Akira Wang
Student ID: 913391

COMP20003

Stage 3 Analysis

OVERVIEW

This experimentation stage will cover a large simulation on both Stage 1 and Stage 2 for the number of comparisons for different lengths of input. Also, the three given data sets will be used when timing tests. All timed tests were done on the UNIX server with /usr/bin/time command.

Methods

1. Run a number of simulations with 50 different sizes (5000-250000 at 5000 steps) 5 times, all randomly generated and tested through Python
2. Run timed tests on the UNIX server and take the average computation time over 5 runs

SPECIFICATIONS (Python - Jupyter Notebook)

make()

Create randomly generated input files starting from 5000 until 250000 at 5000 increments. Also create random keys to be tested with at least 3 random keys not in the dictionary.

run(dict_version)

Run both stages with the randomly generated input files and keys. The output is from stdout and the output dictionary.

count(dict_version)

Count the number of comparisons per randomly generated file and average it out for analysis.

Code: https://github.com/akiratwang/not-uni-repo/blob/master/Stage%203%20.ipynb

Implementation Notes:

The binary search tree algorithm implemented for both stage 1 and 2 is recursive, meaning a significantly slower time compared to an iterative binary search tree. However, for the sake of this analysis, the differences between recursive and iterative will be ignored.

Part 1

	Dictionary Input Size	Stage 1	Stage 2
0	5000	14.685039	14.625984
1	10000	16.142857	16.039683
2	15000	17.784314	17.623529
3	20000	18.266667	18.078431
4	25000	18.615686	18.376471
5	30000	18.517647	18.223529

(Figure 1.1) - Sample output from randomly generated files

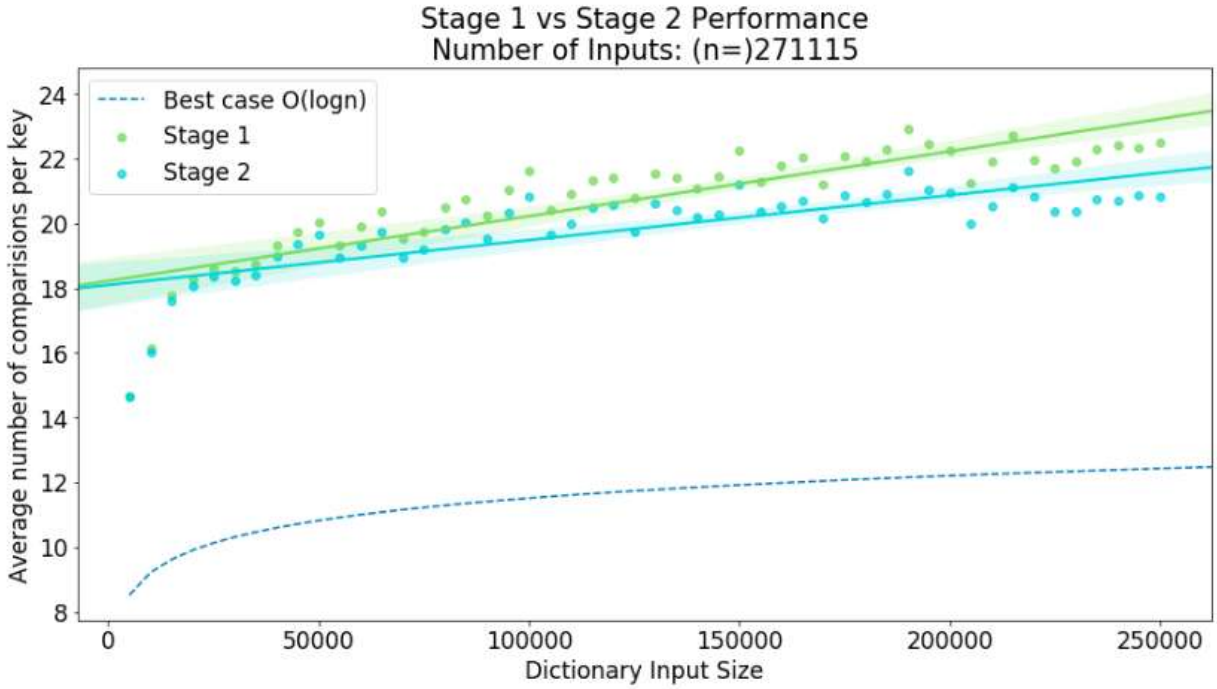
As seen above in Figure 1.1, randomly generated files tested in Stage 2 show that the average number of comparisons are slightly slower than that of Stage 1.

As expected of a binary search tree, Stage 1 had inserted duplicates to the left node (root->left), whilst Stage 2 had duplicates inserted into a linked list (root->next). The difference in performance is only when there are a large number of duplicates where instead of checking for duplicates left recursively, Stage 2 implementation will automatically then traverse a linked list until it hits NULL, signifying the last duplicate.

This means in terms of comparisons, Stage 1 will have on average (number of comparisons + number of duplicates) whilst Stage 2 will have on average (number of comparisons + 1 to traverse list).

A visualisation of all the tests done on the data can be seen below Figure 1.2. Although the implementation used for my assignment is not perfect, we can see that the distribution from the randomly generated tests still follow an O(logn) distribution. Furthermore, Stage 2's implementation starts becoming slightly more efficient (-2 comparisons on average) as the number of inputs increase.

For any improvements that may be done for this assignment, it would be ideal to implement an AVL binary search tree, which would significantly decrease the number of comparisons to O(logn).



(Figure 1.2) - Visual graph of stage 1 against stage 2, including a trendline using 95% Confidence Interval (If these tests were to be run again, 95% of the time will yield in the trend line being between the line and the lighter gradient surrounding it)

Part 2

```
-bash-4.1$ time dict1 athlete_events_filtered.csv output1_1.txt < keyfile.txt
A Dijiang --> 1
Zzimo Alves Calazans --> 34
randomperson --> 23
12345 --> 1
Aquil Hashim Abdullah (Shumate-) --> 20
Genowefa Minicka (Cielik-) --> 28
Per Johan Daniel Wallner --> 30

real    0m2.589s
user    0m0.558s
sys     0m0.145s
-bash-4.1$
```

(Figure 2.1) - Example of the bash command used to time the stages.

The actual command used (in order to get te output to a text file) was:
/usr/bin/time dict1 athlete_events_filtered.csv output1_1.txt < keyfile.txt 2> time1_1.txt
and variations depending on the csv, stage and output.

	Stage 1	Stage 2
normal	2.47	2.06
alternative	2.18	1.93
alternative2	641.87	656.88

(Figure 2.2) - The average time of five tests for each stage and the input csv

```
Keys to be searched:
A Dijiang
Zzimo Alves Calazans
randomperson
12345
Aquil Hashim Abdullah (Shumate-)
Genowefa Minicka (Cielik-)
Per Johan Daniel Wallner

Key: A Dijiang, is the first value in the csv
Key: Zzimo Alves Calazans, is the final value in the csv
```

(Figure 2.3) - The 7 keys used to search which include the first key, last key, 3 random keys that exist and 2 random keys that do not exist in the input

As all the tests were done using the keys from Figure 2.3, we can be assured that the only factor that affects the result is the server CPU and load which will be ignored.

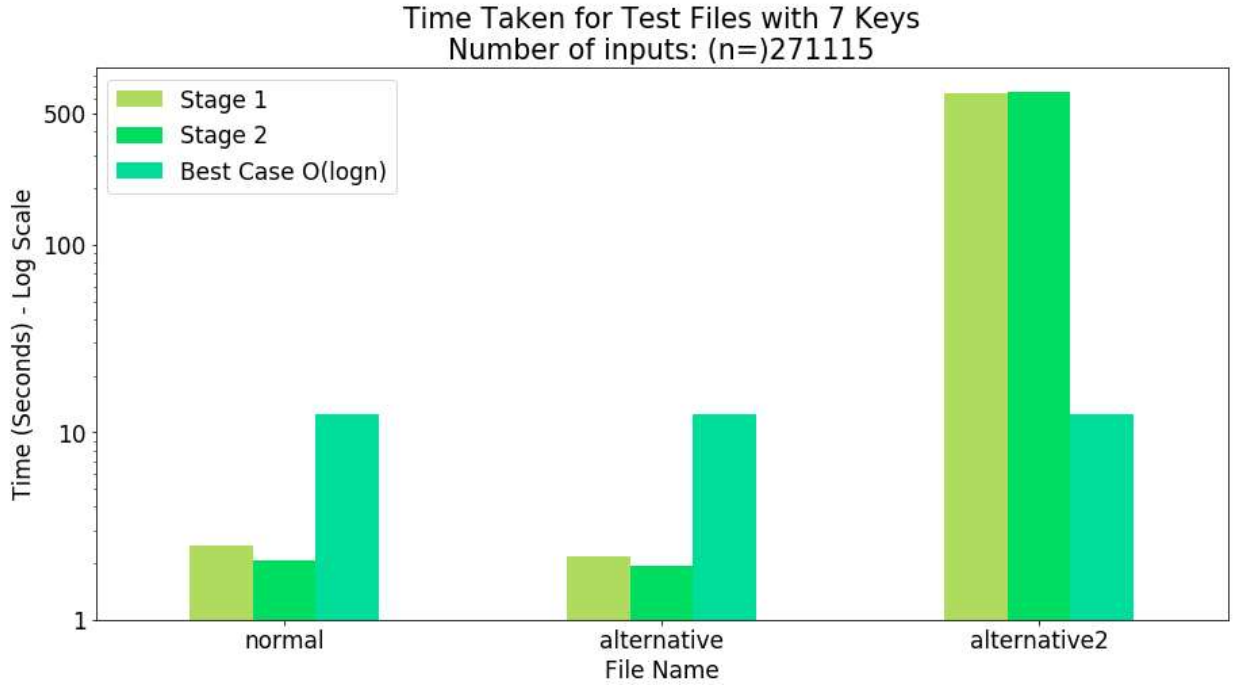
From what the results say in Figure 2.2, we can clearly differentiate between the csvs.

athlete_events_filtered.csv	Random, duplicates sorted
athlete_events_filtered_alternative.csv	Random, duplicates are random
athlete_events_filtered_alternative2.csv	Sorted, duplicates are sorted

For both stages with the normal and alternative input, not much has changed due to the nature of a Binary Search Tree. As the keys are inserted using a compare function, it is expected that having a random values inserted will greatly reduce the time taken (root->left then root->right or root->right then root->left will both cut the other half of the tree out). However, for the alternative2 file, the input is already sorted - including the duplicates, resulting in a worse case scenario of O(n). This means that if we were to search for the last key, the algorithm would need to traverse the entire left branch in order to find the key.

Figure 2.3 is a visualisation of the time taken to complete each task. The y-axis is set to a logarithmic scale for easier viewing. With the graph, we see an average case scenario of O(logn) for n = number of inputs, and that the current algorithm implemented does indeed perform well and as expected - A worse case of O(n) and best case of O(logn).

(Figure 2.4) - Visualisation of the time taken between each stage and best case (constant)



Conclusion of results

As expected, the binary search tree used in both stages yield close resemblance of O(logn) time, considering the use of recursive functions to insert and search keys. Although the algorithm shines best when there is more random ordering in input (comparisons will “drop” more results), we can see in the worst case scenario using the alternative2 file will display O(n) time.

A future implementation may be to use an AVL like algorithm, which would both considerably reduce time with ordered inputs whilst maintaining the characteristics of the original algorithm. This is due to the AVL algorithm auto balancing the tree height, reordering the nodes so that it is in order regardless and result in the worst case O(logn) time compared to the BST worst case of O(logn).

Notes to marker:

The struct in my assignment uses pre-buffered arrays for input, given that it will always be well formatted. I know we were supposed to malloc each record in but I didn’t have enough time to implement it and as such, have left it with predefined buffers. Also I would have actually liked to have a test run through on the AVL implementation to see the actual differences (if you guys do release solutions or show us during the tutes that’d be sick). Overall a very challenging yet fun assignment to accomplish!