so that I could visually see the difference in the number of comparison operations it took for both **PowerAVLApp** and **PowerBSTApp**.

Trial Run for PowerAVLApp

The outputs for the known, unknown and no parameter cases were saved to different files.

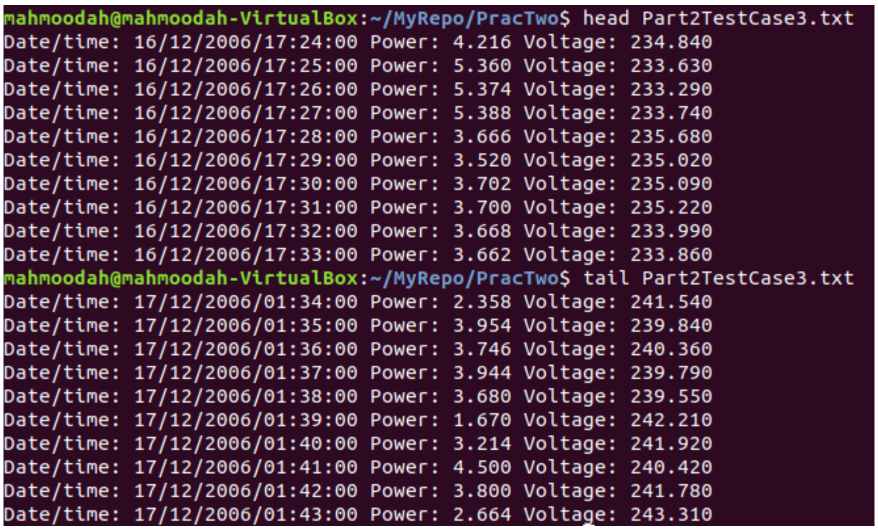
Date/time	**Operation Counts: find**	**Operation: insert**
Case 1: 16/12/2006/23:20:00	1	3831
Case 1: 16/12/2006/18:40:00	**5**	3831
Case 1: 17/12/2006/00:25:00	3	3831
Case 2: Unknown: 16/12/2006/18:05:78	6	3831
Case 3: No parameters	500	3831

Table 1: Trial Run for PowerAVLApp



```
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracTwo$ java -cp bin/ PowerAVLApp DateTimeList.txt
17/12/2006/01:28:00
Date/time: 17/12/2006/01:28:00 Power: 2.440 Voltage: 240.800
Comparison Operations for find method: 3
Comparison Operations for insert method: 3831
17/12/2006/00:14:00
Date/time: 17/12/2006/00:14:00 Power: 2.472 Voltage: 237.550
Comparison Operations for find method: 3
Comparison Operations for insert method: 3831
16/12/2006/22:02:00
Date/time: 16/12/2006/22:02:00 Power: 3.182 Voltage: 235.990
Comparison Operations for find method: 4
Comparison Operations for insert method: 3831
16/12/2006/20:09:00
Date/time: 16/12/2006/20:09:00 Power: 3.370 Voltage: 233.690
Comparison Operations for find method: 5
Comparison Operations for insert method: 3831
16/12/2006/17:28:00
Date/time: 16/12/2006/17:28:00 Power: 3.666 Voltage: 235.680
Comparison Operations for find method: 8
Comparison Operations for insert method: 3831
16/12/2006/17:22:72
Date/time not found
Comparison Operations for find method: 9
Comparison Operations for insert method: 3831
```

Figure 3: Output of a text file when list of date times is given for AVL Tree

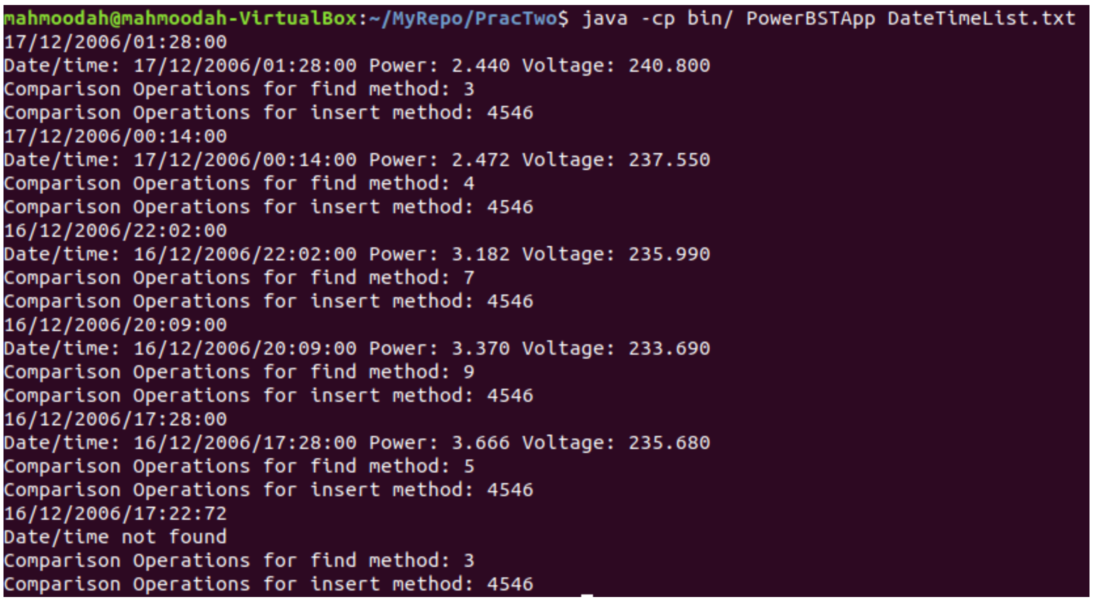

```
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracTwo$ head Part2TestCase3.txt
Date/time: 16/12/2006/17:24:00 Power: 4.216 Voltage: 234.840
Date/time: 16/12/2006/17:25:00 Power: 5.360 Voltage: 233.630
Date/time: 16/12/2006/17:26:00 Power: 5.374 Voltage: 233.290
Date/time: 16/12/2006/17:27:00 Power: 5.388 Voltage: 233.740
Date/time: 16/12/2006/17:28:00 Power: 3.666 Voltage: 235.680
Date/time: 16/12/2006/17:29:00 Power: 3.520 Voltage: 235.020
Date/time: 16/12/2006/17:30:00 Power: 3.702 Voltage: 235.090
Date/time: 16/12/2006/17:31:00 Power: 3.700 Voltage: 235.220
Date/time: 16/12/2006/17:32:00 Power: 3.668 Voltage: 233.990
Date/time: 16/12/2006/17:33:00 Power: 3.662 Voltage: 233.860
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracTwo$ tail Part2TestCase3.txt
Date/time: 17/12/2006/01:34:00 Power: 2.358 Voltage: 241.540
Date/time: 17/12/2006/01:35:00 Power: 3.954 Voltage: 239.840
Date/time: 17/12/2006/01:36:00 Power: 3.746 Voltage: 240.360
Date/time: 17/12/2006/01:37:00 Power: 3.944 Voltage: 239.790
Date/time: 17/12/2006/01:38:00 Power: 3.680 Voltage: 239.550
Date/time: 17/12/2006/01:39:00 Power: 1.670 Voltage: 242.210
Date/time: 17/12/2006/01:40:00 Power: 3.214 Voltage: 241.920
Date/time: 17/12/2006/01:41:00 Power: 4.500 Voltage: 240.420
Date/time: 17/12/2006/01:42:00 Power: 3.800 Voltage: 241.780
Date/time: 17/12/2006/01:43:00 Power: 2.664 Voltage: 243.310
```
Figure 4: First and last ten lines when invoking **printAllDateTimes()**

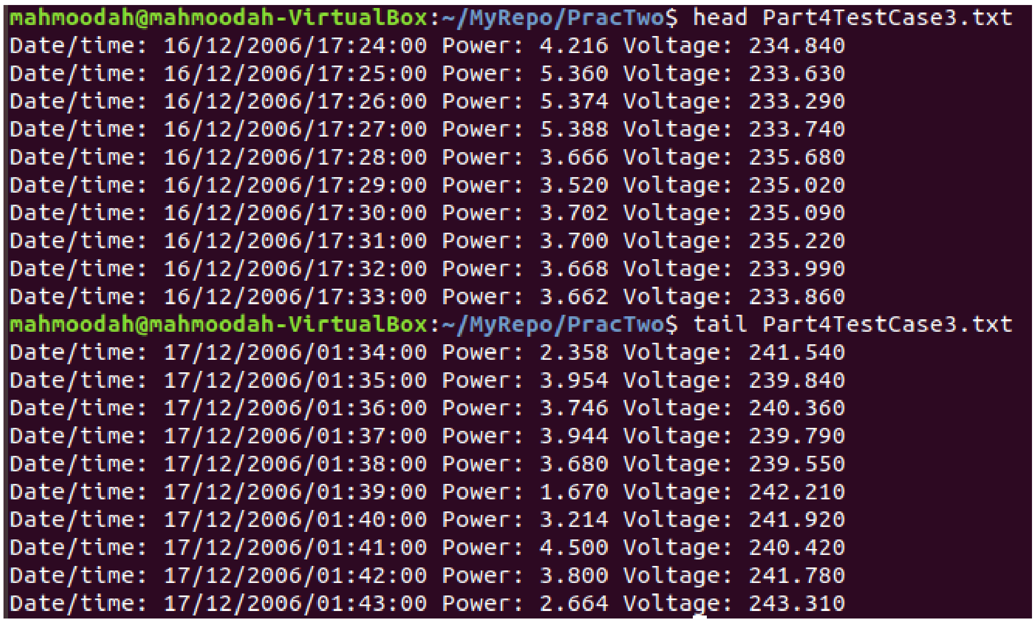
Trial Run for PowerBSTApp

The outputs for the known, unknown and no parameter cases were saved to different files.

Date/time	**Operation Counts: find**	**Operation Count: insert**
Case 1: 16/12/2006/23:20:00	1	4546
Case 1: 16/12/2006/18:40:00	8	4546
Case 1: 17/12/2006/00:25:00	4	4546
Case 2: Unknown: 16/12/2006/18:05:78	8	4546
Case 3: No parameters	500	4546

Table 2: Trial Run for PowerBSTApp


```
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracTwo$ java -cp bin/ PowerBSTApp DateTimeList.txt
17/12/2006/01:28:00
Date/time: 17/12/2006/01:28:00 Power: 2.440 Voltage: 240.800
Comparison Operations for find method: 3
Comparison Operations for insert method: 4546
17/12/2006/00:14:00
Date/time: 17/12/2006/00:14:00 Power: 2.472 Voltage: 237.550
Comparison Operations for find method: 4
Comparison Operations for insert method: 4546
16/12/2006/22:02:00
Date/time: 16/12/2006/22:02:00 Power: 3.182 Voltage: 235.990
Comparison Operations for find method: 7
Comparison Operations for insert method: 4546
16/12/2006/20:09:00
Date/time: 16/12/2006/20:09:00 Power: 3.370 Voltage: 233.690
Comparison Operations for find method: 9
Comparison Operations for insert method: 4546
16/12/2006/17:28:00
Date/time: 16/12/2006/17:28:00 Power: 3.666 Voltage: 235.680
Comparison Operations for find method: 5
Comparison Operations for insert method: 4546
16/12/2006/17:22:72
Date/time not found
Comparison Operations for find method: 3
Comparison Operations for insert method: 4546
```
Figure 5: Output of a text file when list of date times is given for Binary Tree



```
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracTwo$ head Part4TestCase3.txt
Date/time: 16/12/2006/17:24:00 Power: 4.216 Voltage: 234.840
Date/time: 16/12/2006/17:25:00 Power: 5.360 Voltage: 233.630
Date/time: 16/12/2006/17:26:00 Power: 5.374 Voltage: 233.290
Date/time: 16/12/2006/17:27:00 Power: 5.388 Voltage: 233.740
Date/time: 16/12/2006/17:28:00 Power: 3.666 Voltage: 235.680
Date/time: 16/12/2006/17:29:00 Power: 3.520 Voltage: 235.020
Date/time: 16/12/2006/17:30:00 Power: 3.702 Voltage: 235.090
Date/time: 16/12/2006/17:31:00 Power: 3.700 Voltage: 235.220
Date/time: 16/12/2006/17:32:00 Power: 3.668 Voltage: 233.990
Date/time: 16/12/2006/17:33:00 Power: 3.662 Voltage: 233.860
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracTwo$ tail Part4TestCase3.txt
Date/time: 17/12/2006/01:34:00 Power: 2.358 Voltage: 241.540
Date/time: 17/12/2006/01:35:00 Power: 3.954 Voltage: 239.840
Date/time: 17/12/2006/01:36:00 Power: 3.746 Voltage: 240.360
Date/time: 17/12/2006/01:37:00 Power: 3.944 Voltage: 239.790
Date/time: 17/12/2006/01:38:00 Power: 3.680 Voltage: 239.550
Date/time: 17/12/2006/01:39:00 Power: 1.670 Voltage: 242.210
Date/time: 17/12/2006/01:40:00 Power: 3.214 Voltage: 241.920
Date/time: 17/12/2006/01:41:00 Power: 4.500 Voltage: 240.420
Date/time: 17/12/2006/01:42:00 Power: 3.800 Voltage: 241.780
Date/time: 17/12/2006/01:43:00 Power: 2.664 Voltage: 243.310
```

Figure 6: First and last ten lines when invoking **printAllDateTimes()**

Results for Part 5

AVL

AVL – Best Case

The big O notation for the best case of an AVL Tree is O (1) which can be seen from the graph shown in Figure 7 below. This is because the best case for an AVL Tree would be when the specified data element is the first node in the AVL Tree. This is the reason why a horizontal line of the value 1 is displayed on the graph.

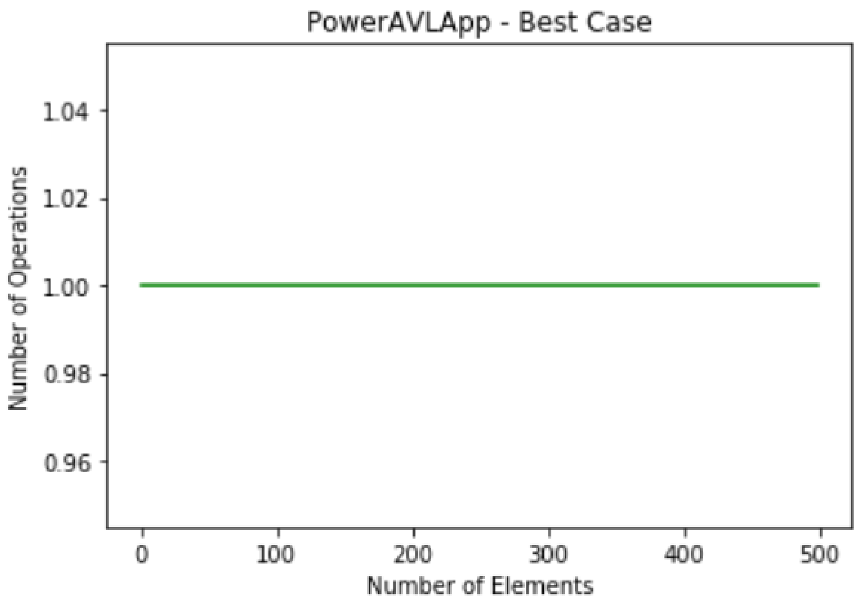


Figure 7: Best Case for AVL Tree

AVL – Average Case

The big O notation for the average case of an AVL Tree is O(log n). As can be seen from the graph in Figure 8 the graph almost resembles that of a log n graph. The graph shows us that while the number of data elements in the AVL increases the number of operations it takes to find a specific element remains relatively small.

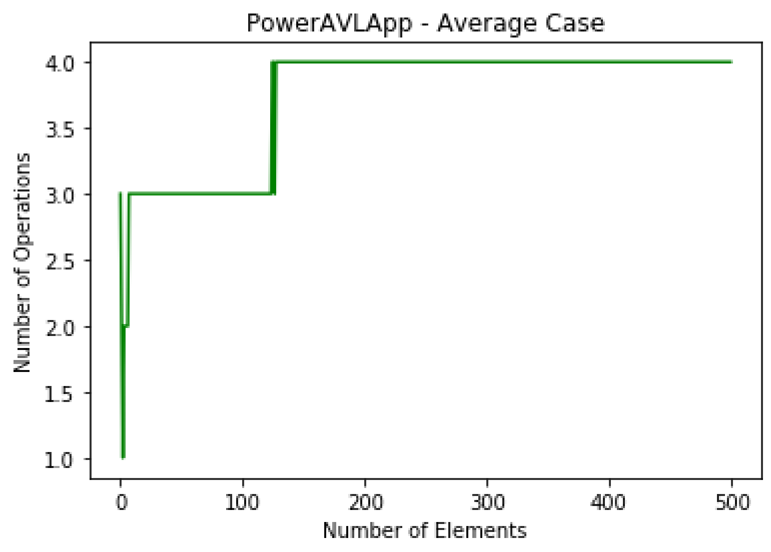


Figure 8: Average Case for AVL Tree

AVL – Worst Case

The big O notation for the worst case of an AVL Tree is O (log n). This can be seen in Figure 9 where the graph looks almost like a log n graph. While it is not an exact replica, we can still see that while the number of elements increase the number of operations remains quite small. The number of operations is double the number of operations required to find an element for the Average Case of an AVL Tree.

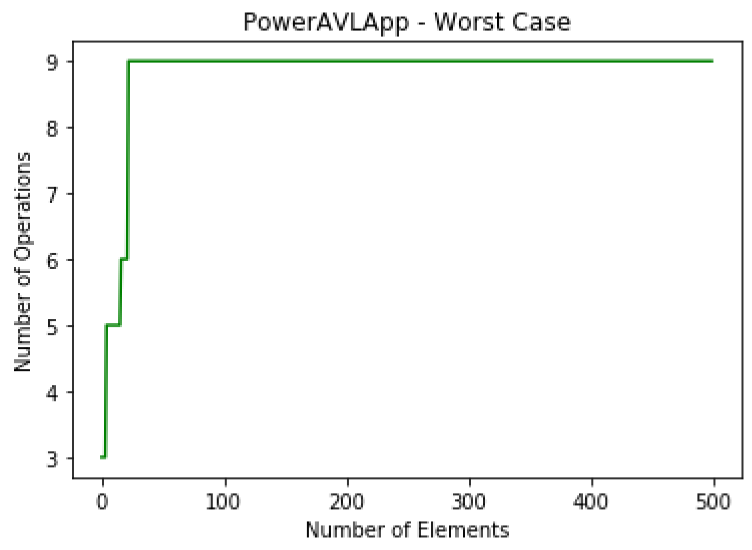


Figure 9: Worst Case for AVL Tree

BST

Binary Search Tree – Best Case

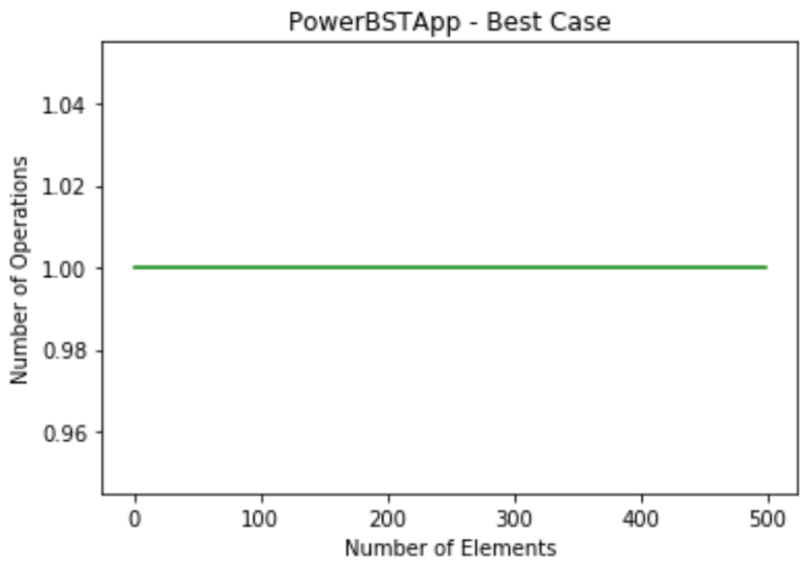


Figure 10: Best Case for Binary Search Tree

The big O notation for the best case of a Binary Search Tree is O (1) which can be seen in the graph above. The relationship of the graph shown in Figure 10 indicates that the best case of the Binary Search Tree always occurs when the data item that is being searched for is in the first node of the Binary Search Tree.

Binary Search Tree – Average Case

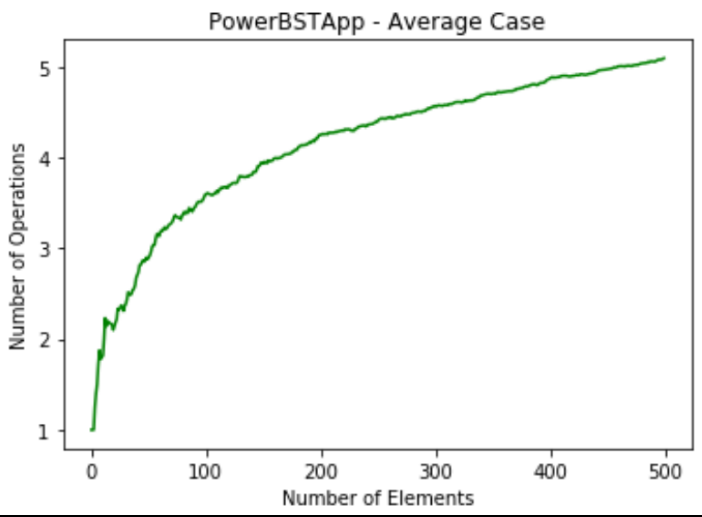


Figure 11: Average Case for Binary Search Tree

The big O notation for the average case of a Binary Search Tree is O(log n). The above graph in Figure 11 almost replicates the exact form of a log(n) graph. The graph shows us that as the number of data elements increases the number of operations does not increase linearly with it like it does for an array – thereby making a Binary Search Tree more efficient than an array.

Binary Search Tree – Worst Case

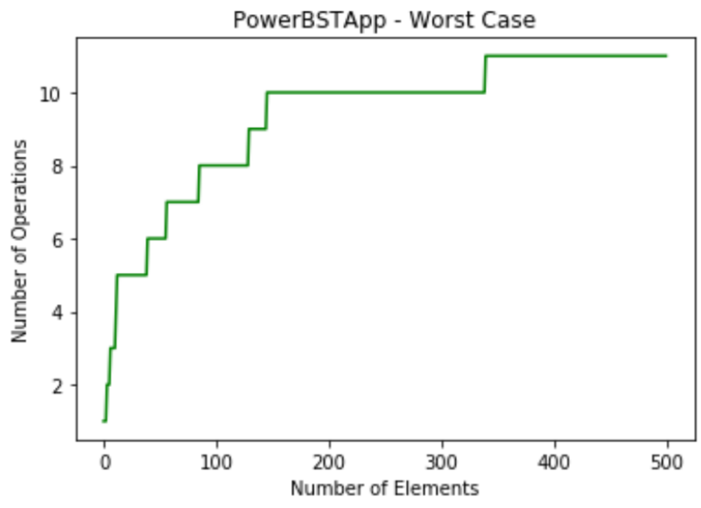


Figure 12: Worst Case for Binary Search Tree

The big O notation for the worst case of a Binary Search Tree is O(log n), which is the same as the big O notation for the average case of a Binary Search Tree. While the graph shown in Figure 12 is not as smooth as the graph in Figure 11 for the average case the shape of the graph still replicates that of a log(n) graph. The graph shows us that as the number of data elements increases the number of comparion operations remains relatively small.

Results for Part 6

The data was sorted in descending order.

Date/time	**Search Operation Count**
Start of data: 17/12/2006/01:34:00	1
Middle of data: 16/12/2006/21:35:00	6
End of data: 16/12/2006/17:24:00	9

Table 3: Results for sorted data in an AVL Tree

AVL Tree: Best case - 1, Average case – 4, Worst case - 9.

Date/time	**Search Operation Count**
Start of data: 17/12/2006/01:34:00	1
Middle of data: 16/12/2006/21:35:00	249
End of data: 16/12/2006/17:24:00	500

Table 4: Results for sorted data in a Binary Search Tree

Binary Search Tree: Best case - 1, Average case – 250, Worst case - 500.

Overall Results

Part 5

From the results, we can see that the insertion operation count of the AVL Tree is lower than that of the Binary Search Tree, with a difference of 715. The reason for this difference is that an AVL Tree is balanced whereas a Binary Search Tree is not.

The search operation count does not differ as much as can be seen from the trial runs in Table 1 and Table 2, when the same date times were searched for using each application.

When looking at the insertion operation of both trees, an AVL Tree is slightly more efficient. When looking at the search operation of trees we can see that it they return almost the same results. This is because both the Binary Search Tree and an AVL Tree have a complexity analysis of O (log n). In this case, there isn't much difference between the search operations of a Binary Search Tree and an AVL Tree but if we were to have a much larger set of data, I do feel that it would be better to use an AVL Tree as opposed to a Binary Search Tree.

Part 6

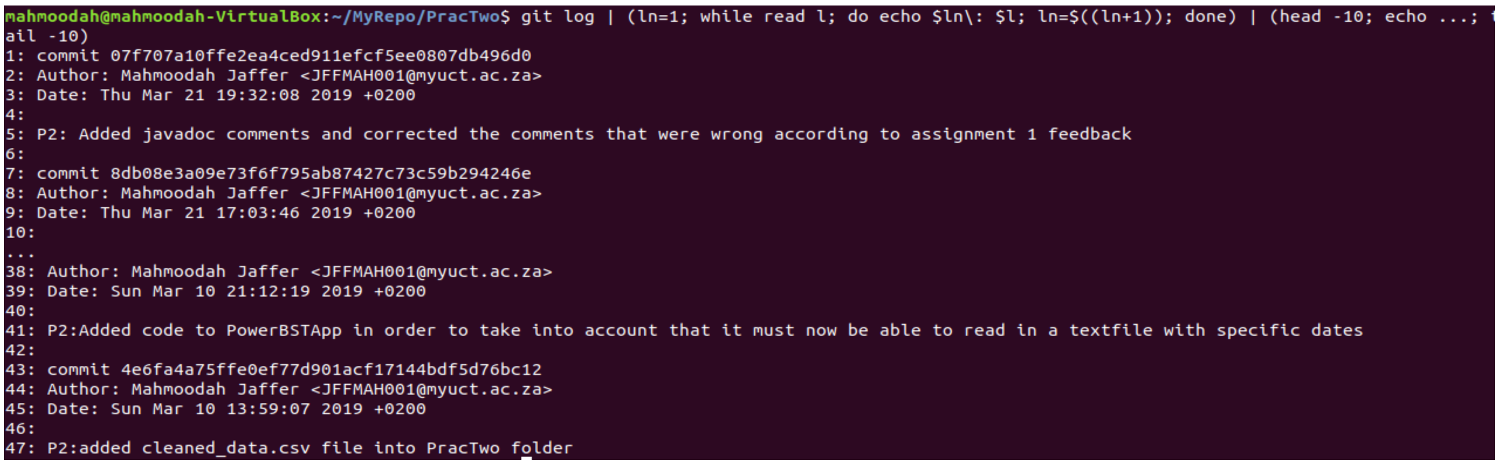
The benefit of using a Binary Search Tree is that it stores the data in 2 different branches thereby decreasing the number of elements it needs to look at by half if you compare it to a linked list. From Table 4 we can see that when a *sorted* set of data is used for a Binary Search Tree, the Binary Search Tree effectively becomes a linked list thereby removing the benefits of using a Binary Search Tree as the complexity analysis for the search operation of a linked list is O(n) which is very inefficient when compared to that of a Binary Search Tree.

However, when a set of sorted data is used for an AVL Tree we can see from the results in Table 4 that the time complexity for the search operation is still O (log n). This is because every time a new element is inserted into the AVL Tree the tree is assessed to see whether it needs to be rebalanced an if it does need to be rebalanced then it is. The balancing of an AVL Tree is what prevents it from becoming a linked list like a Binary Search Tree does when sorted data is used.

Creativity

By creating an object PowerStation to store the date/time, power and voltage it was easier to obtain the required information. I used Bash to generate the subsets of data instead of using an array in java to iterate through the given data. I used pythons' matplotlib to obtain the best, average and worst case as well as using it to generate the graphs, as opposed to using Excel.

Git Usage



```
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracTwo$ git log | (ln=1; while read l; do echo $ln\: $l; ln=$((ln+1)); done) | (head -10; echo ...; 
ail -10)
1: commit 07f707a10ffe2ea4ced911efcf5ee0807db496d0
2: Author: Mahmoodah Jaffer <JFFMAH001@myuct.ac.za>
3: Date: Thu Mar 21 19:32:08 2019 +0200
4:
5: P2: Added javadoc comments and corrected the comments that were wrong according to assignment 1 feedback
6:
7: commit 8db08e3a09e73f6f795ab87427c73c59b294246e
8: Author: Mahmoodah Jaffer <JFFMAH001@myuct.ac.za>
9: Date: Thu Mar 21 17:03:46 2019 +0200
10:
...
38: Author: Mahmoodah Jaffer <JFFMAH001@myuct.ac.za>
39: Date: Sun Mar 10 21:12:19 2019 +0200
40:
41: P2:Added code to PowerBSTApp in order to take into account that it must now be able to read in a textfile with specific dates
42:
43: commit 4e6fa4a75ffe0ef77d901acf17144bdf5d76bc12
44: Author: Mahmoodah Jaffer <JFFMAH001@myuct.ac.za>
45: Date: Sun Mar 10 13:59:07 2019 +0200
46:
47: P2:added cleaned_data.csv file into PracTwo folder
```

Figure 13: Summary statistics for git log usage

References

[1] https://algorithms.tutorialhorizon.com/binary-search-tree-complete-implementation/