University of Cape Town

CSC2001F

Assignment 2

AVL Trees

Electronic Telephone Directory

BSSDIN001

Dino Bossi

The Problem

The problem needing to be solved was similar to last week’s, however now using an AVL tree as the internal data structure.

We were required to create an Electronic Telephone Directory java application, using an AVL tree as the internal data structure. The AVL tree was to be populated with data from a supplied data file (testdata). Each item in the data file contains an address followed by a telephone number and finally a name. The data from the datafile was separated with a | character. The AVL tree nodes were to hold the address and telephone number information, while using the name as the key to be compared.

The AVL tree was required to have numerous actions performed on it (using methods), more specifically methods for deletion and insertion of nodes, as well as a method to search for a specific node (via the name-key). The data inserted into the AVL tree was required to be compared using the key. Since the key is a string an adequate method to compare keys was needed.

Searching of the AVL tree was to be done using a query file. The query file was to contain queries relating to elements from the original dataset. Each query is a name and thus searching for it in the tree had to be done using the specified query-name.

An additional application, the already created SearchIt, was to be used in a comparison experiment. The aim of the experiment is to compare the insertion, deletion and searching times of the SearchAVL (implementing an AVL tree) and SearchIt (implementing a Binary Search Tree) applications.

Design

The source code for the Binary Search Tree and Node classes implemented in SearchIt were not my own, but were modified to handle the supplied phone directory data.

The source code for the AVLTree and other subsequent code were not my own, but modified to increase ease of usability and to handle the supplied phone directory data.

SearchAVL

Static methods were used for the three necessary tasks completed in SearchAVL. The three tasks were:

1. Initialising the AVL tree and inserting the testdata data into the tree.
2. Searching for the queries specified in the query file and printing the found results to the screen.
3. Deleting specified elements from the tree

The use of static methods allowed for the code to be completely self-contained and the methods which executed the above tasks to be re-used if needed. Since the code needed was minimal and the tasks required relatively simple it became more practical to just use three static methods, all finally being called in a main method. Since there is no object-oriented connection between the application and the tasks it was required to achieve, a ‘SearchAVL’ object and thus the need for non-static methods was decided against.

The SearchAVL application is comprised of three static methods and a main method. The static methods handle the loading of the testdata file, initialising the AVL tree, inserting each entry (a line in testdata) into the tree, searching for nodes specified in a query file and deleting of nodes specified in the query file. The deleting of nodes was solely for the purpose of experimentation.

The static methods were as follows:

1. loadData()
2. search()
3. deletes()

i) The static method loadData() consists of code to load the file testdata into an AVL Tree. An AVL Tree object is created and initialised. A scanner object is used to load the testdata file and subsequently read each line from testdata, one at a time. The code is placed within a try and catch statement in order to catch any exceptions which may occur when attempting to read testdata. Each line from testdata represents a telephone directory entry. A node is added using the line from testdata, using the line as the parameter. A while loop is used to create a node from each line from testdata and add it to the AVL tree. The method returns the populated AVL Tree, which contains all the entries from the testdata file.

ii) The static method search() is used to load in the query file and search for each query in the AVL Tree which has been initialised by calling the loadData() method. A scanner object is initialized in order to read the queries from the query file. A while loop is used to loop through each line of the query file, and thus each query. The query read from the file is used as the argument to search through the AVL Tree. Each query is full name and is compared to the key of each node in order to find the entry associated with the specified full name (query). If the query is found the full entry is printed to the screen, if it is not found a message “Not Found” is printed instead. The not found message is called if the search yields a null result, meaning that the node does not exist in the tree. The code is encapsulated by a try-and-catch statement in order to catch any errors that may occur.

iii) The static method deletes() is used to delete elements from the tree, and while this is not a requirement of the application its functionality was used in an experiment to compare relative operation speeds of an AVL Tree and a Binary Search Tree. The method aims to delete nodes from the tree with keys equal to the queries from queries.txt (the same queries search() searched for). A scanner object is initialised in order to read in the queries.txt file. A while loop is used to loop through each query in the query file. Each query is used as an argument for the delete() method, which searches for the specified node in the AVL Tree and deletes it if it is found. The code is surrounded by a try-and-catch statement in order to prevent any encountered errors from crashing the application.

DirectoryNode

A directory node class was made to store the data for each node in the AVL Tree. This included the key which was set to the full name of the entry and the value which was set to the full entry (address|telephone number|full name). The class contains methods to obtain the value and key attributes of a DirectoryNode object. A toString() method was also included so that it could be used to print the entry attribute of the object with ease.

Query File Queries

Below are the contents of a 20-line query file.

Paucek Sabrina  
Smitham Jaycee  
Mayert Kelton  
Mayert Aditya  
Macejkovic Antonietta  
Ullrich Bertha  
Williamson Dustin  
Weber Ashton  
Adams Newton  
Stokes Waino  
McCullough Hailey  
Beatty Nigel  
Weissnat Milan  
Kuhlman Angelica  
Quitzon Catherine  
Bechtelar Vallie  
Kirlin Noel  
Ledner Maximo  
Kuphal Kacey  
Mueller Regan

SearchAVL Output.

Below is the output of the SearchAVL application when run with the above specified query file.

79830 Meda Pass #363, Bell Gardens|1-016-609-7836|Paucek Sabrina  
67828 Aleen Fort Apt. 392, Demopolis|[1-469-508-0160](tel:(469)%20508-0160)|Smitham Jaycee  
99508 Mayert Courts, Homer|296-910-6876 x374|Mayert Kelton  
23235 Ilene Street, Birmingham|122.008.8997 x6636|Mayert Aditya  
15916 Micaela Dale #868, La Mirada|(462)809-4482 x2731|Macejkovic Antonietta  
11269 Hauck Mountain #623, Signal Hill|[(630)593-2008 x86864](tel:(630)%20593-2008)|Ullrich Bertha  
31319 Stehr Forest, Escondido|(333)283-8009 x71057|Williamson Dustin  
02762 Seth Spring Suite 301, San Gabriel|[(657)209-4288 x6284](tel:(657)%20209-4288)|Weber Ashton  
94873 Floor 589, Covina|(001)291-8661 x72578|Adams Newton  
48000 Laron Ridge, Valparaiso|836.405.1738|Stokes Waino  
06795 Marvin Harbors Apt. 167, Half Moon Bay|[864.222.5386](tel:(864)%20222-5386)|McCullough Hailey  
53484 Koepp Radial #414, Louisville|[231-936-3585](tel:(231)%20936-3585)|Beatty Nigel  
92832 Bednar Shore, Texarkana|[445-936-2637 x26051](tel:(445)%20936-2637)|Weissnat Milan  
24815 Brown Hollow Suite 040, Costa Mesa|[(484)918-1264 x18732](tel:(484)%20918-1264)|Kuhlman Angelica  
30446 Durgan Glens, Chandler|(334)041-7832|Quitzon Catherine  
34929 Bartell Rapids, Huntington Beach|1-086-530-5520|Bechtelar Vallie  
43872 Jenkins Lodge Suite 257, Sierra Vista|(082)816-5346|Kirlin Noel  
77475 Hellen River #011, Carson|194-749-6722 x71755|Ledner Maximo  
36330 Hangar 97-F, Lakewood|[330-968-5301 x4048](tel:(330)%20968-5301)|Kuphal Kacey  
30694 Space 6-J, Agoura Hills|[606.733.6187](tel:(606)%20733-6187)|Mueller Regan

Comparison Experiment

The time taken to complete a search, delete and insert operation using an AVL Tree and a Binary Search Tree should be compared to accurately identify the ‘faster’ data structure. It is known that an AVL Tree search, insert and delete operations have an O(log(n)) algorithm analysis, in the worst case. The Binary Search Tree search, insert and delete operations have an O(log(n)) algorithm analysis, in the best case. Thus the condition of the Binary Search Tree object (i.e. how balanced it is, the order in which data was inserted etc.) is influential to the speed of the operations performed on it, while an AVL Tree is managed in such a way as to rectify the same factors which inhibit a Binary Search Tree’s operation performance.

Experimental Design:

A separate application QueryFileMaker is to be used to create the appropriate query and data files. The application creates a data file (with n telephone directory entries) which is a subset of the testdata file. It also creates a query file with n/2 (except when n=1, then number of queries = 1) elements taken from the newly created data file. The query file does not contain any duplicates and the elements populating the data and query files are chosen at random in both cases.

The varying query file size allows for adequate and accurate time keeping, since operations using a small number of queries compared to the data execute too quickly. Since the ratio of data to queries is always the same for each value of n and each application executes the operations on the same data and query files, the experimental tests remain fair.

Both data structures load in the data (of length n) from the data file as well as the queries (of length n/2) from the query file. The values of n used for the experiment can be found below. Since each created data file has a subsequent query file a script was used to run the applications (SearchAVL and SearchIt) with the specific length (n and n/2 respectively) data and query files, outputting the time recordings to a .txt file for easy access and observation.

The insert operation is recorded as the insertion of data from the data file into the data structure. The search operation is considered to be the searching of queries (from the query file) in the data structure. The delete operation uses the query file again, however this time deleting the data structure nodes whose key matched that of a query from the query file. This allowed the re-use of the query file as well as maintaining equality in the tests between the BST and AVL Tree.

Each application was run 5 times for each value of n to ensure a fair test as well as reduce the effect of unexpected occurrences (e.g. operating system slowing down execution due to OS processes taking place, other programs potentially slowing down the execution, caching speeding up application execution etc.). The average of the 5 test times (for each value of n) will be determined. Upon completion of the time recordings the averages for the various operations must be compared in order to determine which data structure has the faster operations for varying values of n.

It is important to note that since both classes use the scanner class and subsequent objects in the same places, it will not influence the relative times but will however decrease the speed at which operations are executed.

Expectations and Recordings:

We would expect the AVL Tree to have a faster search operation, while inserting and deleting could take longer due to the need to invoke balancing methods (single and double rotations). However, it is not to be expected for all values of n (and subsequently, different data sets) since the specific data inserted into the tree would influence operation performance.

Below are the tabulated results for the search, insert and delete operations for both the SearchIt (Binary Search Tree data structure) and SearchAVL (AVL Tree data structure).

Values chosen for n: 1, 5, 10, 25, 50, 100, 250, 500, 1000, 1500, 2000, 3000, 4000, 5000, 5500, 6000, 7000, 8000, 9000, 9500, 10000

Values for time are given in milliseconds.

SearchIt Insert

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **Time 1** | **Time 2** | **Time 3** | **Time 4** | **Time 5** | **Average** |
| **1** | 24 | 53 | 23 | 51 | 42 | 38.6 |
| **5** | 25 | 28 | 57 | 50 | 66 | 45.2 |
| **10** | 59 | 45 | 72 | 66 | 34 | 55.2 |
| **25** | 54 | 26 | 53 | 60 | 50 | 48.6 |
| **50** | 71 | 29 | 55 | 49 | 47 | 50.2 |
| **100** | 89 | 39 | 52 | 74 | 107 | 72.2 |
| **250** | 41 | 44 | 46 | 77 | 40 | 49.6 |
| **500** | 57 | 47 | 58 | 72 | 55 | 57.8 |
| **1000** | 175 | 56 | 52 | 51 | 76 | 82 |
| **1500** | 96 | 93 | 146 | 103 | 77 | 103 |
| **2000** | 130 | 146 | 120 | 117 | 92 | 121 |
| **3000** | 162 | 192 | 118 | 98 | 164 | 146.8 |
| **4000** | 238 | 271 | 218 | 183 | 160 | 214 |
| **5000** | 229 | 263 | 216 | 209 | 267 | 236.8 |
| **5500** | 283 | 255 | 228 | 210 | 247 | 244.6 |
| **6000** | 293 | 284 | 322 | 242 | 221 | 272.4 |
| **7000** | 294 | 215 | 243 | 291 | 246 | 257.8 |
| **8000** | 340 | 345 | 318 | 316 | 274 | 318.6 |
| **9000** | 292 | 261 | 276 | 317 | 406 | 310.4 |
| **9500** | 352 | 279 | 263 | 282 | 235 | 282.2 |
| **10000** | 268 | 300 | 290 | 304 | 302 | 292.8 |

SearchIt Search

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **Time 1** | **Time 2** | **Time 3** | **Time 4** | **Time 5** | **Average** |
| **1** | 1 | 0 | 0 | 0 | 1 | 0.4 |
| **5** | 0 | 1 | 1 | 1 | 1 | 0.8 |
| **10** | 1 | 0 | 0 | 0 | 6 | 1.4 |
| **25** | 1 | 1 | 1 | 1 | 1 | 1 |
| **50** | 8 | 2 | 3 | 2 | 1 | 3.2 |
| **100** | 8 | 4 | 1 | 4 | 3 | 4 |
| **250** | 6 | 6 | 5 | 7 | 6 | 6 |
| **500** | 16 | 5 | 5 | 15 | 3 | 8.8 |
| **1000** | 2 | 6 | 5 | 19 | 3 | 7 |
| **1500** | 3 | 12 | 8 | 3 | 3 | 5.8 |
| **2000** | 3 | 7 | 4 | 9 | 12 | 7 |
| **3000** | 10 | 10 | 10 | 22 | 15 | 13.4 |
| **4000** | 15 | 33 | 36 | 17 | 29 | 26 |
| **5000** | 34 | 29 | 18 | 24 | 28 | 26.6 |
| **5500** | 24 | 19 | 17 | 41 | 21 | 24.4 |
| **6000** | 27 | 39 | 37 | 35 | 18 | 31.2 |
| **7000** | 25 | 23 | 29 | 28 | 24 | 25.8 |
| **8000** | 35 | 37 | 28 | 48 | 48 | 39.2 |
| **9000** | 39 | 33 | 34 | 39 | 47 | 38.4 |
| **9500** | 36 | 39 | 53 | 44 | 44 | 43.2 |
| **10000** | 42 | 70 | 44 | 78 | 43 | 55.4 |

SearchIt Delete

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **Time 1** | **Time 2** | **Time 3** | **Time 4** | **Time 5** | **Average** |
| **1** | 1 | 0 | 0 | 0 | 1 | 0.4 |
| **5** | 4 | 1 | 1 | 1 | 1 | 1.6 |
| **10** | 0 | 0 | 0 | 1 | 1 | 0.4 |
| **25** | 2 | 1 | 2 | 2 | 1 | 1.6 |
| **50** | 1 | 1 | 0 | 1 | 0 | 0.6 |
| **100** | 4 | 1 | 1 | 2 | 1 | 1.8 |
| **250** | 1 | 1 | 3 | 7 | 1 | 2.6 |
| **500** | 1 | 9 | 22 | 10 | 1 | 8.6 |
| **1000** | 3 | 12 | 17 | 16 | 7 | 11 |
| **1500** | 13 | 8 | 22 | 28 | 30 | 20.2 |
| **2000** | 28 | 27 | 8 | 16 | 11 | 18 |
| **3000** | 19 | 19 | 8 | 15 | 12 | 14.6 |
| **4000** | 22 | 20 | 21 | 24 | 21 | 21.6 |
| **5000** | 31 | 19 | 22 | 21 | 25 | 23.6 |
| **5500** | 33 | 35 | 27 | 44 | 31 | 34 |
| **6000** | 32 | 50 | 31 | 55 | 26 | 38.8 |
| **7000** | 45 | 29 | 41 | 48 | 38 | 40.2 |
| **8000** | 36 | 42 | 33 | 57 | 51 | 43.8 |
| **9000** | 64 | 70 | 34 | 33 | 42 | 48.6 |
| **9500** | 51 | 47 | 53 | 61 | 38 | 50.2 |
| **10000** | 58 | 37 | 93 | 42 | 55 | 57 |

SearchAVL Insert

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **Time 1** | **Time 2** | **Time 3** | **Time 4** | **Time 5** | **Average** |
| **1** | 35 | 28 | 23 | 33 | 24 | 28.6 |
| **5** | 26 | 28 | 46 | 24 | 38 | 32.4 |
| **10** | 29 | 60 | 48 | 50 | 59 | 49.2 |
| **25** | 28 | 28 | 37 | 27 | 47 | 33.4 |
| **50** | 30 | 51 | 45 | 63 | 31 | 44 |
| **100** | 64 | 70 | 72 | 80 | 37 | 64.6 |
| **250** | 97 | 114 | 79 | 62 | 55 | 81.4 |
| **500** | 112 | 61 | 52 | 49 | 45 | 63.8 |
| **1000** | 55 | 136 | 88 | 64 | 68 | 82.2 |
| **1500** | 122 | 76 | 74 | 108 | 123 | 100.6 |
| **2000** | 120 | 90 | 87 | 147 | 118 | 112.4 |
| **3000** | 110 | 103 | 137 | 131 | 253 | 146.8 |
| **4000** | 192 | 200 | 191 | 245 | 194 | 204.4 |
| **5000** | 212 | 233 | 272 | 193 | 200 | 222 |
| **5500** | 231 | 347 | 276 | 216 | 243 | 262.6 |
| **6000** | 257 | 284 | 292 | 237 | 212 | 256.4 |
| **7000** | 264 | 276 | 290 | 227 | 245 | 260.4 |
| **8000** | 265 | 299 | 233 | 288 | 293 | 275.6 |
| **9000** | 252 | 293 | 360 | 329 | 281 | 303 |
| **9500** | 243 | 263 | 305 | 328 | 290 | 285.8 |
| **10000** | 285 | 278 | 311 | 253 | 271 | 279.6 |

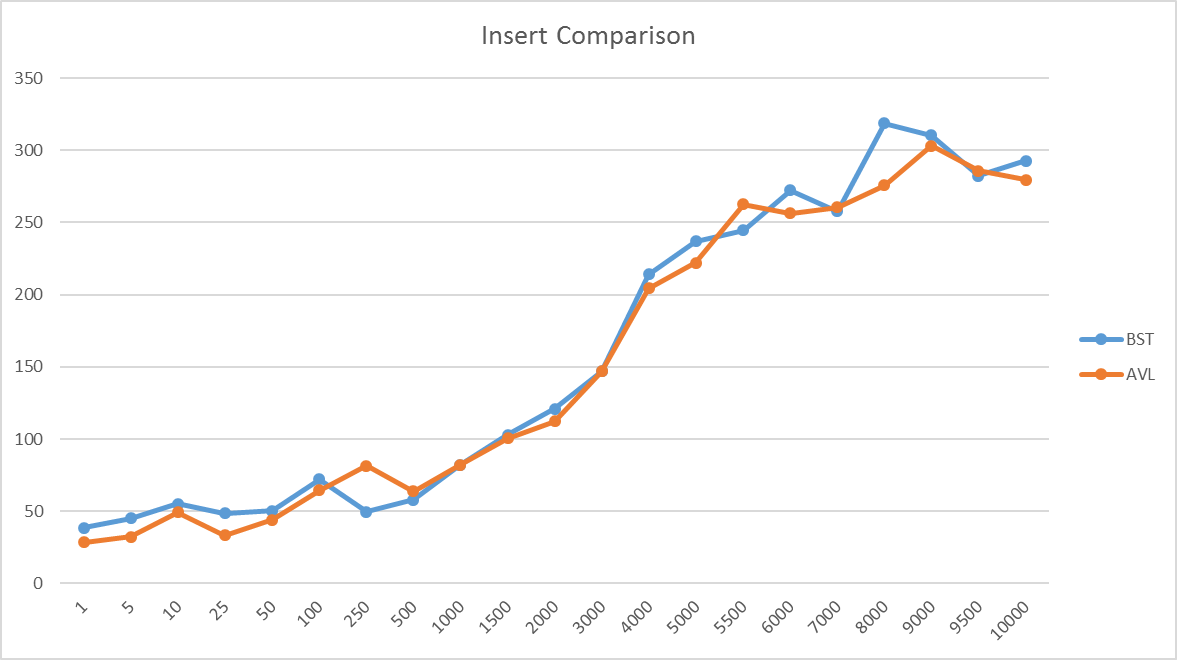
SearchAVL Search

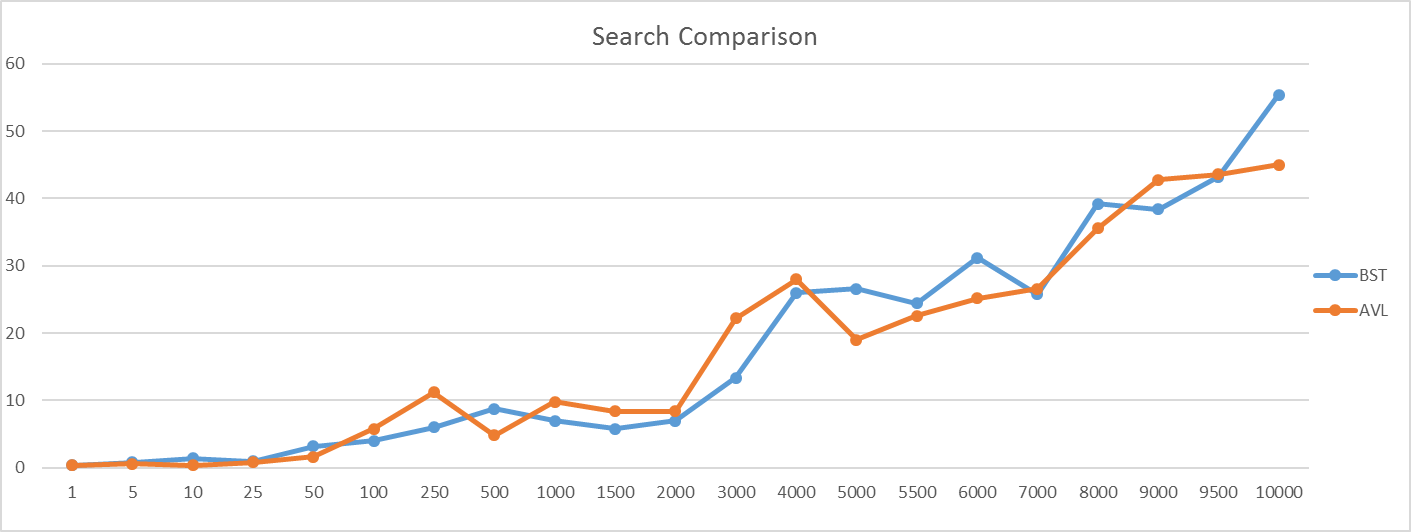
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **Time 1** | **Time 2** | **Time 3** | **Time 4** | **Time 5** | **Average** |
| **1** | 1 | 1 | 0 | 0 | 0 | 0.4 |
| **5** | 1 | 1 | 0 | 0 | 1 | 0.6 |
| **10** | 1 | 1 | 0 | 0 | 0 | 0.4 |
| **25** | 0 | 1 | 1 | 1 | 1 | 0.8 |
| **50** | 2 | 2 | 1 | 2 | 1 | 1.6 |
| **100** | 10 | 3 | 4 | 8 | 4 | 5.8 |
| **250** | 14 | 12 | 7 | 16 | 7 | 11.2 |
| **500** | 6 | 5 | 4 | 4 | 5 | 4.8 |
| **1000** | 6 | 3 | 26 | 12 | 2 | 9.8 |
| **1500** | 15 | 7 | 6 | 3 | 11 | 8.4 |
| **2000** | 7 | 15 | 9 | 8 | 3 | 8.4 |
| **3000** | 18 | 22 | 28 | 14 | 29 | 22.2 |
| **4000** | 38 | 17 | 21 | 44 | 20 | 28 |
| **5000** | 21 | 20 | 19 | 21 | 14 | 19 |
| **5500** | 16 | 24 | 30 | 21 | 22 | 22.6 |
| **6000** | 18 | 35 | 36 | 23 | 14 | 25.2 |
| **7000** | 30 | 32 | 24 | 21 | 26 | 26.6 |
| **8000** | 31 | 44 | 26 | 41 | 36 | 35.6 |
| **9000** | 60 | 39 | 33 | 45 | 37 | 42.8 |
| **9500** | 39 | 74 | 34 | 36 | 35 | 43.6 |
| **10000** | 44 | 56 | 41 | 43 | 41 | 45 |

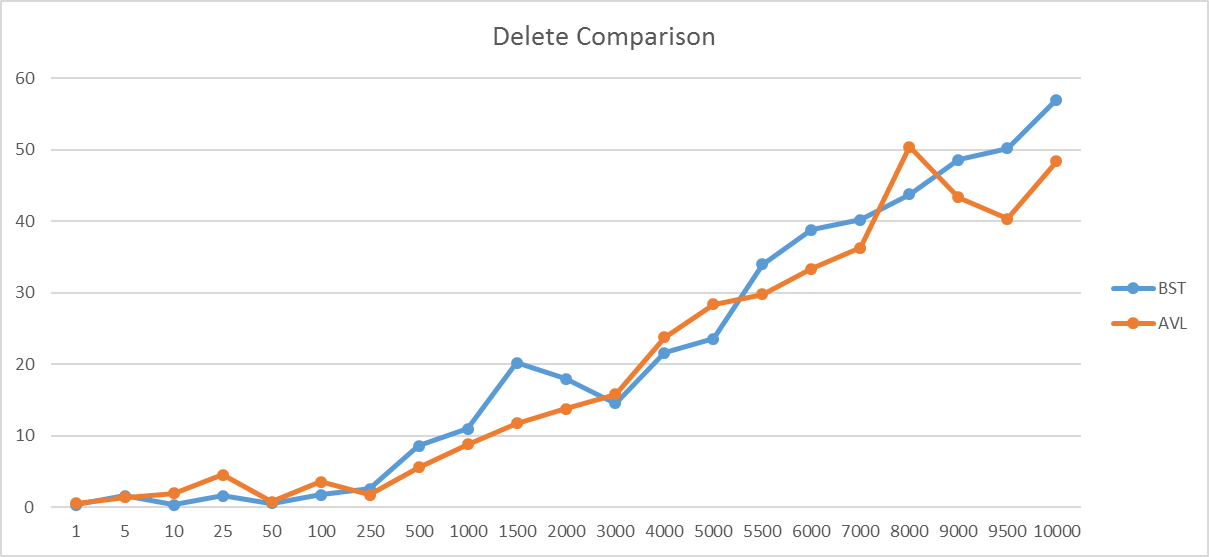
SearchAVL Delete

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **Time 1** | **Time 2** | **Time 3** | **Time 4** | **Time 5** | **Average** |
| **1** | 1 | 1 | 0 | 1 | 0 | 0.6 |
| **5** | 0 | 1 | 0 | 5 | 1 | 1.4 |
| **10** | 1 | 1 | 0 | 5 | 3 | 2 |
| **25** | 1 | 8 | 1 | 12 | 1 | 4.6 |
| **50** | 1 | 0 | 0 | 2 | 1 | 0.8 |
| **100** | 6 | 1 | 4 | 5 | 2 | 3.6 |
| **250** | 0 | 1 | 1 | 1 | 6 | 1.8 |
| **500** | 1 | 2 | 9 | 5 | 11 | 5.6 |
| **1000** | 10 | 3 | 14 | 2 | 15 | 8.8 |
| **1500** | 10 | 7 | 29 | 8 | 5 | 11.8 |
| **2000** | 12 | 19 | 22 | 6 | 10 | 13.8 |
| **3000** | 11 | 12 | 15 | 26 | 15 | 15.8 |
| **4000** | 18 | 19 | 24 | 29 | 29 | 23.8 |
| **5000** | 28 | 21 | 49 | 21 | 23 | 28.4 |
| **5500** | 44 | 26 | 23 | 29 | 27 | 29.8 |
| **6000** | 41 | 31 | 29 | 33 | 33 | 33.4 |
| **7000** | 29 | 24 | 25 | 69 | 36 | 36.3 |
| **8000** | 36 | 76 | 66 | 35 | 39 | 50.4 |
| **9000** | 75 | 39 | 38 | 28 | 37 | 43.4 |
| **9500** | 51 | 37 | 42 | 38 | 34 | 40.4 |
| **10000** | 68 | 36 | 45 | 42 | 51 | 48.4 |

Three graphs representing the average time versus the value of n for each operation can be found below. Both data structure’s results are plotted on the same graph to allow for comparison between the two sets. SearchIt results are in plotted in blue while the SearchAVL results are plotted in orange.







Discussion and Conclusion:

It is to be noted that both the results for SearchAVL and SearchIt contained outliers (e.g. delete comparison n=9000). However, the results obtained can still be used to show that for almost all values of n the speed of operation execution is relatively similar for both data structures, as expected.

Unfortunately the recorded data is not as conclusive as expected, since the faster data structure for each operation often changes as the value of n changes. This can be attributed to the data being inserted into the trees, but there is still reason to believe that the recorded data is slightly inaccurate.

A possible explanation for the extremely varying time recordings of the AVL Tree operations is the extra operations it is required to make in order to keep the tree balanced (according to the AVL Balance Properties).

Interestingly the results obtained for the largest data size (n = 10000) yielded similar results, with all of the AVL Tree operations executing faster than those of the Binary Search Tree. This is evidence for the fact that the AVL Tree is indeed the faster data structure in the worst case.

While the overall experiment can not be used to draw a conclusion favouring either data structure, there is strong evidence which shows that both structures’ time efficiency is determined by the data to be inserted into them. The data does show however, that both data structures have similar operation execution times on average.

The table below shows which data structure had the faster operations for each value of n (i.e. whether the BST was faster than AVL and how many times in the recordings it took place).

|  |  |  |
| --- | --- | --- |
| Operation | AVL Tree | Binary Search Tree |
| Insert | Faster on 14 recordings | Faster on 7 recordings |
| Delete | Faster on 10 recordings | Faster on 11 recordings |
| Search | Faster on 13 recordings | Faster on 8 recordings |

Thus from the table above we can see that the AVL Tree was faster than the BST on average for the insert and search operations. For the delete operation however the BST was on average faster than the AVL Tree. From this comparison, we can draw the conclusion that on average the AVL Tree was faster than the BST. However, it is important to note that the specific data used does influence the speed of operation execution and the above results might not always be obtained if the experiment be repeated.

Summary Statistics

JUnit Testing:

Two Junit test classes were created to test the SearchAVL application as well as the DirectoryNode class. Both of the test classes were created in a way that uses a pre-created version of the testdata data file and query file. This allowed for testing of functionality rather than ensuring that specific output is correct (this means that if the tests are passed the applications can be expected to run successfully regardless of the contents of testdata and queries.txt). Below are the tests (a, b, c, etc.) per application:

1. TestSearchAVL
   1. Test to ensure that loadData() method loaded all the data successfully and inserted it into the AVL Tree correctly.
   2. Test to ensure that search() method loads the query file correctly and subsequently searches for each query in the Tree.
   3. Test to ensure that deletes() method loads the query file correctly and subsequently deletes the specified elements from the tree. The functionality was tested by printing the tree after the deletions had taken place.
2. TestDirectoryNode
   1. Test to ensure that objects can be successfully created.
   2. Test to ensure that entry and key attributes get set correctly when the argument for object initialisation is a full entry (address|telephone number|full name).
   3. Test to ensure that getKey() method operates as expected and returns a full name
   4. Test to ensure that getValue() method operates as expected and returns the full telephone directory entry in the correct format (address|telephone number|full name).

Please note that the tests were constructed in a way that requires the test .class files to be present in the test folder, rather than in the bin folder. Tests were not created to test the AVLTree source code, since they were created and tested by Hussein Suleman.

Git: