CSC2001F: Assignment 1
Mahmoodah Jaffer – JFFMAH001
6 March 2019

Contents

Object-Oriented Design

UML for Array

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

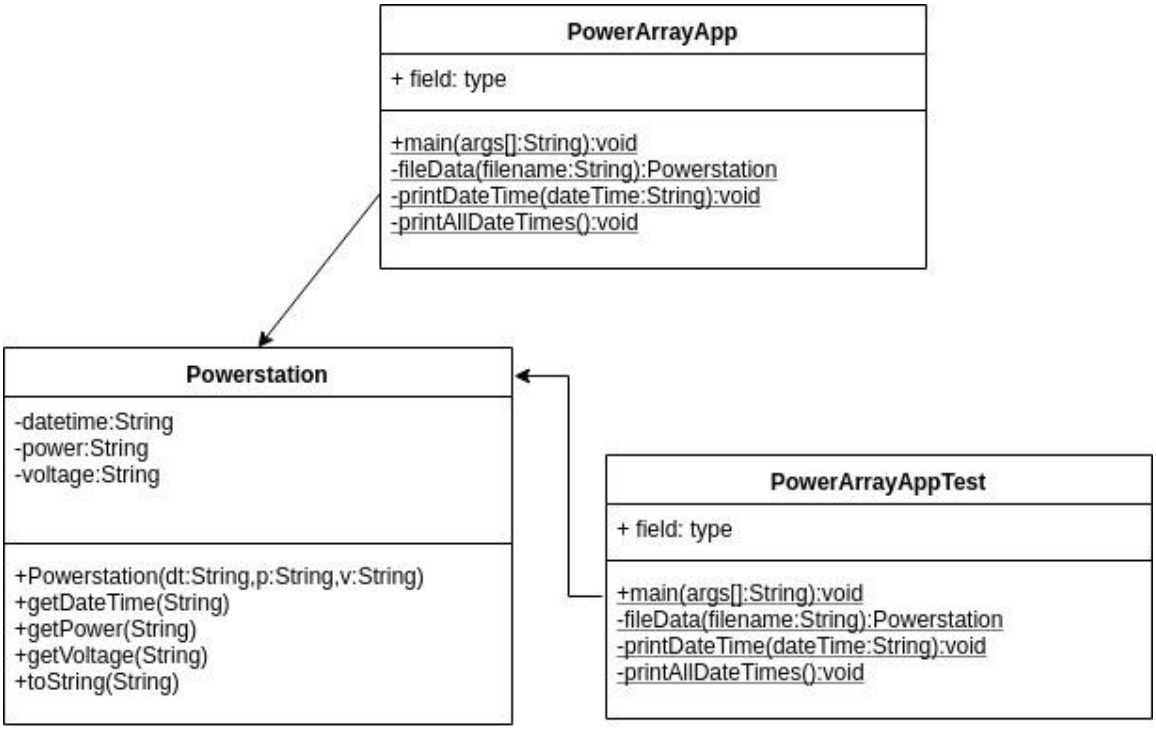

Figure 1: UML Diagram for Part 1

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**. Each element of the array in **PowerArrayApp** was of type **PowerStation**, thereby making it easier to store and retrieve the required data. The main method in **PowerArrayApp** calls on the method *printAllDateTimes* if args is empty and it calls on the method *printDateTime* if args is not empty. A java file **PowerArrayAppTest** was made when doing the testing for Part 5. It is the same as **PowerArrayApp**, besides the fact that it only returns the number of comparison operations 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:

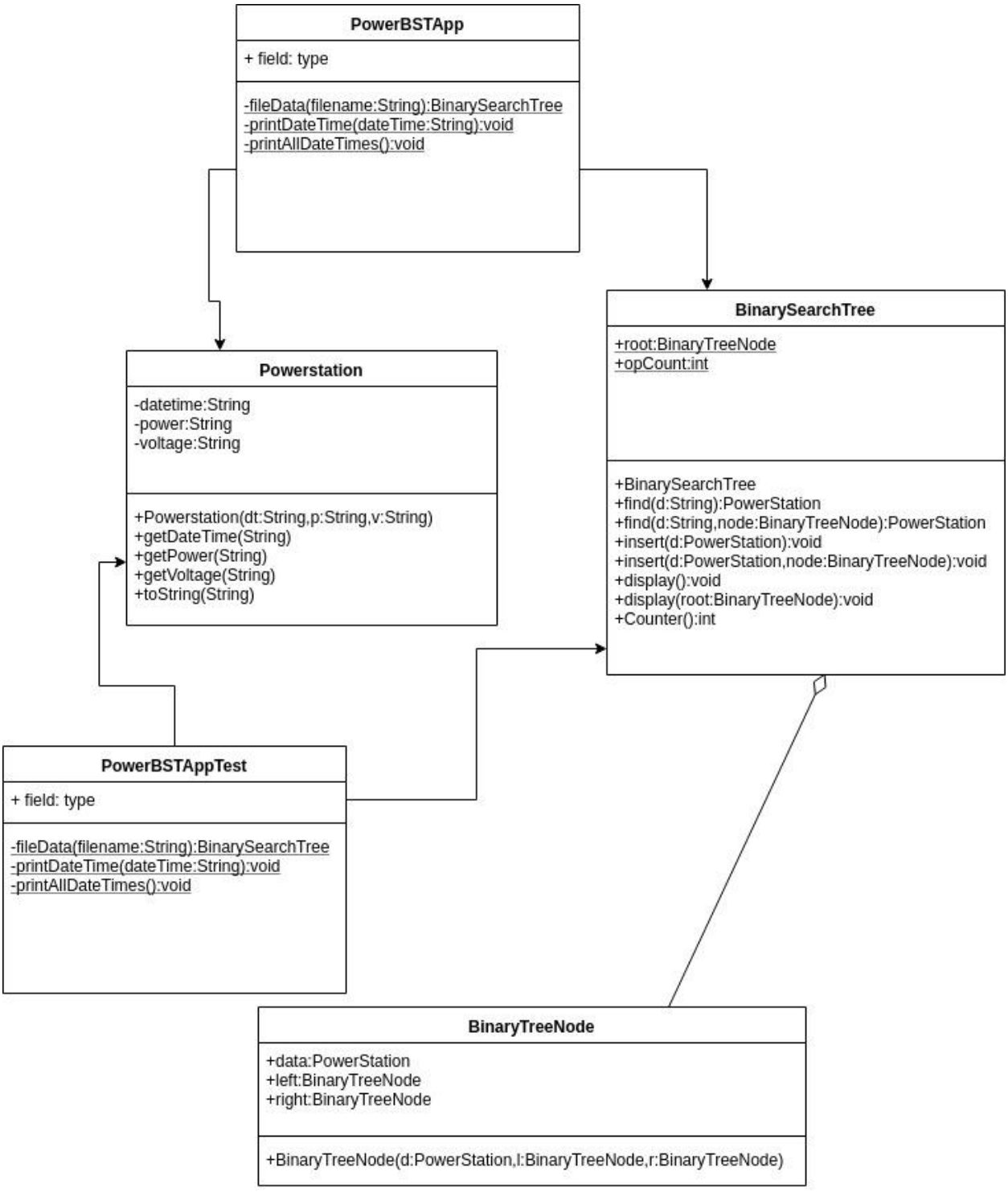

Figure 2: UML Diagram for Part 3 and Part 4

PowerStation was used in almost the same way for **PowerBSTApp**. The value of each node in the Binary Search Tree was an object of type **PowerStation**. **BinaryTreeNode**, which is an inner class inside **BinarySearchTree**, was used to create each node in the tree, the tree of which was created using **BinarySearchTree**. The code for **BinaryTreeNode** and **BinarySearchTree** was obtained from the original authors – Patrick Marais and SJ (via https://algorithms.tutorialhorizon.com/binary-search-tree-complete-implementation/). I modified their original code to build the required **BinarySearchTree**.

Goal and Execution of Experiment

The goal of the experiment is to determine whether a Binary Search Tree, which is a data structure, is more efficient than a traditional unsorted array when trying to find a specific data item in a set of given data.

The experiment was executed by creating a java program called **PowerArrayApp** to create an array of type PowerStation that would be iterated through using a for loop to find the required date/time being searched for by the user. 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. **PowerBSTApp** and **PowerArrayapp** were very similar and only differed in the methods for **printDateTime** and **printAllDateTimes**. For both **PowerArrayApp** and **PowerBSTApp**, code was added for instrumentation – meaning that the number of comparison operations were counted.

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 **PowerArrayApp** 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 **PowerArrayApp** and **PowerBSTApp**.

Trial Run for PowerArrayApp

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

Date/time	**Operation Counts**
Case 1:16/12/2006/19:51:00	1
Case 1: 16/12/2006/18:05:00	142
Case 1: 16/12/2006/17:41:00	495
Case 2: Unknown: 16/12/2006/17:41:72	500
Case 3: No parameters	500

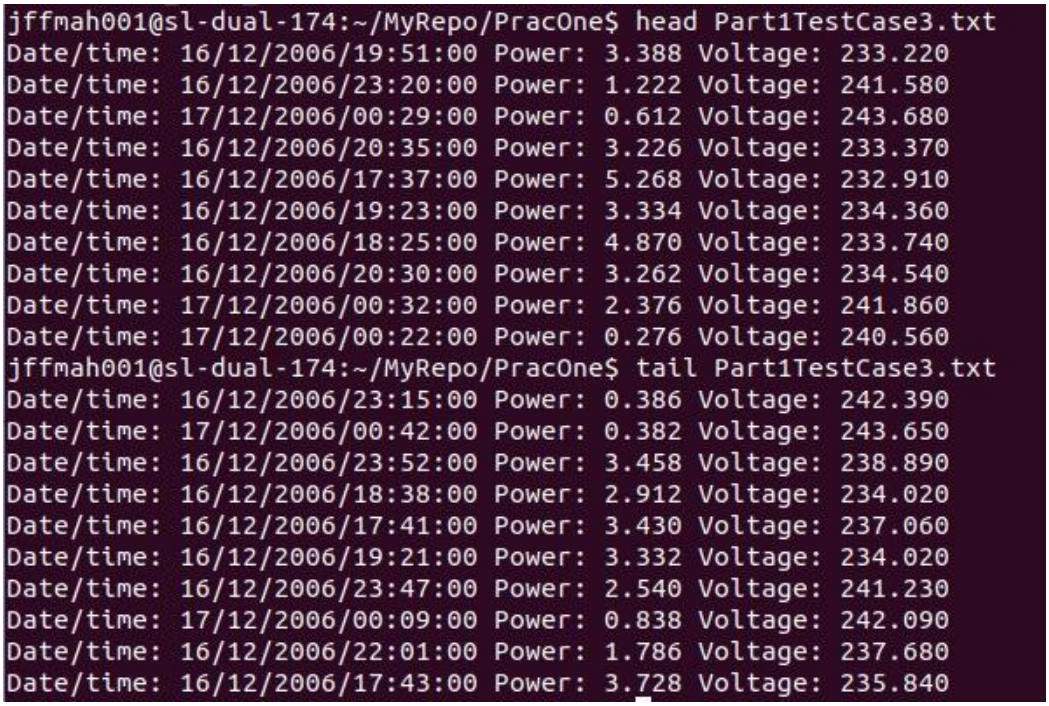

Figure 3: First and last ten lines when invoking **printAllDateTimes()**

Trial Run for PowerBSTApp

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

Date/time	**Operation Counts**
Case 1: 16/12/2006/19:51:00	1
Case 1: 16/12/2006/18:05:00	8
Case 1: 16/12/2006/17:41:00	10
Case 2: Unknown: 16/12/2006/17:41:72	9
Case 3: No parameters	500

```
jffmah001@sl-dual-174:~/MyRepo/PracOne$ 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
jffmah001@sl-dual-174:~/MyRepo/PracOne$ 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
jffmah001@sl-dual-174:~/MyRepo/PracOne$
```

Figure 4: First and last ten lines when invoking **printAllDateTimes()**

Results for Part 5

Array – Best Case

The big O notation for the best case of an array is O (1) which can be seen from the graph shown in Figure 5 below. This is because the best case for an array would be when the specified data element is the first item in the array. This is the reason why a horizontal line of the value 1 is displayed on the graph.

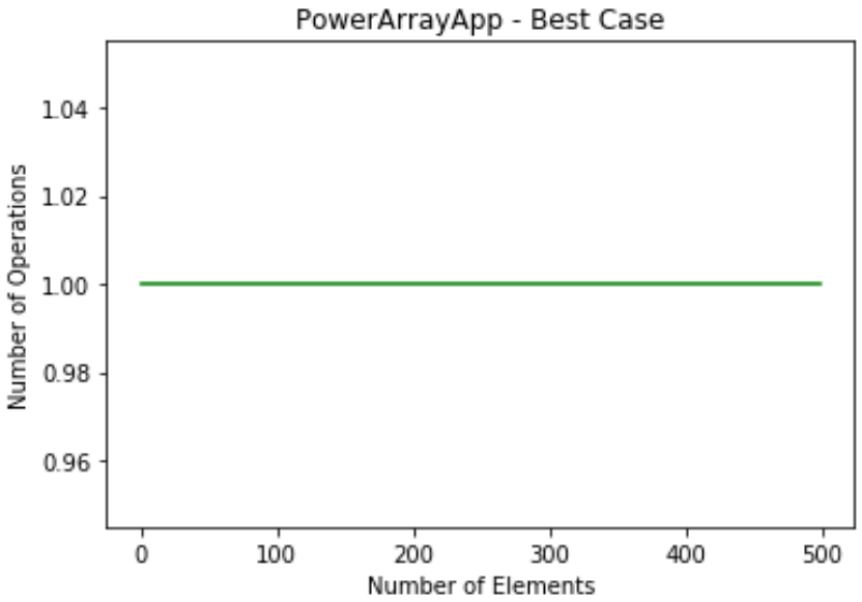

Figure 5: Best Case for Array

Array – Average Case

The big O notation for the average case of an array is ((N+1)/2) with N being the number of elements in the array. As can be seen from the graph in Figure 6 as the number of data elements increase so does the number of comparison operations – in a mostly linear relationship. While the average case of the array is not as bad as the worst case of an array due to the number of comparison operations being halved it is still not as efficient as the average case for a Binary Search Tree.

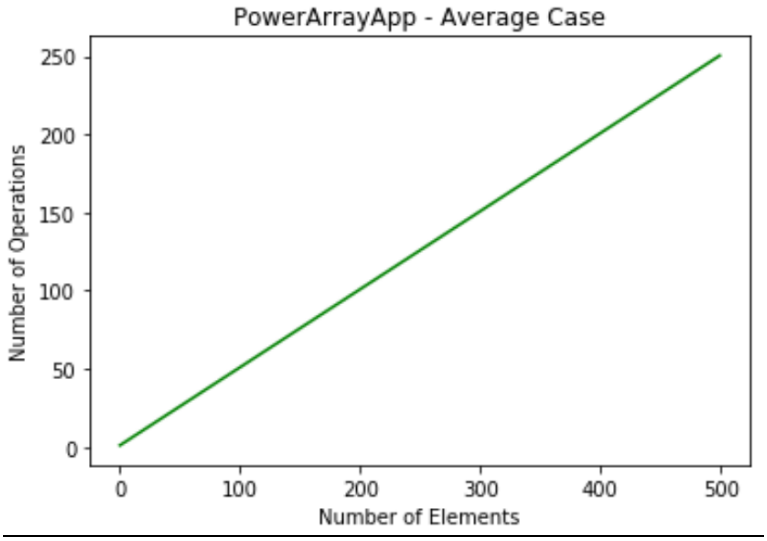

Figure 6: Average Case for Array

Array – Worst Case

The big O notation for the worst case of an array is O(n). This can be seen in Figure 7 where there is a linear relationship between the number of data elements and the number of comparison operations. This is because if an array is searching for a specified value it will have to iterate through the entire array if the specified data item is the last element in the array.

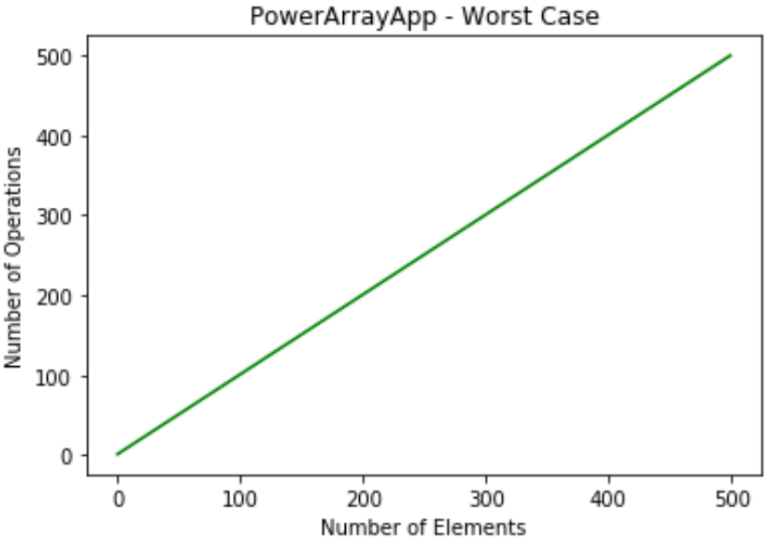

Figure 7: Worst Case for Array

Binary Search Tree – Best Case

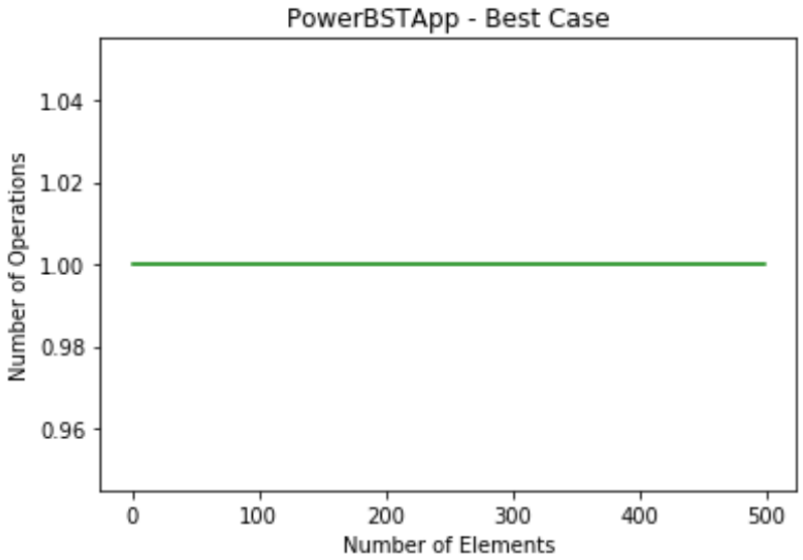

Figure 8: Best Case for Binary Search Tree

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 8 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

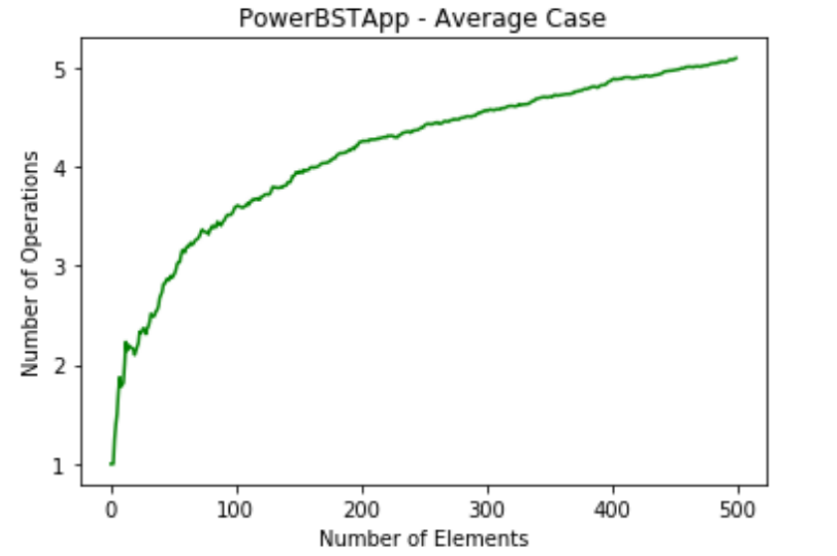

Figure 9: Average Case for Binary Search Tree

The big O notation for the average case of a Binary Search Tree is O(log n). The above graph in Figure 8 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

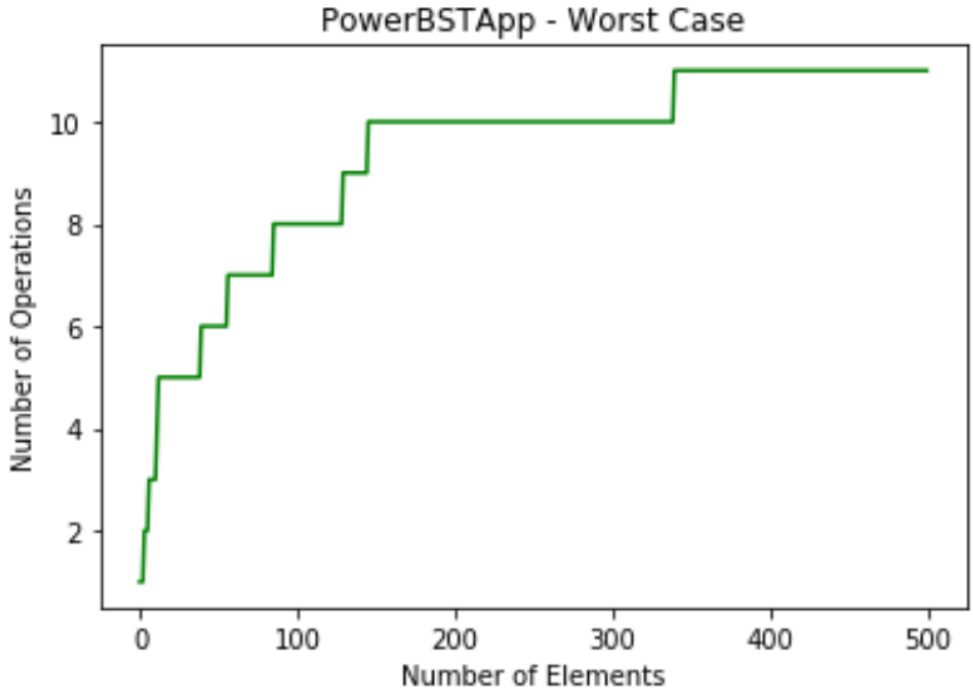

Figure 10: Worst Case for Binary Search Tree

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 10 is not as smooth as the graph in Figure 9 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. Therefore, it can be determined that even during the worst case a Binary Search Tree is still better than array because it's number of operations does not increase linearly and it is faster to run. The fact that a Binary Search Tree is faster than an array was evident when creating the subsets for testing. The subsets used for PowerBSTApp were completely generated almost an hour before those generated for PowerArrayApp.

Overall Result

Overall, from the above results it can be said that a Binary Search Tree is much more efficient and faster than an array when trying to find a specific data item in a set of given data. An array searches for data items linearly – meaning that it will iterate through the entire array until it finds the required data item or until it reaches the end of the array. That is why when you look at the complexity analysis of an array it is of O(n). A Binary Search Tree however – sorts the data that it stores into 2 different branches thereby decreasing the number of elements it needs to look at before finding the required data item by half. That is why when you look at the complexity analysis of a Binary Search Tree it is of O(log(n)).

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

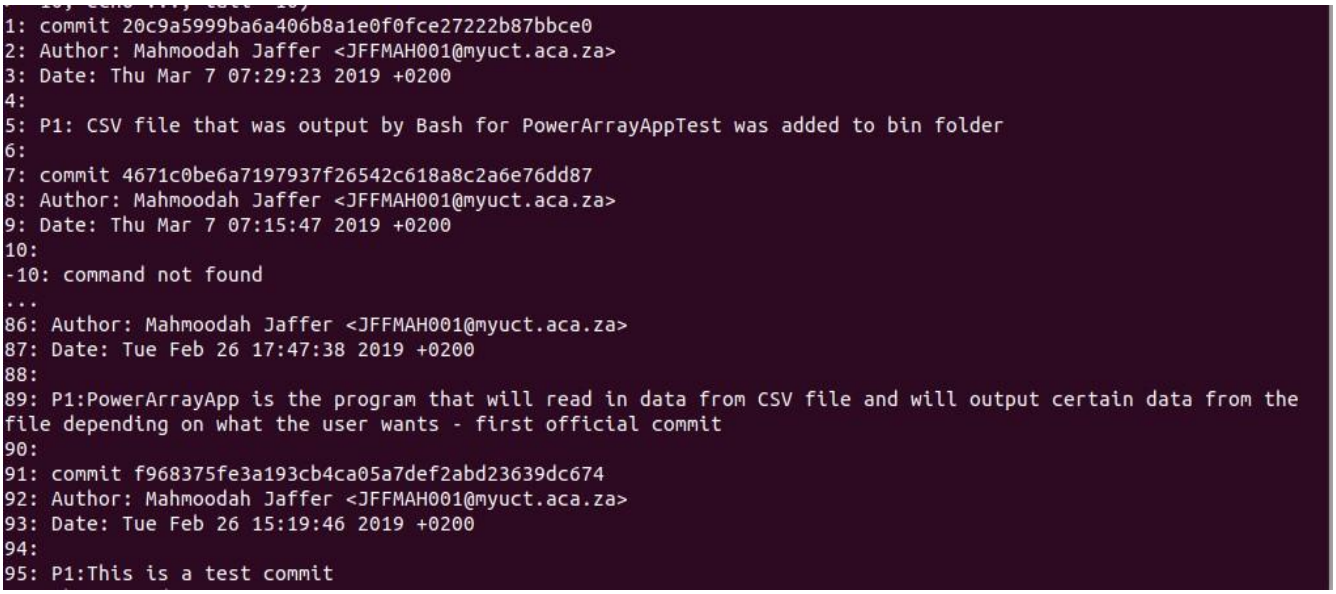

Figure 11: Summary statistics for git log usage