CSC2001F: Assignment 1

Comparison of Binary Search Tree and Array

6 March 2019

Leanne January

JNRLEA001

Contents

1. Experiment Goal and Execution
2. Explanation of OO Design
3. Evaluation of Trials in Part 2 and 4
4. Experiment Results (Part 5)
5. Statement of Creativity
6. Git Log

NOTE: All output from Part 2 and 4 are in the Output folder in the zipped file. As well as the .dat files used to create the graphs in gnuplot.

1. Experiment Goal and Execution

In this assignment I had to experiment inserting data and finding values in two different data structures – a Binary Search Tree and a traditional unsorted Array. The goal of this experiment was to determine the number of operations it took to find values in a Binary Search Tree and an Array, and compare the best, worst and average cases to each other.

A comma separated value (.csv) file was given, that contained real power consumption data. For each line in the file the values for Date/Time, Power (Global active power), and Voltage needed to be inserted as one data item into an Array and Binary Search Tree.

This is how the experiment was executed:

1. Design a class, called PowerUsage, which has a Date/Time. a Power and a Voltage attribute. In this way, these three values can be inserted as one data item.
2. Design a class, called PowerArrayApp, that reads in data from the given .csv file, assigns the relevant values from each line to a PowerUsage constructor and inserts the new PowerUsage object into an Array.
3. Add methods to the PowerArrayApp:
   a. One that prints all the objects in the array. This method is invoked if there are no arguments in the command line.
   b. One that is given a Date/Time and outputs that Date/Time and its corresponding Power and Voltage values. This method is invoked if there

are any Date/Time string arguments in the command line. For example, 'java PowerArrayApp "DateTime1" "DateTime2" … "DateTimeN"' will output N Date/Times and their corresponding Power and Voltage values.

4. Test that these methods work using 3 known Date/Times. 1 unknown Date/Time and a command line with no arguments and redirect the output into a text file.
5. Instrument the PowerArrayApp so that the Number of Operations needed to find a specific Date/Time is outputted as well. Repeat number 4.
6. Implement a Binary Search Tree, using a BinarySearchTree and a BinaryTreeNode class.
7. Design a class, called PowerBTSApp, which is executed in the same way that the PowerArrayApp is (outlined in steps 2-5), only the PowerUsage objects are inserted into the Binary Search Tree that was implemented in 5 and the command line will read 'java PowerBTSApp … '.
8. Create a subset of data, starting with the 1st Date/Time and going up in intervals of 25 so that the subset consists of the 1st, 25th, 50th, 100th, … 500th Date/Time in the .csv file (i.e. the data that was entered 1st, 25th or 100th into the data structure). Use the Binary Search Tree and Array to find all 20 Date/Times, taking note of the number of operations needed to find all Date/Times and for both the Binary Search Tree and Array. Record these results in separate text files for each data structure.
9. Use these results to build a graph for each data structure, and a graph with results for both in order to compare them (This is done using Gnuplot). The graphs should show how the insert order (whether a Date/Time is inserted 1st or 75th) corresponds to the number of operations it takes to find the Date/Time in the two data structures.
10. Calculate the Best, Worst and Average cases for both data structures and compare.

2. Explanation of Object Orientated Design

- PowerUsage

  Attributes

  Date/Time: string; Power: string; Voltage: string

  Methods

  Constructor: PowerUsage(String dateTime, String power, String voltage)
  getDateTime, getPower, getVoltage: returns String
  toString method

- PowerArrayApp

  Attributes

  dataFile: String, PowerUsage Array, opCount: int

  Methods

  insertData: String datafile, printAllDateTimes, printDateTime: accepts DateTime, returns String, getOpCount: returns int, resetOpCount
  Uses PowerUsage

- BinaryTreeNode:

  Attributes

  Key: String, value: PowerUsage, left, right: BinaryTreeNode

  Methods

  Constructor: BinaryTreeNode(String key, PowerUsage value), getLeft, getRight: return BinaryTreeNode, getValue: return PowerUsage, getKey: return String, insert: accepts key, value, find: accepts key, opCount
  Uses PowerUsage

- Binary Search Tree

  Attributes: root: BinaryTreeNode

  Methods

  getRoot: return BinaryTreeNode, insert: accepts key, value, find: accepts key, opCount, printTree: accepts BinaryTreeNode
  Uses BinaryTreeNode, Uses PowerUsage

- PowerBTSApp

  Attributes

  dataFile: String, BinarySearchTree: PowerUsage, opCount: int

  Methods

  insertData: String datafile, printAllDateTimes, printDateTime: accepts DateTime, returns String,
  Uses BinarySearchTree, Uses PowerUsage

3. Evaluation of Trials in Part 2 and 4

In Part 2 and Part 4 respectively, the code for the PowerArrayApp and PowerBTSApp classes was instrumented. This means that an *opCount* variable was added to both classes with the aim that it would keep track of the number of operations performed to find Date/Time values and return the total of this number. For the PowerArrayApp, the *opCount* variable tracked how many iterations the Array had to perform to find the Date/Time values. For the PowerBTSApp, the *opCount* variable tracked the number of recursions the Binary Search Tree had to perform to find the Date/Time values.

This is the output for Part 2 and 4:
Note: For the 3 known values I picked one from the top, one from the middle and one from the end.
These values are – 4th: "16/12/2006/20:35:00", 278th:"16/12/2006/22:56:00", 498th: "17/12/2006/00:09:00"

- Array (Part 2)

```
16/12/2006/20:35:00 Power: 3.226, Voltage: 233.370
Number of Iterations: 4
16/12/2006/22:56:00 Power: 2.488, Voltage: 239.030
Number of Iterations: 278
17/12/2006/00:09:00 Power: 0.838, Voltage: 242.090
Number of Iterations: 498
```

From outputPart2_knownDates

- Binary Search Tree (Part 4)

```
Number of Iterations: 3.
16/12/2006/20:35:00 Power: 3.226, Voltage: 233.370
Number of Iterations: 12.
16/12/2006/22:56:00 Power: 2.488, Voltage: 239.030
Number of Iterations: 12.
```

17/12/2006/00:09:00 Power: 0.838, Voltage: 242.090

From outputPart4_knownDates

These outputs show that the number of operations needed to find a specific value in a data structure is less if one is using a Binary Search Tree. In fact, complexity analysis for the Array is exactly O(n) – meaning that if a value is the 4th element of the array, it will take exactly 4 times to find it. Theoretically, the complexity analysis for a Binary Search Tree's average case minimum is O(log n) which is a lot smaller than the number of operations shown in my output. However, the Big-Oh complexity analysis is designed to work with rather large numbers, therefore because I am working with very small numbers this discrepancy is expected.

In conclusion, these outputs show how much more efficient it is to use a Binary Search Tree when finding a specific value in a database.

4. Experiment Results (Part 5)

To begin, I created a subset of the data from the .csv file. I recorded only Date/Time values (to be searched for in the data structures), starting with the 1st Date/Time and going up in intervals of 25 so that the subset consists of the 1st, 25th, 50th, 100th, … 500th Date/Time in the .csv file (i.e. the data that was entered 1st, 25th or 100th into the data structure). In total there were 20 Date/Time items to be searched for:

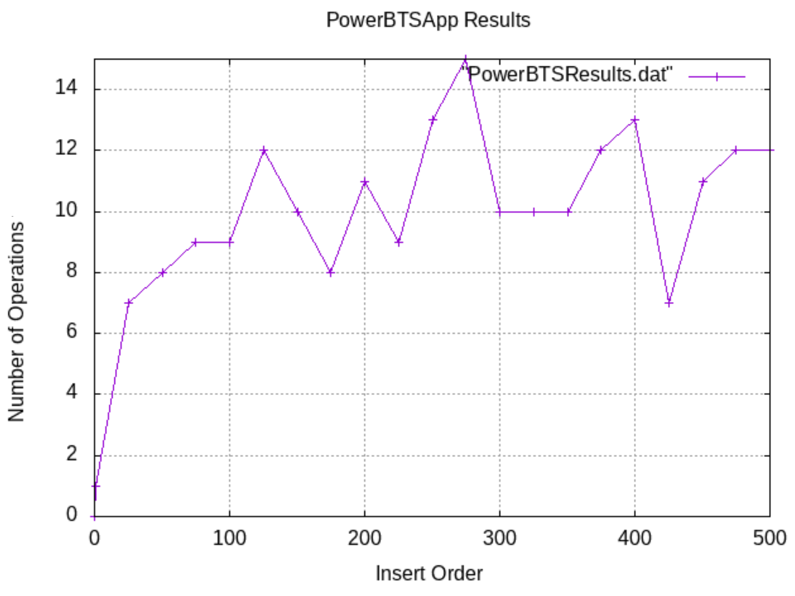
S = {1: 16/12/2006/19:51:00, 25: 17/12/2006/00:38:00, 50: 17/12/2006/01:23:00, 75: 17/12/2006/01:25:00, 100: 16/12/2006/17:56:00, 125: 16/12/2006/22:19:00, 150: 17/12/2006/00:47:00, 175: 16/12/2006/18:56:00, 200: 16/12/2006/23:34:00, 225: 16/12/2006/19:00:00, 250: 17/12/2006/01:11:00, 275: 16/12/2006/23:49:00, 300: 16/12/2006/17:48:00, 325: 16/12/2006/19:57:00, 350: 16/12/2006/19:29:00, 375: 16/12/2006/19:06:00, 400: 16/12/2006/23:37:00, 425: 16/12/2006/19:49:00, 450: 16/12/2006/21:20:00, 475: 16/12/2006/19:31:00, 500: 16/12/2006/17:43:00}

I recorded the output (which consisted of the number of operations for each Date/Time value) from each data structure into two text files, one for each data structure.

Array: PowerArrayAppResults.dat,   Binary Search Tree: PowerBTSResults.dat.

I then used gnuplot to plot graphs based on these text files.

Binary Search Tree:



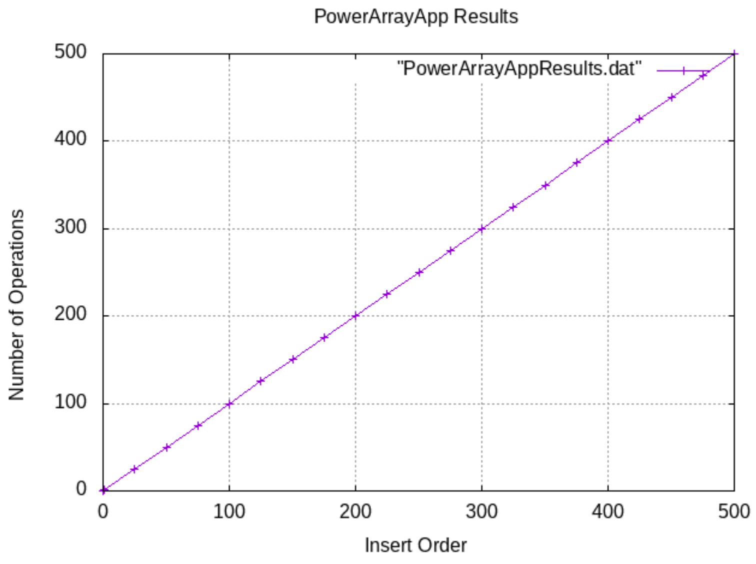
This graph measures, on the x-axis, the 'insert order' which is the number given to a Date/Time value based on when it was inserted into the Binary Search Tree, and on the y-axis, the 'number of operations', which is how many operations (recursions) it took the Binary Search Tree to find a given key.

<u>Best case</u>: In this graph, the best case is O(1) – It took one operation to find the key, because the key was the root node in the structure. Theoretically, O(1) is also the best case for a Binary Search Tree.

<u>Average case:</u> In this graph, the average case is 10,45 operations. Theoretically, the minimum average case is O(logN), which for this example is 2.31 operations on average. Although, as I mentioned earlier, the Big-Oh complexity analysis is meant for larger numbers and therefore, the use of such small numbers will result in discrepancies such as this.

<u>Worst case</u>: In this graph, the worst case took 15 operations. Theoretically, at best the worst case scenario is O(h) – h being the height of the tree (the key could be at the bottom of the tree), and at absolute worst the complexity analysis for the worst case is O(N) . This happens when a binary search tree acts like a linked list (which happens when data is inserted in ascending or descending order). In this case the height of the tree is equal to the number of nodes.

Array:



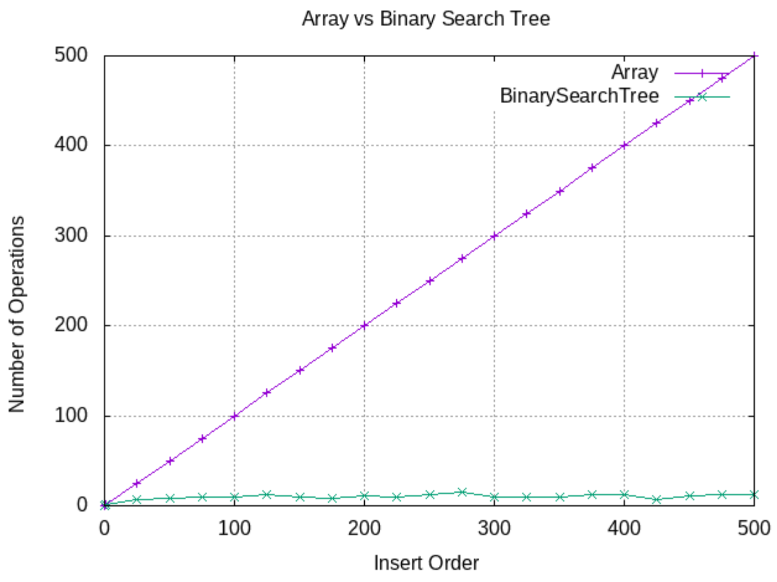
This graph measures, on the x-axis, the 'insert order' which is the number given to a Date/Time value based on when it was inserted into the Array – essentially which element of the array it was, and on the y-axis, the 'number of operations', which is how many operations (iterations) it took the Array to find a given key.

Best case: In this graph, the best case is O(1) – It took one operation to find the key, because the key was the first element in the Array. Theoretically, O(1) is also the best case for an Array.

Average case: In this graph, the average number of operations is 262.5. Theoretically, the average case for an Array is O(N). This is because an Array always has to iterate N times to get to the Nth element. Therefore if the key is the 60th element, the operation will be O(60).

Worst case: In this graph, the worst case took 500 operations. This is because the key was the last element in the array. Theoretically, the worst case is the same as the average case, O(N) .

Combined:



In conclusion:

- The time complexity of Arrays is therefore dependent on which element the key is searching for in the Array, while the time complexity of a Binary Search Tree depends on how deep the tree is, or how great the height of the tree is – which depends on how the data is ordered before it is sorted into the tree.

- A Binary Search Tree is a better data structure to use when data is constantly being searched as its average case produces much smaller values than the average case of an Array. Although, at its very worst, a Binary Search Tree will have the same worst case as an Array's average and worst case.
- The graph above clearly shows that on average a Binary Search Tree results in much fewer operations and therefore much better performance than an Array.

5. Statement of Creativity

1. I edited the PowerArrayApp and the PowerBSTApp to accept multiple string arguments in the command line and parse them all into the printDateTime method – essentially allowing any number of Date/Times to be searched in one command line.
   For example, 'java PowerArrayApp "DateTime1" "DateTime2" … "DateTimeN"' will output N Date/Times and their corresponding Power and Voltage values.
2. I created the PowerUsage class to store the three different values from the .csv file in data item.

6. Git log

```
commit e2d04a79764070a972bdde664f623e1d6134d099
Author: Leanne January <JNRLEA001@myuct.ac.za>
Date: Wed Mar 6 11:01:51 2019 +0200

final 2 - Modified make file to account for directories

commit f14f1167ce3fa7720c4356552406c6fcd136ede1
Author: Leanne January <JNRLEA001@myuct.ac.za>
Date: Wed Mar 6 11:01:06 2019 +0200
```

```
10:

...

62: Author: Leanne January <JNRLEA001@myuct.ac.za>

63: Date: Sun Mar 3 14:55:53 2019 +0200

64:

65: 1.2 - Made PowerArrayApp which inserts values from cleaned_data.csv into an array of PowerUsage objects. Modified syntax errors in PowerUsage.java and updated Makfile.

66:

67: commit 90b4b80436b987c3f3d6dc3cd401252a1e7f3040

68: Author: Leanne January <JNRLEA001@myuct.ac.za>

69: Date: Sun Mar 3 10:00:41 2019 +0200

70:

71: 1.1 - Made a PowerUsage Object which stores date/time, power and voltage and a Makefile for it.
```