University of Cape Town

CSC2001F

Assignment 1

Binary Search Trees

Electronic Telephone Directory

BSSDIN001

Dino Bossi

The Problem

We were posed with the challenge of creating an Electronic Telephone Directory application in Java, while implementing a Binary Search Tree as the internal data structure. Data from a supplied data file was to be inserted into a Binary Search Tree. Each data item contains an address followed by a telephone number and finally a name. Each node of the Binary Search Tree contains a data item, using the specific name as the key for each node.

Once the appropriate Binary Search Tree was created various operations needed to be completed using the Tree. An application allowing the in-order printing of the Tree was required. This required the ability to traverse the tree in order, leading to the issue of correctly initialising a Binary Search Tree which contained a string as the key (which meant the alphabetical comparison of the keys was required to correctly create the Tree). The ability to alphabetically order keys beginning with the same letter was not a requirement, but was implemented in my solution.

Multiple search operations were also to be performed on the tree, namely searching for specified queries contained in a query file. Each query is a name and thus searching the Tree was done by searching for a key equal to the query. Since the query file was populated from the original data a match was found and a 'found' result always obtained.

An additional application was required which stored the data in an unsorted array and then used the same query file to perform a search on the array. The aim of creating 2 applications with the same end functionality was so that an experiment could be conducted to compare the relative run-times. Both searching applications (SearchIt for the Binary Search Tree and SearchItLinear for the unsorted array) were to be assessed, with regards to their execution speed, in the experiment and the results presented.

Design

The source code for the Binary Search Tree and Node classes was not my own, but was modified to handle the phone directory data. The following changes were made to the Binary Search Tree and Node source code:

i. Nodes have two attributes, namely a key and a fullentry String attribute. The key attribute contains the name taken from the telephone directory entry, while the fullentry attribute contains the entire directory entry (i.e. a single line from testdata).
ii. The comparison made when inserting a new node into the tree uses the .compareToIgnoreCase() method rather than a simple greater or less than operation.

The use of a static methods is to allow each application to have only one class which contained all the code necessary to perform the task required of it as well as being completely self-contained. Since the code needed was very minimal it was more practical to only write one or two static methods. This removed the need for a specific object to be created in the main method, as well as allowing for the methods to be called statically.

PrintIt

The PrintIt application is comprised of a single static method and a main method. The static method handles the loading of the testdata file, initialising the Binary Search Tree, inserting each entry (which was each line in testdata) into the Tree and calling the .inOrderTraverseTree() method which prints each node in the Binary Search Tree in order.

The static method print() mainly consists of code to load the file testdata into the Binary Search Tree. A Binary Search Tree object is created and initialised. A scanner object is used to read each line from the testdata file, one at a time. The code is placed within a try and catch statement in order to catch any errors that could potentially occur when attempting to read testdata. A node is added to the Tree using the information provided in the line from testdata. The substring method is used to obtain the name from the line. The name obtained is used as the key for the node, while the fullentry attribute is set as the specific line from testdata. Since the code is contained in a while loop each line of testdata is added as individual nodes to the Binary Search Tree. The .inOrderTraverseTree() method is used to print the tree in order, starting from the root node. The method .print() is called in the main method, allowing for the execution of the application.

SearchIt

The SearchIt application is comprised of two static methods and a main method. The static method loadData() handles the initialising of the Binary Search Tree and the loading of data from testdata into the Tree. The second static method search() handles the loading of the query file (queries.txt) and the searching for each query in the Tree.

The static method loadData()'s purpose is to load the testdata into a Binary Search Tree. It creates and initialises a Binary Search Tree object. A scanner object is created to allow the reading of testdata. A while loop is used to loop through each line (or entry) of testdata, 'adding' each entry to the Binary Search Tree. The node which is inserted into the tree has a

key equal to the name from the entry (obtained using substring), and its fullentry attribute equal to an entire line from testdata (which is a single entry). Almost all of the code in the method is contained within a try and catch statement in order catch any possible errors when working with the scanner object. The method then returns the populated Binary Search Tree.

The static method search() is used to load in the query file and search for each query in the Binary Search Tree initialised by loadData(). A Binary Search Tree object is initialised by calling the loadData() method. A scanner object is initialised to allow the reading in of queries from queries.txt. A while loop is used to loop through each line of the query file. The line read from the file is used as an argument to search through the Binary Search Tree. Since each query (each line in the query file) is a name it is comparable to each node's key, which allows for the searching of the tree. Thus, each query is used as a key to search through the Tree. A string is used to hold the returned entry. If the string is null (because the node was not found) then "Not found" is printed to the screen. If the node was found then its fullEntry attribute is printed to the screen.

The main method calls the search() method which in turn allows the application to be run.

SearchItLinear

The SearchItLinear application functions in almost the same way as SearchIt. Instead of initialising a Binary Search Tree populated by testdata it initialises an array and loads the contents of testdata into that array. It then loads the queries in the same way as the search() method from SearchIt, but instead of searching through the Tree it searches through the array implementing a linear search algorithm. If the entry that matches the query is found it is then printed to the screen. If it is not found then "Not found" is printed to the screen.

PrintIt Output

Below is the first 20 lines of output when the PrintIt application is run.

03707 Botsford Fork, Lima|489-848-7299|Abbott Alec
44812 Wilderman Mountain, Vallejo|318.679.5603 x712|Abbott Alexandria
76400 Barton Fields #044, Cerritos|507.340.1186|Abbott Alia
02519 Zackery Village, San Mateo|602.992.4016|Abbott Brando
88126 Bruen Common, Beverly Hills|788.603.8604|Abbott Elwyn
51832 Bayer Pass, Simi Valley|1-035-079-0176 x61480|Abbott Hosea
87191 Suite Z, Selma|823.283.2198 x7192|Abbott Ima
27010 Sanford Center, Stanton|822.752.1004|Abbott Josh
17296 Elta Crossroad #362, Newport Beach|516-835-0116|Abbott Leann
18565 Suite B, Fountain Valley|1-117-789-3061|Abbott Meda
22345 Runte Garden, Steubenville|1-654-279-2374|Abbott Murray
32763 Langosh Route, San Diego|297-763-2822|Abbott Novella
90282 Haag Keys, Garden Grove|(681)856-6604 x642|Abbott Rahsaan
52000 Marques Loaf #288, Placentia|(961)238-9093|Abbott Sadye
78469 Renner Mill, Agoura Hills|1-515-459-1556|Abbott Santina
96179 Feil Tunnel #352, Canton|1-052-394-1236 x29668|Abernathy Amparo
98827 Gerlach Pike Apt. 743, Apple Valley|1-486-893-0367|Abernathy Austyn
14576 Harber Knolls, Riverside|1-331-934-0147|Abernathy Catalina
23694 Pier F, Tempe|(552)753-8320 x85031|Abernathy Chadd
36296 Batz Walk, San Francisco|(637)882-6835 x72457|Abernathy Cicero

SearchIt and SearchItLinear

Below is the data in a randomly generated Query file, containing 20 queries (with each query being a name) taken from the testdata file. This data was given as input when executing SearchIt and SearchItLinear.

Query File Contents:

Blick Michel
Bogisich Ryann
Satterfield Madison
Lang Janet
Simonis Julien
Huels Sunny
Tillman Elnora
Dicki Nicolette
Wiza Jensen
Stracke Emmanuel
Moen Gennaro
Koelpin Ruby
Farrell Taryn
Brown Trinity
Weber Craig
Cole Estella
Becker Nicole
Jast Rogers
Schultz Weston
Crist Cassie

SearchIt Application Output:

41829 Grayce Mountains Apt. 812, Richmond|(662)107-2337 x510|Blick Michel
18911 Emmitt Curve #899, Concord|(118)484-5939 x1921|Bogisich Ryann
39214 Litzy Courts Apt. 688, Vincennes|849.197.3242|Satterfield Madison
09623 Gwen Branch #975, Tustin|709.291.8376 x3636|Lang Janet
99851 Domingo Shoal Apt. 682, Diamond Bar|355-743-8960 x49783|Simonis Julien
69746 Gutmann Drive, Jeffersontown|498-358-4276|Huels Sunny
61562 Alaina Avenue #213, Chino Hills|1-951-702-5279 x853|Tillman Elnora
20364 Smith Cliff #647, Huntsville|530-073-5904|Dicki Nicolette
37046 Casimir Island #241, Santa Barbara|(800)635-0385|Wiza Jensen
64877 Monserrate Mountains, Jasper|(218)885-9379|Stracke Emmanuel
95515 Thompson Streets Suite 627, Seward|604-975-2182|Moen Gennaro
39247 Shawn Roads Suite 254, Roseville|(633)638-9613 x19633|Koelpin Ruby
71314 Coy Road, Springdale|269-922-2604|Farrell Taryn
73022 Arlie Grove, Temple City|239.902.1574 x5410|Brown Trinity
23131 Wilkinson Gateway Apt. 196, Manhattan Beach|(125)533-6056|Weber Craig
05721 Penthouse, Sitka|1-584-900-7998 x42073|Cole Estella
86650 Lemke Alley #666, Crown Point|1-333-828-9758 x807|Becker Nicole
03967 Raynor Estates Apt. 478, Irwindale|1-821-969-4090 x520|Jast Rogers
74094 Reynold Canyon, Burlingame|1-832-247-1368|Schultz Weston
39527 Lelia Loaf, Laguna Hills|1-330-895-8258|Crist Cassie

SearchItLinear Application Output:

41829 Grayce Mountains Apt. 812, Richmond|(662)107-2337 x510|Blick Michel
18911 Emmitt Curve #899, Concord|(118)484-5939 x1921|Bogisich Ryann
39214 Litzy Courts Apt. 688, Vincennes|849.197.3242|Satterfield Madison
09623 Gwen Branch #975, Tustin|709.291.8376 x3636|Lang Janet
99851 Domingo Shoal Apt. 682, Diamond Bar|355-743-8960 x49783|Simonis Julien
69746 Gutmann Drive, Jeffersontown|498-358-4276|Huels Sunny
61562 Alaina Avenue #213, Chino Hills|1-951-702-5279 x853|Tillman Elnora
20364 Smith Cliff #647, Huntsville|530-073-5904|Dicki Nicolette
37046 Casimir Island #241, Santa Barbara|(800)635-0385|Wiza Jensen
64877 Monserrate Mountains, Jasper|(218)885-9379|Stracke Emmanuel
95515 Thompson Streets Suite 627, Seward|604-975-2182|Moen Gennaro
39247 Shawn Roads Suite 254, Roseville|(633)638-9613 x19633|Koelpin Ruby
71314 Coy Road, Springdale|269-922-2604|Farrell Taryn
73022 Arlie Grove, Temple City|239.902.1574 x5410|Brown Trinity
23131 Wilkinson Gateway Apt. 196, Manhattan Beach|(125)533-6056|Weber Craig
05721 Penthouse, Sitka|1-584-900-7998 x42073|Cole Estella
86650 Lemke Alley #666, Crown Point|1-333-828-9758 x807|Becker Nicole
03967 Raynor Estates Apt. 478, Irwindale|1-821-969-4090 x520|Jast Rogers
74094 Reynold Canyon, Burlingame|1-832-247-1368|Schultz Weston
39527 Lelia Loaf, Laguna Hills|1-330-895-8258|Crist Cassie

Comparison Experiment

The time taken to complete a search operation using a Binary Search Tree and an array should be compared in order to accurately identify the 'faster searching' data structure. It is known that a Binary Search Tree search operation has a O(log(n)) algorithm analysis while an array search operation (i.e. comparing each element to that which is being searched for, utilising the linear search algorithm) has a O(n) analysis, however an experiment must be conducted to ensure that the Binary Search Tree does indeed have a faster search operation.

Experimental Design:

Both data structures load in the data from the testdata file. A query file containing n queries will be created once for each value of n (the values of n between 1 and 10000; the n values used for the experiment are found below). The query file is created using another application (this application takes n number of random entries from the testdata file and extracts the names from each to form queries; no duplicate queries are present in any n length query file). Each application's (SearchIt and SearchItLinear) execution will be timed using the 'time' unix command. The system and user time from the output will be summed to determine a test execution time. Each application will be 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.). Each test execution time recording will be recorded and used to determine an average. Upon completion of the time recordings the averages must be found and compared to determine which application has the faster execution time for varying values of n.

Expectations and Recordings:

We would expect the Binary Search Tree search operation to complete much faster than a search operation on an array, if both have the same size (i.e. same value of n). This expectation is due to the 'Big O' analysis, as well as a Binary Search Tree implementing concepts from the Binary Search algorithm.

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 seconds and milliseconds. (xx.xxx)

Time Readings for SearchIt Application Execution

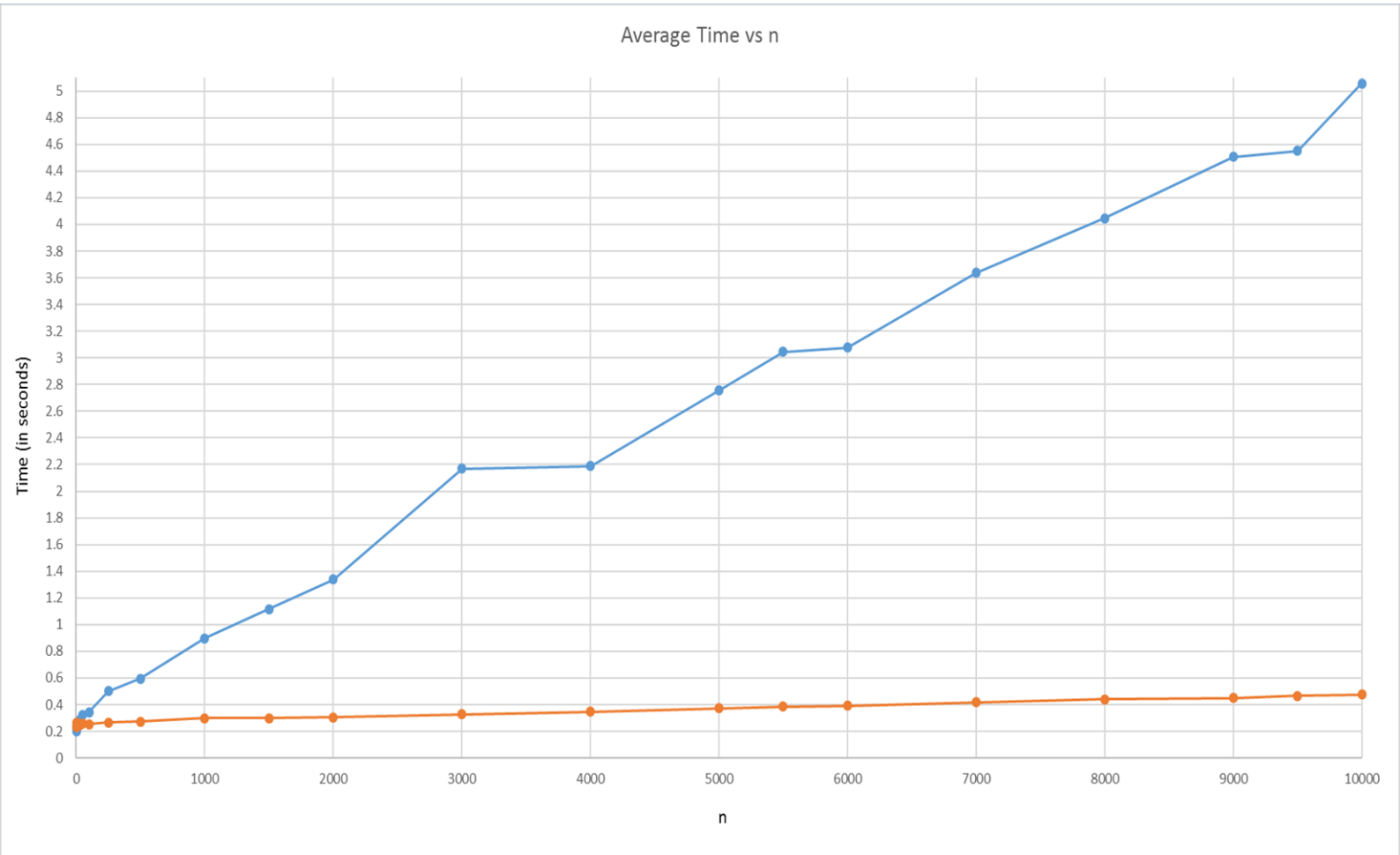
n	Time 1	Time 2	Time 3	Time 4	Time 5	Average
1	0.228	0.232	0.240	0.228	0.240	0.2336
5	0.300	0.292	0.252	0.232	0.252	0.2656
10	0.228	0.248	0.232	0.236	0.240	0.2368
25	0.236	0.260	0.236	0.244	0.260	0.2472
50	0.256	0.240	0.292	0.248	0.242	0.2556
100	0.248	0.240	0.256	0.276	0.248	0.2536
250	0.264	0.264	0.272	0.276	0.256	0.2664
500	0.276	0.272	0.268	0.276	0.264	0.2712
1000	0.280	0.320	0.304	0.304	0.280	0.2976
1500	0.316	0.288	0.288	0.300	0.296	0.2976
2000	0.296	0.300	0.308	0.320	0.296	0.304
3000	0.328	0.336	0.324	0.332	0.324	0.3288
4000	0.340	0.344	0.352	0.356	0.344	0.3472
5000	0.376	0.368	0.370	0.372	0.372	0.3716
5500	0.408	0.380	0.376	0.364	0.401	0.3858
6000	0.376	0.388	0.396	0.400	0.392	0.3904
7000	0.440	0.424	0.400	0.420	0.408	0.4184
8000	0.460	0.452	0.428	0.416	0.440	0.4392
9000	0.432	0.452	0.448	0.448	0.464	0.4488
9500	0.468	0.476	0.460	0.464	0.468	0.4672
10000	0.480	0.480	0.464	0.476	0.472	0.4744

Time Readings for SearchItLinear Application Execution

n	Time 1	Time 2	Time 3	Time 4	Time 5	Average
1	0.204	0.196	0.204	0.200	0.203	0.2014
5	0.220	0.220	0.220	0.284	0.228	0.2344
10	0.248	0.240	0.256	0.260	0.252	0.2512
25	0.272	0.292	0.260	0.292	0.288	0.2808
50	0.288	0.332	0.332	0.328	0.332	0.3224
100	0.344	0.324	0.368	0.340	0.336	0.3424
250	0.472	0.464	0.600	0.520	0.456	0.5024
500	0.568	0.600	0.576	0.648	0.584	0.5952
1000	0.892	0.880	0.920	0.896	0.904	0.8984
1500	1.004	1.188	1.156	1.052	1.184	1.1168
2000	1.396	1.348	1.308	1.380	1.260	1.3384
3000	1.828	2.404	2.388	1.812	2.412	2.1688
4000	2.304	2.084	2.260	2.308	1.980	2.1872
5000	2.728	2.812	2.768	2.674	2.800	2.7564
5500	2.928	2.748	3.552	2.936	3.056	3.044

6000	2.844	3.184	3.036	3.064	3.260	3.0776
7000	4.254	3.548	3.840	3.108	3.444	3.6388
8000	3.520	4.248	4.048	4.164	4.252	4.0464
9000	4.654	4.504	3.960	4.600	4.820	4.5076
9500	4.684	4.164	4.584	4.556	4.764	4.5504
10000	4.980	4.952	5.548	4.356	5.448	5.0568

A graph representing the average time versus the value of n can be found below. Both SearchIt and SearchItLinears results are plotted on the same graph to allow for comparison between the two sets. SearchIt results are in plotted in orange while the SearchItLinear results are plotted in blue.



Discussion and Conclusion:

It is to be noted that the results for SearchIt and SearchItLinear contained certain outliers (e.g. SearchItLinear at n = 3000). However, the results obtained can still be used to show that for almost all values of n the SearchIt algorithm executed faster than its linear counterpart.

Interestingly we also note the rate at which execution time increases as n increases is much less for SearchIt than SearchItLinear. This indicates that while an array can be searched just as sufficiently as a Binary Search Tree for very small data sizes, as the data size increases it becomes exponentially slower.

The noticeably faster searching execution of the Binary Search Tree can be attributed to the concepts it lends from the Binary Search algorithm. The most notable concept 'borrowed' is the idea of splitting the data into two sets, one set containing values lower than a specific value and the other set containing values higher than the specific value. This allows for the continual rejection of previously potential matches just due to the 'data structure' of the Binary Search Tree.

In conclusion, the tabulated time recordings as well as the graph all indicate that the Binary Search Tree has the faster searching algorithm than searching linearly through an array.

Summary Statistics

JUnit Testing:

Numerous Junit test classes were created to test each application as well as test the BinarySearchTree and Node source code. The tests for PrintIt, SearchIt and SearchItLinear were conducted by providing a testing version of testdata and queries.txt. 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:

i. TestBinarySearchTree
    a. Test to ensure added nodes are in fact added to the BinarySearchTree
    b. Test to ensure a non-existing node is not found when searched for
    c. Test to ensure existing node can be found when searched for
    d. Test to ensure in order traversal is printing nodes in correct order (alphabetical order in this case)
ii. TestPrintIt
    a. Test to ensure printing of BinarySearchTree is correct and that all methods within .print() are producing expected results. Only one test was created since any errors in the .print() method will result in a test failure.
iii. TestSearchIt
    a. Test to ensure that queries in queries.txt are searched for and found if present in testdata and the BinarySearchTree, as well as ensuring the data is loaded into BinarySearchTree successfully. The test includes searching for queries which are not present in the Tree, resulting in a "Not found" response.
iv. TestSearchItLinear
    a. Test to ensure that queries in queries.txt are searched for and found if present in testdata and in the array (initialised to contain all data/entries from testdata). This test determines whether the array is initialised successfully as well as testing if searching and returning found entries works successfully.

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.

Git:

Git was used for this assignment to keep a remote up-to-date version of the entire assignment. Commits to the master branch were made only after a work-in-progress version of the application was running correctly. The make clean command was always run before adding, committing and pushing to Git in order to remove unnecessary files from the repository.

In total 18 commits were made over the duration of the assignment. The repository consisted of all source code, the JUnit test classes and the makefile. If further assurance is needed the repository can be found at this website: https://github.com/zzzaJ/Assignment1