CSC2001F: Assignment 2

Comparison of Binary Search Tree and AVL Tree

3/22/2019

Leanne January

JNRLEA001

Contents

NOTE: All output is in the Output folder in the zipped file. This includes the .dat files used to create graphs in gnuplot.

1. Experiment Goal and Execution

In this assignment, I implemented a Binary Search Tree and an AVL Tree and conducted an experiment that tested how many operations it took to insert and search for values in each data structure. The goal of the experiment was to determine which data structure was more efficient and to achieve this I used the Big-O complexity analysis – which allowed me to compare the best, worst and average cases of each data structure to each other.

I executed this experiment by instrumenting the code in the Binary Search Tree and AVL Tree classes with the result being that I could see how many operations were being performed to insert and search for values. Each operation count for each data structure was written to a separate text file.

For the first part of the experiment, I compared the time complexities of the Binary Search Tree and the AVL Tree which both inserted unsorted data. To achieve this, I used the operation counts that were written to text files to draw up graphs. I recorded the best, worst and average case for each data structure.

For the second part of the experiment, I compared the time complexities of the Binary Search Tree and the AVL Tree using the same data as input – only it was sorted (in ascending order). I recorded the best, worst and average cases and compared them the two data structures to each other and to the cases recorded in the first part of the experiment.

2. Explanation of Object Orientated Design

- PowerUsage
  - The PowerUsage class stores the Date/Time, Power and Voltage values from the cleaned_data.csv files as 3 different Strings within one object.
    <u>Methods</u>
    Constructor: PowerUsage(String dateTime, String power, String voltage)
    getDateTime, getPower, getVoltage: and a toString method

- PowerBSTApp
  - The main function of the PowerBTSApp is to read values from the given csv file and insert the relevant data into different PowerUsage objects. These PowerUsage objects are inserted into a Binary Search Tree and, given a Date/Time key, individual PowerUsage objects can be accessed. Alternatively, the program can print the entire Binary Search Tree.

    - It also has other functions, such as to write the search and insert operation counts into different text files.
    - The PowerBSTApp therefore uses the PowerUsage and BinarySearchTree classes.

- BinaryTreeNode:
    - The BinaryTreeNode class is a helper class for the BinarySearchTree class.
    - The BinaryTreeNode class stores PowerUsage objects as values and therefore uses the PowerUsage class.

- Binary Search Tree
    - The Binary Search Tree class implements the data structure known as a Binary Search Tree.
    - The Binary Search Tree class consists of multiple BinaryTreeNodes, and therefore uses the BinaryTreeNode class.

- PowerAVLApp
    - The main function of the PowerAVLApp is to read values from the given csv file and insert the relevant data into different PowerUsage objects. These PowerUsage objects are inserted into an AVL Tree and, given a Date/Time key, individual PowerUsage objects can be accessed. Alternatively, the program can print the entire AVL Tree.
    - It also has other functions, such as to write the search and insert operation counts into different text files.
    - The PowerAVLApp therefore uses the PowerUsage and AVLTree classes.

- AVLTreeNode:
    - The AVLTreeNode class is a helper class for the AVLTree class.
    - The AVLTreeNode class stores PowerUsage objects as values and therefore uses the PowerUsage class.

- AVL Tree
    - The AVL Tree class implements the data structure known as an AVL Tree.
    - The AVL Tree class consists of multiple AVLTreeNodes, and therefore uses the AVLTreeNode class.

3. Evaluation of Trials in Part 2 and 4

The code in both the Binary Search Tree and the AVL Tree classes were instrumented in order to determine how many operations were being performed to insert and search for values in the data structure. Each operation count for each data structure was written to a separate text file.

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.

3 known Date/Time keys:

4th:	"16/12/2006/20:35:00"
278th:	"16/12/2006/22:56:00"
498th:	"17/12/2006/00:09:00"

1 Unknown Date/Time key:
"16/28/2006/22:00:00"

```
AVL TREE TEST:
(PART 1/2):
1. 3 KNOWN DATES
one from top, middle and bottom of cleaned_data.csv

16/12/2006/20:35:00 Power: 3.226, Voltage: 233.370
16/12/2006/22:56:00 Power: 2.488, Voltage: 239.030
17/12/2006/00:09:00 Power: 0.838, Voltage: 242.090

2. 1 UNKNOWN DATE
Date/time not found.

3. NO PARAMETERS - PRINTS ALL DATE TIMES
(first 10)
17/12/2006/01:43:00 Power: 2.664, Voltage: 243.310
17/12/2006/01:42:00 Power: 3.800, Voltage: 241.780
17/12/2006/01:41:00 Power: 4.500, Voltage: 240.420
17/12/2006/01:40:00 Power: 3.214, Voltage: 241.920
17/12/2006/01:39:00 Power: 1.670, Voltage: 242.210
17/12/2006/01:38:00 Power: 3.680, Voltage: 239.550
17/12/2006/01:37:00 Power: 3.944, Voltage: 239.790
17/12/2006/01:36:00 Power: 3.746, Voltage: 240.360
17/12/2006/01:35:00 Power: 3.954, Voltage: 239.840
17/12/2006/01:34:00 Power: 2.358, Voltage: 241.540

(last 10)
16/12/2006/17:33:00 Power: 3.662, Voltage: 233.860
16/12/2006/17:32:00 Power: 3.668, Voltage: 233.990
16/12/2006/17:31:00 Power: 3.700, Voltage: 235.220
16/12/2006/17:30:00 Power: 3.702, Voltage: 235.090
16/12/2006/17:29:00 Power: 3.520, Voltage: 235.020
16/12/2006/17:28:00 Power: 3.666, Voltage: 235.680
16/12/2006/17:27:00 Power: 5.388, Voltage: 233.740
16/12/2006/17:26:00 Power: 5.374, Voltage: 233.290
16/12/2006/17:25:00 Power: 5.360, Voltage: 233.630
16/12/2006/17:24:00 Power: 4.216, Voltage: 234.840
```

from OutputAVL

```
BINARY SEARCH TREE TEST:
(PART 3/4):
1. 3 KNOWN DATES
one from top, middle and bottom of cleaned_data.csv
```

```
16/12/2006/20:35:00 Power: 3.226, Voltage: 233.370
16/12/2006/22:56:00 Power: 2.488, Voltage: 239.030
17/12/2006/00:09:00 Power: 0.838, Voltage: 242.090

2. 1 UNKNOWN DATE
Date/time not found.

3. NO PARAMETERS - PRINTS ALL DATE TIMES
(first 10)
17/12/2006/01:43:00 Power: 2.664, Voltage: 243.310
17/12/2006/01:42:00 Power: 3.800, Voltage: 241.780
17/12/2006/01:41:00 Power: 4.500, Voltage: 240.420
17/12/2006/01:40:00 Power: 3.214, Voltage: 241.920
17/12/2006/01:39:00 Power: 1.670, Voltage: 242.210
17/12/2006/01:38:00 Power: 3.680, Voltage: 239.550
17/12/2006/01:37:00 Power: 3.944, Voltage: 239.790
17/12/2006/01:36:00 Power: 3.746, Voltage: 240.360
17/12/2006/01:35:00 Power: 3.954, Voltage: 239.840
17/12/2006/01:34:00 Power: 2.358, Voltage: 241.540

(last 10)
16/12/2006/17:33:00 Power: 3.662, Voltage: 233.860
16/12/2006/17:32:00 Power: 3.668, Voltage: 233.990
16/12/2006/17:31:00 Power: 3.700, Voltage: 235.220
16/12/2006/17:30:00 Power: 3.702, Voltage: 235.090
16/12/2006/17:29:00 Power: 3.520, Voltage: 235.020
16/12/2006/17:28:00 Power: 3.666, Voltage: 235.680
16/12/2006/17:27:00 Power: 5.388, Voltage: 233.740
16/12/2006/17:26:00 Power: 5.374, Voltage: 233.290
16/12/2006/17:25:00 Power: 5.360, Voltage: 233.630
16/12/2006/17:24:00 Power: 4.216, Voltage: 234.840
```
from OutputBST

The files with the corresponding search and insert operations can be found in the Output folder.

<u>Searching:</u>

```
Search Count BST: 4, 12, 12
Search Count AVL: 4, 7, 12
```
(from OutputPart1-4 folder)

From this data, the average number of operations to search for a Binary Search Tree is 9.33, and for an AVL Tree is 7.66.
Therefore, the number of operations needed to search for a specific value is less if one is using an AVL Tree. Theoretically, the complexity analysis for a Binary Search Tree's average case is O(log n) which is the same for an AVL Tree, therefore the number of operations needed to search for a value should be more or less the same. The difference in number of operations is due to the fact that Binary Search Trees are unbalanced, while AVL Trees are balanced. It takes longer for the algorithm to search for values in unbalanced trees as they might be searching down one really long subtree for one value, whereas in an AVL Tree, all the subtrees are the same height or differ by 1. This means that the average case for a Binary Search Tree will have a greater number of operations performed than an AVL Tree. Most often, the worst case of Binary Search Tree is 0(h), h being the height of the tree – this is also the worst case for an AVL Tree. However, as a

Binary Tree can have one or two really long subtrees, the height of a Binary Search Tree is mostly greater than the height of an AVL Tree. The Binary Search Tree also may degenerate into a linked list if the input is ordered which makes the time complexity for the very worst case O(n). Therefore, an AVL Tree performs better than a Binary Search Tree when searching for a specific value in both the average and worst cases and, so it is the more efficient data structure.

Inserting:
These values are the same as those mentioned in Experiment Results (Part 5) and are discussed there.

4. Experiment Results (Part 5)

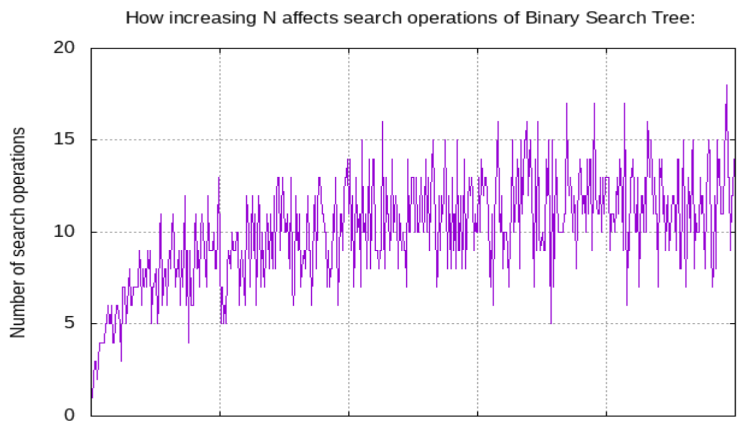
For Part 5, I searched for every single Date/Time value inserted into the data structures from the cleaned_data.csv file. I did this by copying the Date/Time values from the cleaned_data.csv file into a text file and used the command line:

Java PowerBTSApp/PowerAVLApp "FileWithSearchLeys.txt" > outputBST/AVL

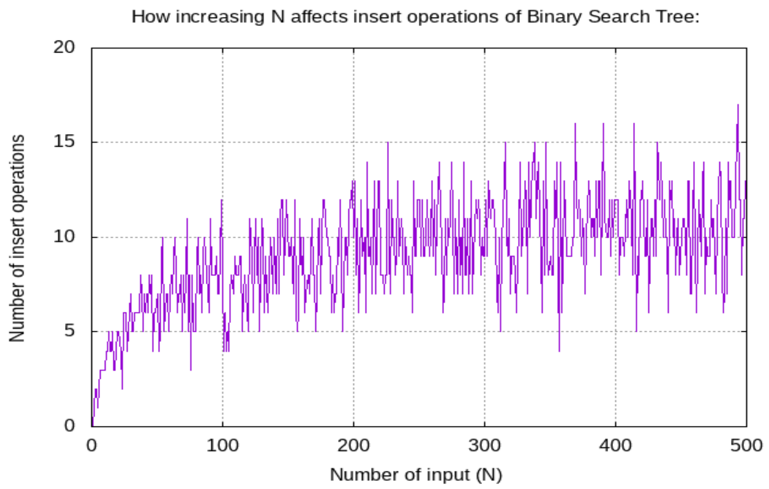
The redirected the output consisted of a list of the correct Date/Times with their corresponding Power and Voltage values. Two other text files were created which consisted of the number of operations to insert each value and search for each value. I converted these two files into .dat files and used them to draw up graphs (using gnuplot) which show the affect that increasing numbers of input have on search and insert operations.

Here are the results:

Binary Search Tree:
Search:
This graph measures, on the x-axis, the 'input number' (number denotes the place it was inserted, eg. Input number 11 was the 11th value inserted), and on the y-axis, the 'number of search operations', which is how many operations it took the Binary Search Tree to find a given key.



Insert:
This graph measures, on the x-axis, the 'input number' (number denotes the place it was inserted, eg. Input number 11 was the 11th value



inserted), and on the y-axis, the ‘number of insert operations’, which is how many operations it took the Binary Search Tree to insert a key.

Best case: The best number of both search and insert operations recorded from this data is 1. This means that it took one operation to find the matching key and it took one operation to insert a key. This is because, both times, that key was the root node in the structure. Theoretically, O(1) is also the best case for searching and inserting in a Binary Search Tree.

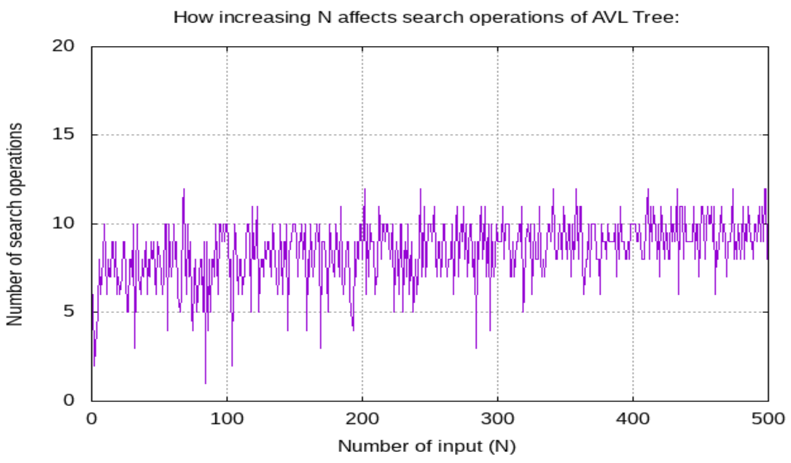
Average case: The average number of operations recorded from this data is 10.092. Theoretically, the minimum average case is O(logN), which for this example is 2.7 (O(log500)) operations on average. The average number of insert operations recorded from this data is 9.092. It should be O(logN), which is also 2.7 operations on average. The difference can be attributed to two factors. Firstly, 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. Secondly, as I mentioned earlier, the theoretical results do not account for how the imbalance of the tree may affect operations (particularly with such a small number of data).

Worst case: The worst number of search operations recorded from this data is 17. 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). The height of this tree is 17, therefore the minimum worst case is O(17). At absolute worst the complexity analysis for the worst case is O(N). This happens when a binary search tree degenerates into a linked list (which happens when data is inserted in ascending or descending order). The worst number of insert operations recorded from this data is 17, and because h=17 the time complexity is O(h).

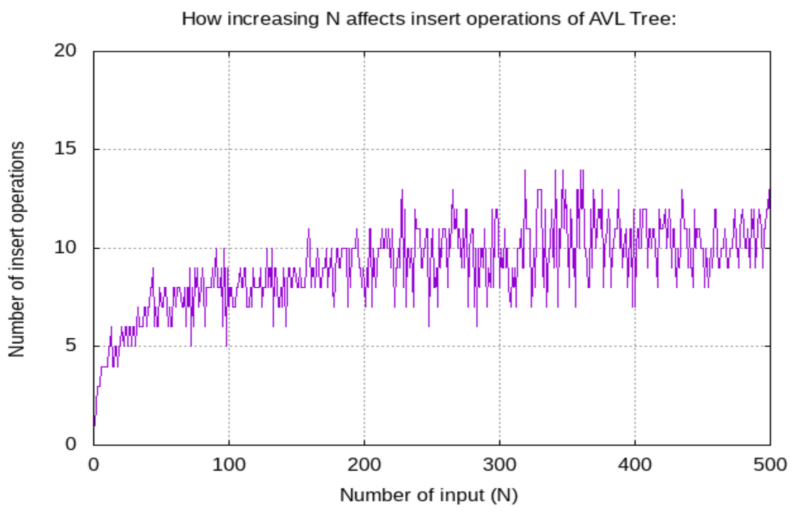
AVL Tree:

Search:

This graph measures, on the x-axis, the ‘input number’ (number denotes the place it was inserted, eg. Input number 11 was the 11th value inserted), and on the y-axis, the ‘number of search operations’, which is how many operations it took the AVL Tree to find a given key.



Insert:
This graph measures, on the x-axis, the 'input number' (number denotes the place it was inserted, eg. Input number 11 was the 11th value inserted), and on the y-axis, the 'number of insert operations', which is how many operations it took the AVL Tree to insert a key.



Best case: The best number of both search and insert operations recorded from this data is 1. This means that it took one operation to find the matching key and it took one operation to insert a key. This is because, both times, that key was the root node in the structure. Theoretically, O(1) is also the best case for searching and inserting in an AVL Tree.

Average case: The average number of search operations recorded from this data is 8.476. Theoretically, the minimum average case is O(logN), which for this example is 2.7 (O(log500)) operations on average. The average number of insert operations recorded from this data is 9.128 operations. It should be O(logN), which is also 2.7 operations on average. The differences can be attributed to the fact that the Big-Oh complexity analysis is meant for larger numbers (so the use of such small numbers will result in discrepancies such as this).

Worst case: The worst number of search operations recorded from this data is 12. Theoretically, at the worst case scenario is O(h) – h being the height of the tree (the key could be at the bottom of the tree). The height of this tree is 14, therefore the worst case is O(14). The worst number of insert operations recorded from this data is 14, and because h=14 the time complexity is O(h).

5. Experiment Results (Part 6)

For Part 6 I did the exact same as Part 5, only I inserted ordered data into the array, and recorded the best, average and worst case values for searching and inserting in the two different data structures.

AVL Tree:

Best:	Inserting: 1 operation	Searching: 1 operation (O(1))
Average:	Inserting: 8.978 operations	Searching: 7.996 operations (O(logN))
Worse:	Inserting: 10 operations	Searching: 9 operations (O(h))

Total Insertion Operations: 4489
Total Search Operations: 3998

Binary Search Tree:

Best:	Inserting: 1 operation	Searching: 1 operation (O(1))
Average:	Inserting: 250.5 operations	Searching: 250.5 operations (O(N/2))
Worse:	Inserting: 500 operations	Searching: 500 operations (O(N))

Total Insertion Operations: 125 250
Total Search Operations: 125 250

6. Conclusion

The time complexity of a Binary Search Tree and an AVL Tree depends on how deep the tree is, or how great the height of the tree is. AVL Trees have a method that balances the tree after each insertion and deletion takes place. A tree is balanced when the heights of the left and the right subtree are equal or differ by one. Binary Search Trees have no such method, and are therefore unbalanced. Unbalanced trees have greater heights than balanced trees as one or more subtrees may be extremely longer than another (making the height of the whole tree very large). This can be seen in the terrible performance of the Binary Search Tree in Part 6 where h = N (height = number of inputs). Therefore, as unbalanced trees have greater heights, they have larger average and worst case time complexities and are less efficient than balanced trees. Thus, an AVL Tree is a more efficient data structure than a Binary Search Tree.

7. 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.
3. I included code to calculate the height of both the AVL and the Binary Search Trees. This was to aid me in my analysis of the time complexities.

8. Git log

```
commit 85d513eacbd2483848f1b82fe4da0b2a46485061
Author: Leanne January <JNRLEA001@myuct.ac.za>
Date: Fri Mar 22 07:00:19 2019 +0200

9. Edited what gets written to the Output Files for PowerAVL and PowerBST.

commit c6ff8780b0f2559c3f33065825525a049ba98f56
Author: Leanne January <JNRLEA001@myuct.ac.za>
Date: Fri Mar 22 03:27:27 2019 +0200

...
Author: Leanne January <JNRLEA001@myuct.ac.za>
Date: Fri Mar 15 16:16:43 2019 +0200

2. Modified BinarySearchTree and BinaryTreeNode - Changed code to look more like the code given to us in class. Added method that calculates and returns height. Corrected name of PowerBSTApp and added a new line that prints out the height.

commit c8d22cfc244ce92a21fe97d3a4c89b29000672e2
Author: Leanne January <JNRLEA001@myuct.ac.za>
Date: Fri Mar 15 12:19:03 2019 +0200

1. Copied all files from Assignment 1 that can be reused.
```