

CS3310

Fourth Assignment (a3)

Report

Brandon Rodriguez

11-14-17

# Preface

First off, a working, compiled jar of the assignment can be found in the root directory, called “a3.jar”. To run this, open up a terminal, navigate to the directory, and type “java -jar a3.jar”. The rest should be self-explanatory.

To find the source code of the assignment, navigate from the assignment’s root folder to “root/a3/src/edu/wmich/cs3310/a3”. A majority of the code is located within the controller.java class and the “DataStructures” subfolder.

Used libraries are included in the “root/a3/lib/” folder.

JavaDocs can be found at “root/Documents/JavaDocs/index.html”, as well as within the project dist folder.

Runtime output logs can be located at “root/Documents/RuntimeOutput/”.

Program was created using JetBrains’ “IntelliJ” IDE.

If not familiar with IntelliJ:

Should the jar file not execute, you should be able to load it into IntelliJ and get the same results. When first opening the project, use “Import Project” through IntelliJ’s splash screen. Then select “Create project from existing sources”.

If IntelliJ doesn’t automatically do it for you, make sure to set the “src” directory as the “Sources Root”, and the “tests” directory as the “Test Sources Root”. You may also need to load in junit, which should be provided within the lib folder.

You may also need to “Edit Configurations” and set Main as an application to launch from. But after that, it should work. Just run the program with the green arrow button and it should launch.

# Problem Statement

This program focuses on implementation of the following binary tree data structures:

- Standard Binary Tree (node-based)
  - Array-based Binary Tree
  - Binary Search Tree
  - Min Heap
  - Max Heap
- 

## Program Description

A file is read in at the start of program. This file is a list of last name, first name pairs separated by tabs. The user will be prompted to enter a first and last name to search for.

The program then builds three different trees based on this input file. Each one is correspondingly searched and various data values are output.

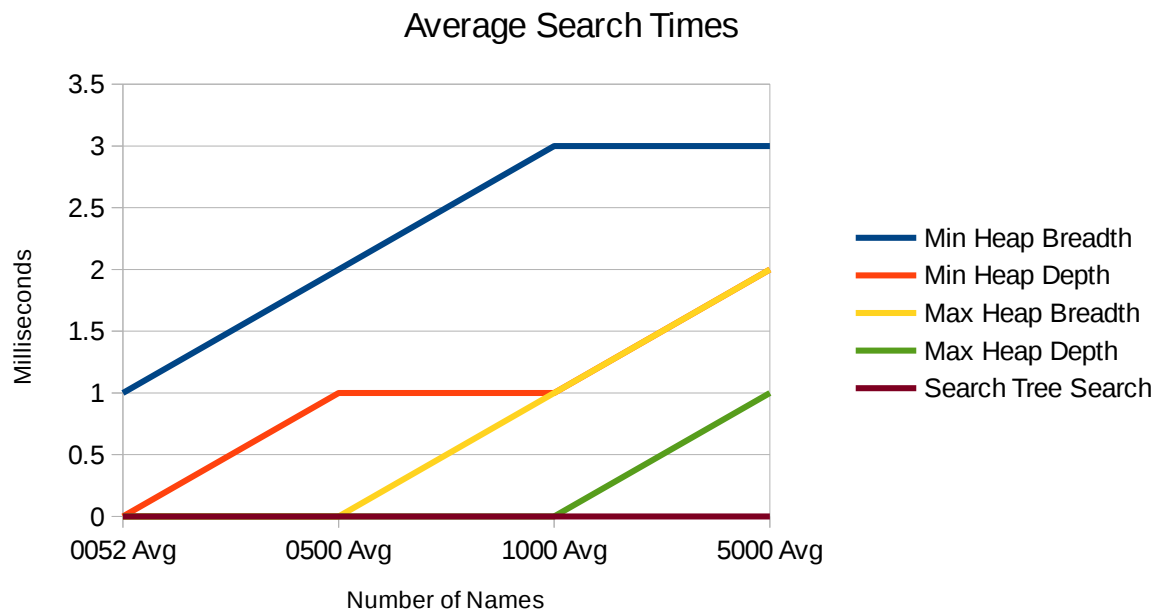
The two heaps each do a breadth-first search and then a depth-first search. The binary search tree just does a normal search tree search. At the professor's request, breadth-first searches are done entirely by queue linked lists. Depth-first searches are done entirely by stack linked lists.

Note that the depth-first search (and possibly breadth-first? Not sure on that) could be handled far better with a standard tree traversal, as this would only require a space of  $O(n)$ , as opposed to the current implementation that has a space of  $O(3n + \text{overhead requirements for two stacks})$ .

# Tables of Observed Time Complexities

Note: Format is in milliseconds.

	Full Tree Creation	Min Heap Breadth	Min Heap Depth	Max Heap Breadth	Max Heap Depth	Search Tree Search
0052 1	04	01	00	00	00	00
0052 2	04	01	00	00	00	00
0052 3	04	01	00	00	00	00
0500 1	19	02	01	00	00	00
0500 2	18	02	01	00	00	00
0500 3	20	02	00	00	01	00
1000 1	26	03	01	00	01	00
1000 2	26	03	01	02	00	00
1000 3	27	02	02	01	00	00
5000 1	56	03	02	02	02	00
5000 2	53	04	02	02	01	00
5000 3	53	03	02	03	01	00



	Min Heap Breadth	Min Heap Depth	Max Heap Breadth	Max Heap Depth	Search Tree Search
<b>0052 Avg</b>	01	00	00	00	00
<b>0500 Avg</b>	02	01	00	00	00
<b>1000 Avg</b>	03	01	01	00	00
<b>5000 Avg</b>	03	02	02	01	00

## Theoretical Expected Times

	Node-Based Min Heap	Array-Based Max Heap	Binary Search Tree
<b>Node Insertion</b>	$O(h)$ which should equate to $O(\log n)$ .	$O(h)$ which should equate to $O(\log n)$ .	$O(h)$ which may be as bad as $O(n)$ , depending on data.
<b>Organize/Heapify</b>	$O(h)$	$O(h)$	$O(1)$ (done via insertion)
<b>Traverse Tree Height</b>	$O(h)$	$O(h)$	$O(h)$
<b>Full Traversal</b>	$O(n)$	$O(n)$	$O(n)$
<b>Standard Search</b>	---	---	$O(h)$
<b>Breadth-First Search</b>	$O(n)$	$O(n)$	--
<b>Depth-First Search</b>	$O(n)$	$O(n)$	--
<b>Space</b>	$O(n)$	$O(n)$	$O(n)$
<b>Space When Searching</b>	$O(3n)$	$O(3n)$	$O(n)$

# Concluding Analysis

## Time Complexities

Overall, it appears that the depth-first searches were superior to breadth-first searches. Due to a small sample size, it's possible that it's due to luck, and I just happened to pick nodes that favored depth-first searching.

The array-based implementation also seemed to perform far better than the node-based implementation. I'm not sure if trees will always behave this way or if I might have had subpar code for the node-based tree, due to being my first attempt at writing a tree structure. At least theoretically, I can think of no reason why the implicit would perform better, as any point where the explicit needs to traverse the tree, the implicit should theoretically be visiting the same number of nodes.

This is further muddled by the fact that one was a min-heap and one was a max-heap, so it's also possible that the nodes I picked simply favored a max-heap over a min-heap.

Regardless, the binary search tree blew both of them out of the water with the current data sets used. Even with larger datasets, it is unlikely that the search tree will perform worse, as it would be incredibly rare (even if still possible) that the tree ends up with a height of  $n$ . Obviously, this can be resolved if the search tree were transformed into a height-balanced search tree.

With the current problem description, it actually makes no sense to ever use a heap of any kind over a search tree. At no point is a node ever removed/deleted, which would be when heaps shine.

---

## Space Complexities

Upon attempting to read in 10,000 names or more, the program would crash and I'd get a `stackOverflow` error. This is almost certainly due to the requested implementation of the depth/breadth searches. In particular, the depth-first search is extremely space-inefficient, as in reality, it should be a simple tree traversal.

Instead, it uses two separate stacks for the search. The first stack holds all tree nodes, in proper order, until they can be used in comparison for the actual search. The second stack is used exclusively to keep track of which nodes still need their children added to the first stack.

I suppose this could be improved somewhat, by iterating through the first stack as it's pushed to, thus keeping it's size closer to  $O(1)$  than  $O(2)$ . But even then, the stacks should just be put away and traversal used instead.

Similar logic was used for the breadth-first search, if only for consistency.