COMP20003 Algorithms and Data Structures Second (Spring) Semester 2019

Assignment 1
Taxi & For-Hire Vehicle Trip Dataset
Information Retrieval using Binary Search Trees

Tristan Chandraratna
(Student ID: 1091851)

Introduction:

This purpose of this project was to learn how to create, search and free binary search trees. The binary search tree that was used in this project consists of multiple nodes in which itself stored a pointer to its own data as well as 2 pointers for the lower nodes that are “left” and “right” of this node.

The data that is stored in this node is in the form of a linked list containing the specific data of each trip the taxis took. When inputing the data into this binary search tree, if the primary key, pick-up time for stage 1 and pick-up location for stage 2, is equivalent to that of an existing node, it would be added into the existing node’s linked list.

For the testing stage of this project, comma separated value(CSV) files from the New York taxi website provided in the assignment handout were downloaded, randomised and changed in file size. Random dates and times were picked from these files for stage 1 and random pick-up locations for stage 2 and entered into a ‘keyfile'. For each csv file, the keys were inputed into the program and the average and worst search times were recorded and calculated. Because the binary search tree in this project is unbalanced, on average, the program will have an average search time of O(log(n)), but with a worst case of O(n).

Assignment structure:

The assignment was split into two different stages. Within stage 1, a binary search tree was created using data that was stored in a CSV file. This was then queried with date time values entered in either manually or through a file using the UNIX operator. The program would find all nodes with the specified date time primary key and output them to a specified file. If none were found, “NOT FOUND” would be outputted to the same file. The number of comparisons made during this process would be outputted onto the stdout screen before all data is freed.

Stage two was very similar to stage 2, except that the primary key was defined as the pick-up location, containing only 256 individual codes.

Stage 1:

Figure 1

Average case	Expected	Tested
10000	13.288	18.000
20000	14.288	22.667
30000	14.873	23.750
40000	15.288	28.417
50000	15.610	24.917
60000	15.873	33.833
70000	16.095	38.917
80000	16.288	44.500
90000	16.458	44.667
100000	16.610	55.917

The figures below depict the data collected from the above mentioned testing. Figure 1 is a table of the average cases recorded for this stage and figure 2 is a graph represent the correlation between the two lines. It can be seen that the tested values are extraordinarily differing to what was expected. This could be skewed by the increase in frequency of larger, differing numbers as shown in Figure 3. It was observed that as the number of lines within each CSV file, a larger spread of results were observed. Through the testing undertaken, the time complexity of this search function is linearly proportional to the number of nodes added to the tree.

Figure 2

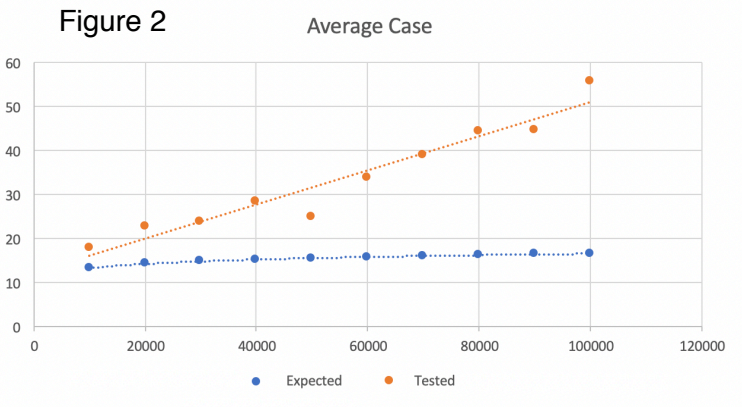


Figure 3

Worst case	Expected	Tested
10000	10000	40
20000	20000	48
30000	30000	43
40000	40000	94
50000	50000	57
60000	60000	101
70000	70000	98
80000	80000	109
90000	90000	71
100000	100000	199

Stage 2:

Figure 4

Average case	Expected	Tested
10000	13.288	12.083
20000	14.288	12.917
30000	14.873	16.083
40000	15.288	15.333
50000	15.610	18.417
60000	15.873	22.167
70000	16.095	15.500
80000	16.288	18.833
90000	16.458	24.333
100000	16.610	22.667

Using the same CSV files that were used in the first stage, figures 4 - 6 show the data recorded during the testing of stage two. Figure 4 shows the numerical relation between the number of lines in the files compared to the number of comparisons required to search the tree. Figure 5 is a graphical representation of the numbers, in which it is easy to see that as more data is added to the tree, the more variable the average case becomes. Whilst the larger amounts of data appear to be more volatile, as supported by figure 6, the average case for this stage tends towards O(log(n)), supporting the average time complexity stated above.

Figure 5

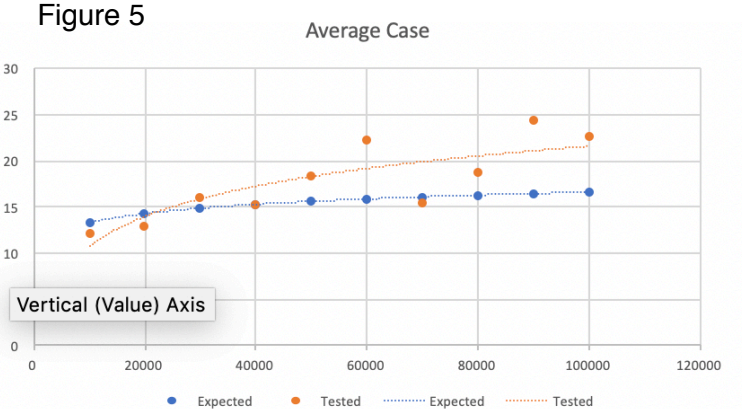


Figure 6

Worst case	Expected	Tested
10000	10000	32
20000	20000	23
30000	30000	43
40000	40000	32
50000	50000	34
60000	60000	43
70000	70000	32
80000	80000	56
90000	90000	63
100000	100000	83

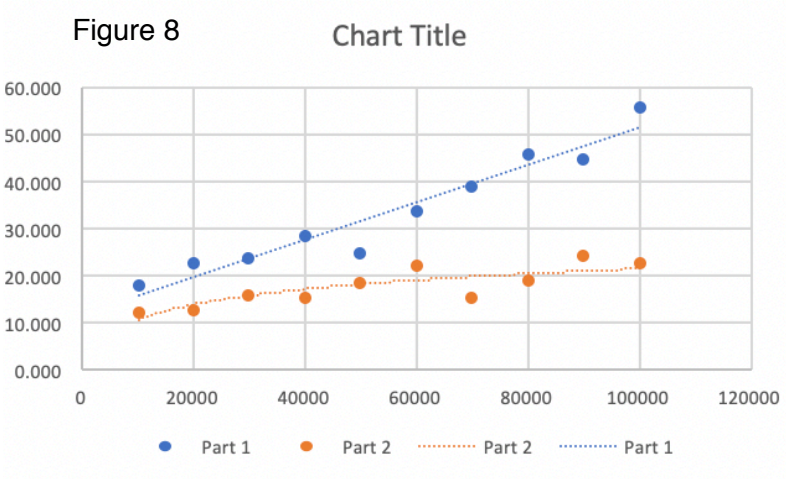
Discussion:

Whilst both of the stages used the same binary search tree, stage 2 appeared to be trending towards log(n) compared to stage 1's more linear trend-line as shown in figure 7 and 8. This could be due to the fact that whilst stage 1 has an infinitely large number of possible values, stage 2 only has 256 location identification numbers in which are available primary keys. This results in there only being a maximum of 256 nodes on the tree, in which all other nodes will be linked by their primary keys.

Figure 7

Average case	Part 1	Part 2
10000	18.000	12.083
20000	22.667	12.917
30000	23.750	16.083
40000	28.417	15.333
50000	24.917	18.417
60000	33.833	22.167
70000	38.917	15.500
80000	46.000	18.833
90000	44.667	24.333
100000	55.917	22.667

Figure 8



This project, whilst successful in completing the task set, could be made more efficient in a multitude of ways. Firstly, the unbalanced binary search tree would need to be converted an AVL tree. This would mean that the tree would have a worst case time complexity of O(log(n)) due to the fact that all branches of the tree are of the same length, causing the longest paths to be shortened, where the shorter paths are barely changed. This would be achieved by rotating branches as nodes are added to the tree.

Secondly, primary keys should be stored within the nodes on the trees instead of in nodes in the linked list. This would both increase the efficiency of memory as well as not requiring to access the linked list every time the primary key is required to be accessed.

Finally, if binary search tree was converted into a quaternary tree or a 2-3-4 tree. This would drastically decrease search times of the tree as well as create an easier, more natural AVL tree. More research would have to be out into this type of tree before it could be implemented into this project.

Conclusion:

Overall, this project successfully completed the task that was given. Whilst the above mentioned inefficiencies could be fixed within the code, this project is able to handle large amounts of code without breaking reliably.