GHRAHM004 – Ahmed Ghoor - CSC2001F Assignment 2

**Goal:**

Compare the efficiency of an AVL Tree and a Regular Binary Search Tree to insert and search through data.

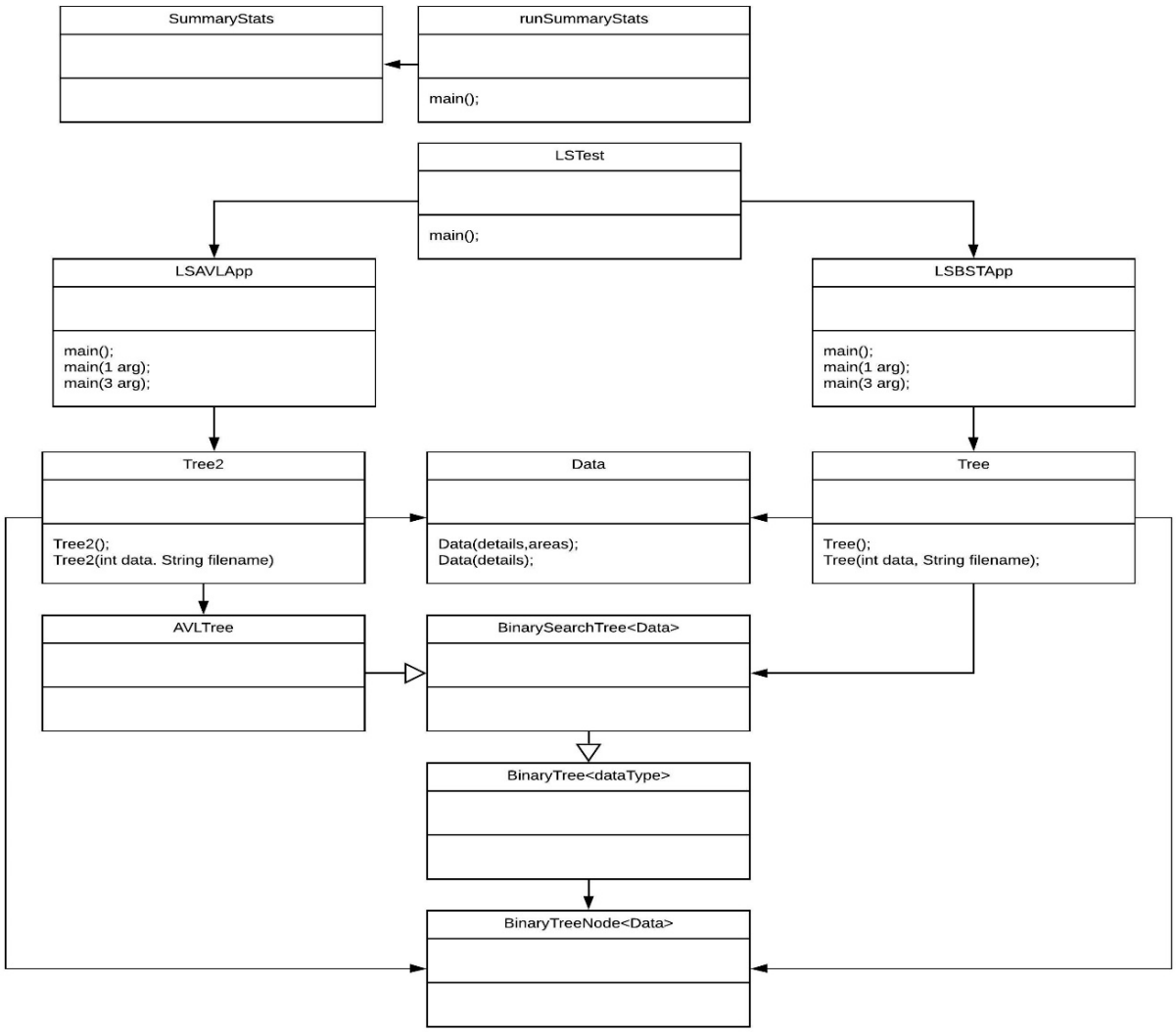
**How:**

Created a regular Binary Search Tree App and an AVL Tree App to read in load shedding data from a file, insert them into the respective data structures and search through them for specific parameters.

Added counters to the code to record the number of comparison operations made when data is being inserted or searched. The count will be provided whenever the user requests an output that requires these operations.

Finally, in order to efficiently compare the data structures, additional applications were created to randomize the data in the input file, repeat the experiment multiple times to collect a large sum of useful data into convenient csv files and then summarize the data in those csv files so that understandable graphs may be plotted.

**Simple Objected Orientated class structure diagram and explanation**



***BinaryTree*** creates methods to traverse through ***BinaryTreeNode*** objects.
***BinsarySearchTree*** inherits ***BinaryTree*** and creates methods to add, remove and search through nodes, as well as count the number of key operations it takes to do so.
***AVLTree*** inherits ***BinarySearchTree*** and adds AVL features such as balancing.
***Tree*** and ***Tree2*** use the regular ***BinarySearchTree*** and ***AVLTree*** respectively to create and use trees using data from a file. Both store the data using ***Data*** objects.
***LSBSTApp*** and ***LSAVLApp*** run ***Tree*** and ***Tree2*** respectively to create and search through trees.
***LSTest*** runs the ***LSAVLApp*** and ***LSBSTApp*** multiple times to output data into csv files which are then used by ***SummaryStats*** to summarize the data for the graphs.

**Part 2 & 4: Test values used in Trial runs with respective queried output & operations counts.**

1) Command: e.g. *java LSBSTApp "1_2_00"* OR *java LSAVLApp "blabla"*

| | Binary Search Tree (Regular) | | | AVL Tree | | |
|---|---|---|---|---|---|---|
| Values | Output | Insert Op's | Search Op's | Output | Insert Op's | Search Op's |
| 1_2_00 | 13 | 270100 | 3 | 13 | 34927 | 13 |
| 4_22_04 | 4, 12, 16, 8 | 270100 | 135 | 4, 12, 16, 8 | 34927 | 19 |
| 7_22_14 | 9, 1, 5, 13, 6, 14, 2 | 270100 | 309 | 9, 1, 5, 13, 6, 14, 2 | 34927 | 23 |
| 1_2_3 | Areas not found | 270100 | 30 | Areas not found | 34927 | 26 |
| 1 | Areas not found | 270100 | 6 | Areas not found | 34927 | 22 |
| blabla | Areas not found | 270100 | 408 | Areas not found | 34027 | 26 |

[bin/TestOutput]

2) Command: (Output in Report is just the first and last 10 lines)
   *a) java LSBSTApp*

1_10_00 15
1_10_02 16
1_10_04 1
1_10_06 2
1_10_08 3
1_10_10 4
1_10_12 5
1_10_14 6
1_10_16 7
1_10_18 8
…
8_9_08 7, 15, 3, 11, 8, 16, 4, 12
8_9_10 8, 16, 4, 12, 9, 1, 5, 13
8_9_12 9, 1, 5, 13, 10, 2, 6, 14
8_9_14 10, 2, 6, 14, 11, 3, 7, 15
8_9_16 11, 3, 7, 15, 12, 4, 8, 16
8_9_18 12, 4, 8, 16, 13, 5, 9, 1
8_9_20 13, 5, 9, 1, 14, 6, 10, 2
8_9_22 14, 6, 10, 2, 15, 7, 11, 3
Insert Operation Count: 270100
Search Operation Count: 0

*b) java LSAVLApp*

1_10_00 15
1_10_02 16
1_10_04 1
1_10_06 2
1_10_08 3
1_10_10 4
1_10_12 5
1_10_14 6
1_10_16 7
1_10_18 8
…
8_9_08 7, 15, 3, 11, 8, 16, 4, 12
8_9_10 8, 16, 4, 12, 9, 1, 5, 13
8_9_12 9, 1, 5, 13, 10, 2, 6, 14
8_9_14 10, 2, 6, 14, 11, 3, 7, 15
8_9_16 11, 3, 7, 15, 12, 4, 8, 16
8_9_18 12, 4, 8, 16, 13, 5, 9, 1
8_9_20 13, 5, 9, 1, 14, 6, 10, 2
8_9_22 14, 6, 10, 2, 15, 7, 11, 3
Insert Operation Count: 34927
Search Operation Count: 0

**Final Results for the comparison:**

a) Differences from above:

As briefly mentioned above, for my final comparison, I created additional testing apps to call the LSBSTApp and LSAVLApp repeatedly and output their results to csv files from which the data was summarized. Furthermore, I created an app to randomize the data in the input file in order to make the test results less predictable.

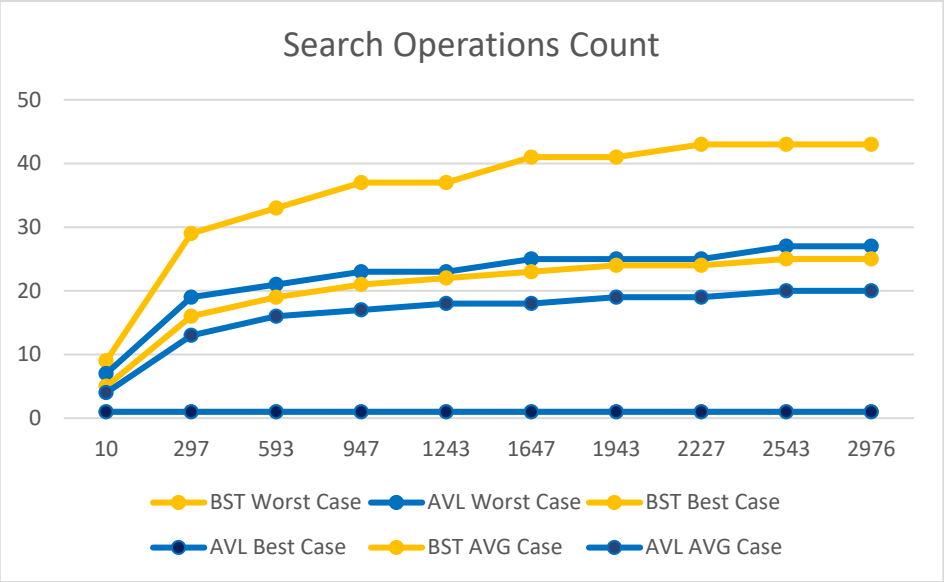
b) What the test does:

It calls a customizable number of lines, 10 times, from the randomized data file. Let's call the number of lines n.
For each value of n, the testing program calls both Apps to insert and search for every value in that subset of n lines.
(For example: if n=10, n lines of the file will be read, inserted into the different data structures, and then each of those 10 values will be searched for.)
Throughout this process, the number of operations for search and insertion are being calculated and sent to csv files.
These csv files are then read in by an app that summarizes their data to provide the Worst, Best and Average Case for each dataset and sent to a single csv file. This csv file is used to plot the graphs below.

**Summarised Data**

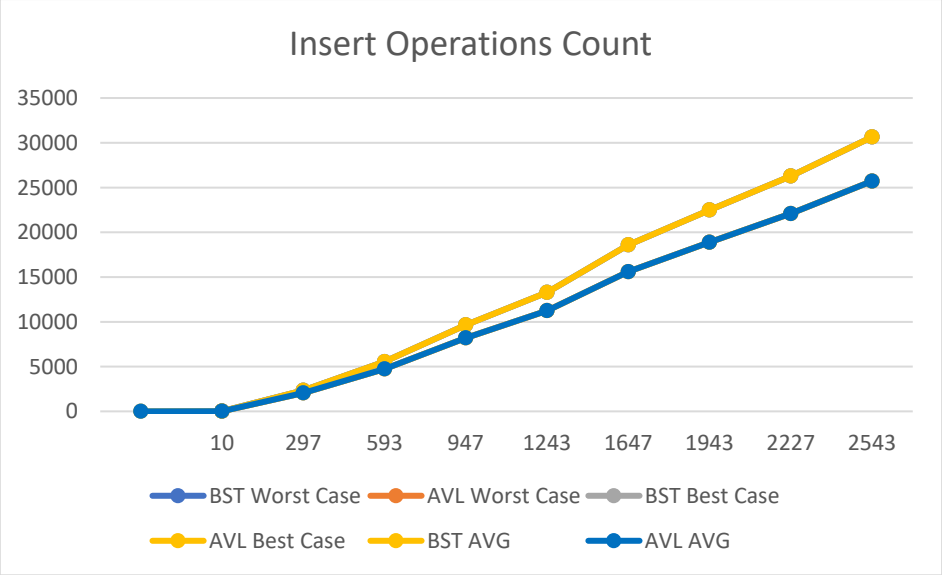
**Number of operations for search**

| DataSet length | BST Worst Case | AVL Worst Case | BST Best Case | AVL Best Case | BST AVG Case | AVL AVG Case |
|---|---|---|---|---|---|---|
| 10 | 9 | 7 | 1 | 1 | 5 | 4 |
| 297 | 29 | 19 | 1 | 1 | 16 | 13 |
| 593 | 33 | 21 | 1 | 1 | 19 | 16 |
| 947 | 37 | 23 | 1 | 1 | 21 | 17 |
| 1243 | 37 | 23 | 1 | 1 | 22 | 18 |
| 1647 | 41 | 25 | 1 | 1 | 23 | 18 |
| 1943 | 41 | 25 | 1 | 1 | 24 | 19 |
| 2227 | 43 | 25 | 1 | 1 | 24 | 19 |
| 2543 | 43 | 27 | 1 | 1 | 25 | 20 |
| 2976 | 43 | 27 | 1 | 1 | 25 | 20 |




**Number of operations for insert**

| DataSet length | BST Worst Case | AVL Worst Case | BST Best Case | AVL Best Case | BST AVG Case | AVL AVG Case |
|---|---|---|---|---|---|---|
| 10 | 22 | 21 | 22 | 21 | 22 | 21 |
| 297 | 2348 | 2058 | 2348 | 2058 | 2348 | 2058 |
| 593 | 5552 | 4743 | 5552 | 4743 | 5552 | 4743 |
| 947 | 9654 | 8206 | 9654 | 8206 | 9654 | 8206 |
| 1243 | 13287 | 11245 | 13287 | 11245 | 13287 | 11245 |
| 1647 | 18599 | 15602 | 18599 | 15602 | 18599 | 15602 |
| 1943 | 22489 | 18890 | 22489 | 18890 | 22489 | 18890 |
| 2227 | 26286 | 22090 | 26286 | 22090 | 26286 | 22090 |
| 2543 | 30649 | 25721 | 30649 | 25721 | 30649 | 25721 |
| 2976 | 36809 | 30780 | 36809 | 30780 | 36809 | 30780 |




**Comments on Data:**

x-axis: Length of Dataset(n)  y-axis: Number of comparison operations

**Search Count:**

- The Average and Worst Search Count graphs for both the AVL and regular BST seem to follow the shape of a log graph. However, the AVL seems to be more consistent and reliable since the Average Case and Worst Case, for each data set, aren't as far apart as the Average and worst case of the regular BST.
- Since, the expected graphs are also log graphs, there aren't any points that deviate significantly enough to be considered an outlier. If one was expecting a straight line, then the counts for the dataset length of 10 would be considered outliers, however their current position is in accordance with a log graph.
- For this dataset, the AVL tree is a tad more efficient in the average case and much more efficient in the worst case. In the best case, they are equal.
- The scale of the difference isn't as great in part 5 as it appeared it would be in *Part 2 & 4*. This is probably because the data inserted this time was randomized using the *RandomizeLSD* app. This randomization should have resulted in the tree being more evenly balanced in the regular Binary Search Tree, and thus closer to the AVL tree which is automatically balanced.

**Insert Count:**

- The graphs suggest that both the AVL and regular BST approximately follow a time complexity of n for their insertion algorithms.
- The Average, Best and Worst case for the AVL and regular Binary Search Tree line was the same since the algorithm was inserting the same randomized file for all the tests. Therefore, the number of times that the insert comparison counter was called remained constant for each data set.
- From this count, the AVL seems to be more efficient than the regular BST, however there are a few points to note:
    - The two seem to be really similar on the diagram, but this may be because the size of the graph had to be reduced for this report. If one looks at the values on the y-axis, a big difference can be seen in the number of operations for each dataset, and the difference is only increasing as the dataset grows larger.

- This experiment was done after the data was randomized using the *RandomizeLSD* app which could have resulted in the regular BST being more balanced out and a lot more similar in height to the AVL, during the insertion process, than it was in *Part 2 & 4*.
- The operations in the balance method of the AVL were not counted since they were not key comparisons and hence, arguably, not significant. However, if they were taken into account, the AVL may have been more inefficient than the regular BST.

**Case for creativity marks**

-Customizable:

- Added a feature that allows you to call the program with a different data-textfile and/or dataset length.
- Added a feature that allows one to repeat the experiment multiple times and output the useful data, with any dataset length

-Integration:

- Created extra Java apps that can integrate with other java classes should one wish to build on the current assignment spec, instead of running quick scripts in a foreign language: Created whole Java applications to randomize the input data, test the applications with several different parameters at once producing useful data and then to summarize the useful data without the need for excel.

**Git Log** (first and last 10 lines)

```
0: commit 62edb12858581d8770f726d22262a0c7819fcdd4
1: Author: Thaddeus Owl <61245401+ThaddeusOwl@users.noreply.github.com>
2: Date: Sat Apr 25 20:02:48 2020 +0200
3:
4: Completed javadocs :)
5:
6: commit 09f92fdfe978d4ff6fd96dca2a790c28c0d9f98c
7: Author: Thaddeus Owl <61245401+ThaddeusOwl@users.noreply.github.com>
8: Date: Sat Apr 25 18:55:47 2020 +0200
9:
...
79: Author: Thaddeus Owl <61245401+ThaddeusOwl@users.noreply.github.com>
80: Date: Fri Mar 27 10:14:39 2020 +0200
81:
82: Test git sync
83:
84: commit 89ee09a484c9f950239c750c609c7d88550b9b12
85: Author: Thaddeus Owl <61245401+ThaddeusOwl@users.noreply.github.com>
86: Date: Mon Mar 16 20:56:36 2020 +0200
87:
88: first commit
```