# PERFORMANCE ANALYSIS OF UNSORTED ARRAY AND BINARY SEARCH TREE

Gift Mugweni – MGWGIF001

APRIL 3, 2020

ASSIGNMENT 1

CSC2001F

# Performance Analysis of Unsorted Array and Binary Search Tree

## Abstract

This report details the performance analysis experiment carried out to test the performance of a Binary Search Tree data structure and an unsorted Array Data structure. Both data structures were implemented in java and were tested from an ever increasing data set. From the experiment, it was determined that that the unsorted array had an approximate linear behavior for the average and worst case scenarios and a constant time behavior for the best case scenario. For the Binary Search Tree, a logarithmic behavior was observed for the average and worst case scenario and a constant time behavior for the best case scenario.

## Introduction

During the CSC2001F lectures, we discussed the importance of creating algorithms with a better than $O(n)$ performance as they allow for faster operations for large data sets. The first data structure under investigation is the Binary Search Tree as it is relatively simple to understand. During the course of the experiment, all the sample dataset  values were obtained from the Eskom Load Shedding Schedule for the various stages. The total dataset had 2976 entries from which sub-datasets were made from. The report below details the creation and analysis of the respective data structures and the final results obtained.

## Data Structure Implementations

For the experiment, both data structures were implemented in a java programming environment. Two runner classes were created for the array and binary search tree data structures named LSArrayApp and LSBSTApp respectively. Aside from this, numerous sub classes were created since both data structures have many relationships within them. Figure 1 is a UML class diagram detailing the various relationships amongst the created classes for both data structures. For more details on the behavior of each method in the respective classes, please see the respective Javadoc file attached along with the report. It is important to note that for ease of access, command line parsers were added to the programs which is detailed more in the Javadoc for the respective runner classes. Finally, Git was also used for version Control as shown in Appendix 1.

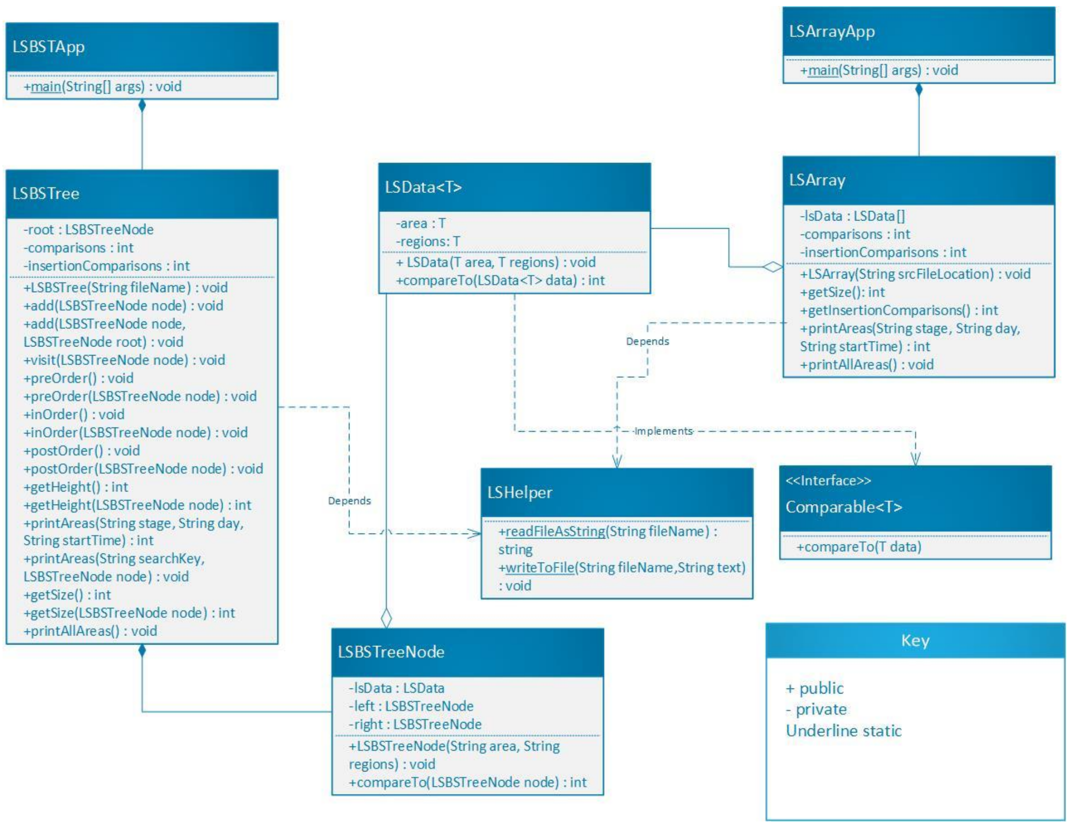
LSBSTApp
+main(String[] args) : void
LSBSTree
-root : LSBSTreeNode
-comparisons : int
-insertionComparisons : int
+LSBSTree(String fileName) : void
+add(LSBSTreeNode node) : void
+add(LSBSTreeNode node, LSBSTreeNode root) : void
+visit(LSBSTreeNode node) : void
+preOrder() : void
+preOrder(LSBSTreeNode node) : void
+inOrder() : void
+inOrder(LSBSTreeNode node) : void
+postOrder() : void
+postOrder(LSBSTreeNode node) : void
+getHeight() : int
+getHeight(LSBSTreeNode node) : int
+printAreas(String stage, String day, String startTime) : int
+printAreas(String searchKey, LSBSTreeNode node) : void
+getSize() : int
+getSize(LSBSTreeNode node) : int
+printAllAreas() : void
LSData<T>
-area : T
-regions: T
+ LSData(T area, T regions) : void
+compareTo(LSData<T> data) : int
LSArrayApp
+main(String[] args) : void
LSArray
-lsData : LSData[]
-comparisons : int
-insertionComparisons : int
+LSArray(String srcFileLocation) : void
+getSize(): int
+getInsertionComparisons() : int
+printAreas(String stage, String day, String startTime) : int
+printAllAreas() : void
Depends
Implements
Depends
LSHelper
+readFileAsString(String fileName) : string
+writeToFile(String fileName,String text) : void
<<Interface>>
Comparable<T>
+compareTo(T data)
LSBSTreeNode
-lsData : LSData
-left : LSBSTreeNode
-right : LSBSTreeNode
+LSBSTreeNode(String area, String regions) : void
+compareTo(LSBSTreeNode node) : int
Key
+ public
- private
Underline static

*Figure 1: UML Class Diagram for Created Program*

## Runner Class Testing

Upon creating the above classes, each runner class was tested with 3 test values that exist in the data set and 3 test values that are invalid. Table 1 below details the respective values used, the respective output and the comparisons carried out for each value. Note, the input provided are passed as command line arguments for the runner classes and the complete Load Shedding dataset was loaded into the respective data structures.

*Table 1: Test Data and Output of the Two Data Structures*

| | | Binary Search Tree | | Array | |
|---|---|---|---|---|---|
| | | Output | Comparisons | Output | Comparisons |
| Values existing in data set | 4 14 12 | 10, 2, 6, 14 | 148 | 10, 2, 6, 14 | 1330 |
| | 7 9 18 | 12, 4, 8, 16, 13, 5, 9 | 342 | 12, 4, 8, 16, 13, 5, 9 | 2561 |
| | 8 16 08 | 8, 16, 4, 12, 5, 13, 1, 9 | 372 | 8, 16, 4, 12, 5, 13, 1, 9 | 2872 |
| Invalid values | 20 14 12 | Areas not found | 47 | Areas not found | 2977 |
| | 2 100 12 | Areas not found | 47 | Areas not found | 2977 |
| | 2 14 -12 | Areas not found | 77 | Areas not found | 2977 |

Aside from the above, the programs all print all the elements of the dataset if no arguments are provided as seen in Table 2 which shows the first and last 10 entries of the dataset.

*Table 2: Sample output for printAllAreas() function*

| Binary Search Tree Output | | Array Output | |
|---|---|---|---|
| First 10 | Last 10 | First 10 | Last 10 |
| 1_1_00 1 | 8_9_04 5, 13, 1, 9, 6, 14, 2, 10 | 1_1_00 1 | 8_27_22 14, 6, 10, 2, 15, 7, 11, 3 |
| 1_17_00 1 | 8_9_06 6, 14, 2, 10, 7, 15, 3, 11 | 1_17_00 1 | 8_12_22 14, 6, 10, 2, 15, 7, 11, 3 |
| 1_10_00 15 | 8_9_08 7, 15, 3, 11, 8, 16, 4, 12 | 1_2_00 13 | 8_28_22 14, 6, 10, 2, 15, 7, 11, 3 |
| 1_11_00 11 | 8_9_10 8, 16, 4, 12, 9, 1, 5, 13 | 1_18_00 13 | 8_13_22 15, 7, 11, 3, 12, 4, 8, 16 |
| 1_10_02 16 | 8_9_12 9, 1, 5, 13, 10, 2, 6, 14 | 1_3_00 9 | 8_29_22 15, 7, 11, 3, 12, 4, 8, 16 |
| 1_10_04 1 | 8_9_14 10, 2, 6, 14, 11, 3, 7, 15 | 1_19_00 9 | 8_14_22 15, 7, 11, 3, 12, 4, 8, 16 |
| 1_10_06 2 | 8_9_16 11, 3, 7, 15, 12, 4, 8, 16 | 1_4_00 5 | 8_30_22 15, 7, 11, 3, 12, 4, 8, 16 |
| 1_10_08 3 | 8_9_18 12, 4, 8, 16, 13, 5, 9, 1 | 1_20_00 5 | 8_15_22 15, 7, 11, 3, 12, 4, 8, 16 |
| 1_10_10 4 | 8_9_20 13, 5, 9, 1, 14, 6, 10, 2 | 1_5_00 2 | 8_31_22 15, 7, 11, 3, 12, 4, 8, 16 |
| 1_10_12 5 | 8_9_22 14, 6, 10, 2, 15, 7, 11, 3 | 1_21_00 2 | 8_16_22 15, 7, 11, 3, 12, 4, 8, 16 |

## Data Structure Performance Experiment

To carry out the experiment, a python script was created to allow for automation of the data collection process. As mentioned above, the complete dataset has a total of 2976 entries. As such, the program was programmed to create 10 ever increasing sub-datasets which contain randomly selected entries. Upon creating each sub-dataset, each dataset is loaded into the runner classes. After which, each entry is queried in both runner classes. Upon carrying out the query, the total comparisons carried out is noted for each entry and stored in a temporary array. Upon storing all the comparisons carried out for the sub-dataset, the best case scenario is obtained by taking the minimum comparison value, the worst case is obtained by obtaining the maximum value and the average is obtained by finding the average comparisons for the sub-dataset values. Upon obtaining all three for each sub-dataset, the values are stored and the program moves on to the next sub-dataset field.

Furthermore, it is important to note that the comparisons for the insertions for the binary search tree were also carried out. Upon obtaining the respective data, Figure 2 and Figure 3 were obtained which compares the array and binary search tree comparison and the zoomed in binary search tree statistics respectively as shown below. Note a csv file containing the raw data is provided in the output folder is included to see the actual results obtained.

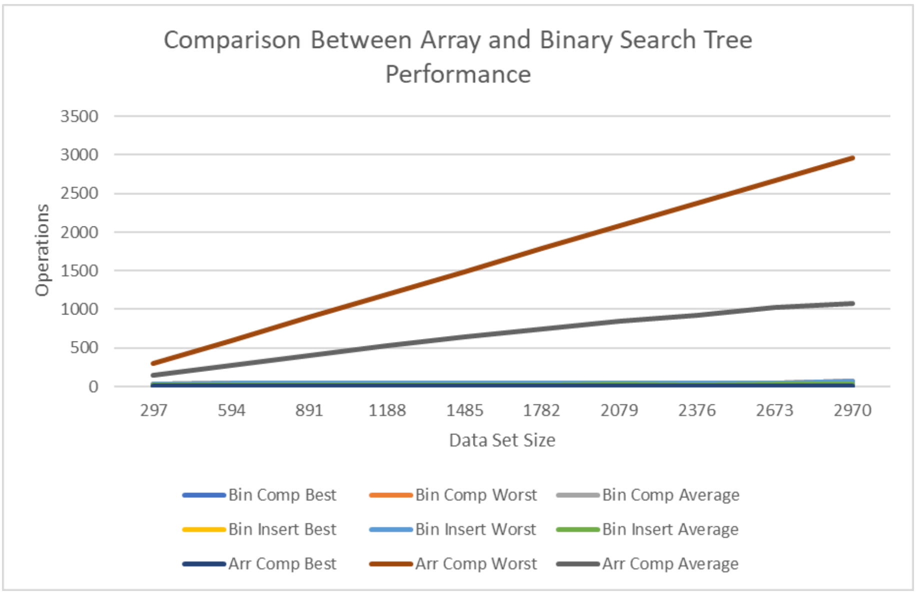

*Figure 2: Comparison between Array and Tree Data Structure*

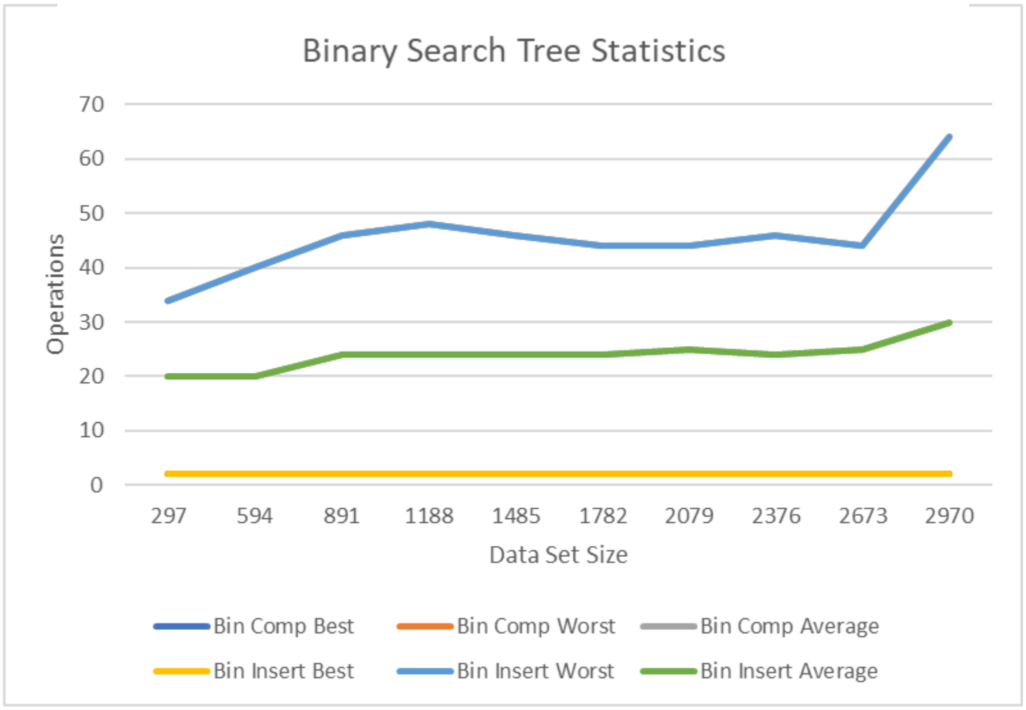

*Figure 3: Binary Search Tree Statistics*

## Discussion

From Figure 2 and Figure 3, it was noted that the array data structure has a linear behavior as the dataset increases in size whilst that of the binary search tree stays fairly constant and follows the logarithmic behavior. An important thing to note though is that upon checking the comparison operations of the randomly sampled sub-datasets, they appear significantly smaller than those obtained during the test value queries in Table 1. The reason behind this lies in the fact that most the provided load shedding file contained mostly sorted data and as such, the respective binary search tree was more linear as a result. On the other hand, the randomly sampled dataset allows for the nodes to be more spread out and as such, the resulting height is smaller on average and as such, the comparisons are smaller in general. Another important aspect to note was that the comparisons needed to insert a node are similar to the comparisons needed to find a node and as such, though slower than inserting in an array, the trade off is acceptable.

## Conclusion

With the above in mind, it was concluded that with regards to searching, a binary search tree is significantly more efficient than the unsorted array data structure.

## Appendix 1: Git Bash Log Output

```
$ git log | (ln=0; while read l; do echo $ln\: $l; ln=$((ln+1)); done) | (head -10;
echo ...; tail -10)
0: commit cd9e5760211a007e9e7e1b98772469694a9c6663
1: Author: Gift Mugweni <giftmugweni@gmail.com>
2: Date: Tue Mar 3 02:07:16 2020 +0200
3:
4: added javadoc file
5:
6: commit 8b89b0ff5948c9a488e33df2f0f519ea8ae40fb3
7: Author: Gift Mugweni <giftmugweni@gmail.com>
8: Date: Tue Mar 3 02:43:42 2020 +0200
9:
...
67: Author: Gift Mugweni <giftmugweni@gmail.com>
68: Date: Sun Feb 23 19:50:47 2020 +0200
69:
70: created array data structure and added requested methods
71:
72: commit 950d85c23267d09e1df806e43ff112ef7a7631cf
73: Author: Gift Mugweni <giftmugweni@gmail.com>
74: Date: Sat Feb 22 12:21:37 2020 +0200
75:
76: first commit
```