COMP20003- Algorithms and Data Structures

Assignment 1- Experimentation

Introduction

In order to empirically test the complexity of the implemented algorithms, an extensive and diverse number of datasets and search keys were constructed and the number of comparisons for each dataset-key pair were recorded. There were datasets with and without duplicate keys, datasets sorted in ascending and descending order as well as in random order, and datasets of varying sizes. These dataset-key pairs were created using the Pandas library in Python, and the pairs were run into the algorithm from within python using the following line:

```
os.system(f"./dict2 data/{key}.csv out.txt < keys.txt *>> results.txt")
```

Each dataset had 20 corresponding keys; 10 that were randomly sampled from the dataset, and ten that was randomly sampled from keys in the full-size dataset that were not in the sample. The data was then collected from results.txt and inserted into a pandas table like the one below for further analysis.

	E0	NE0	E1	NE1	E2	NE2	E3	NE3	E4	NE4	...	E7	NE7	E8	NE8	E9	NE9	N	Dup	Avg E	Avg NE
0	16	11	10	18	15	9	21	14	9	9	...	4	10	21	14	6	14	1000	0	12.0	12.8
1	20	14	18	9	9	14	10	14	11	11	...	12	21	12	12	15	10	2000	0	13.3	12.7
2	16	18	13	16	12	9	18	16	13	15	...	4	19	11	16	18	14	3000	0	13.6	15.0
3	14	22	14	17	12	12	18	12	15	16	...	13	18	15	13	17	20	4000	0	14.8	16.4
4	12	21	14	15	14	15	11	21	10	13	...	12	19	15	19	12	14	5000	0	12.9	17.0
5	15	16	12	22	24	16	15	12	16	15	...	10	11	19	25	20	16	6000	0	15.6	17.0

Each row corresponds to a separate dataset with sample size N. Column names E refer to keys that Existed within the dataset, column names NE refer to keys that did Not Exist within the dataset. The numbers next to the E's and NE's refer to the 10 different randomly sampled keys that were trialled on each dataset. This repetition was done to offset noise from random sampling somewhat. The column Dup was a 1 or 0 depending on if the dataset contained duplicate keys or not. The averages of the E and NE columns was also taken and placed in the end average columns.

Curve fitting was done with the numpy library, and only equations of forms y = ax and y = aln(x) were considered. All graphs were constructed with the Python Seaborn library.

A quick note before the report: It is important to realise that Big O notation describes the behaviour of algorithms in the limit as n approaches infinity, therefore it is impossible to conclusively show the complexity of an algorithm empirically by observing just a graph. Perhaps there is a lurking n^2 term with a tiny coefficient (<<1). Such analysis can only be confirmed by an analysis of the algorithm itself.

Stage 1

Method

1. First, create two subsets of the full-sized dataset:
    a. A dataset with duplicates in *PUdatetime* column – 135,701 rows
    b. A dataset with no duplicates in *PUdatetime* column – 135,344 rows
2. For each of these datasets, create sample datasets of three "kinds" with sizes 1,000 to 100,000 (in jumps of 1000). The "kinds" are as follows:
    a. A dataset in random order – named *shuffle*
    b. A dataset with PUdatetimes in ascending order – named *fsorted* (forward)
    c. A dataset with PUdatetimes in descending order – named *bsorted* (backward)
3. For each of these kinds of dataset, randomly choose twenty keys of two kinds:
    a. 10 keys that are in the sample dataset – named ***existing keys***
    b. 10 keys that are in the full-sized dataset, but not in the sample dataset – named ***non-existing keys***
4. Now, for each of these keys and associated datasets, run the dict1 program with the key and dataset and count the number of string comparisons
5. Store this data for later aggregation and analysis

Data & Comparisons to Theory

The **number of key comparisons for datasets pre-sorted in ascending order and descending order appeared to be identical** except for random variation due to sampling (see Figure 1). Straight lines could be fit to both forwards and backwards sorted datasets indicating that a search on these binary trees could be done in O(n). This is expected even in theory as when sorted data is inserted into a binary search tree, a stick is created i.e. The binary search tree becomes essentially a one-dimensional linked list with O(n) searches since each child only has one child.

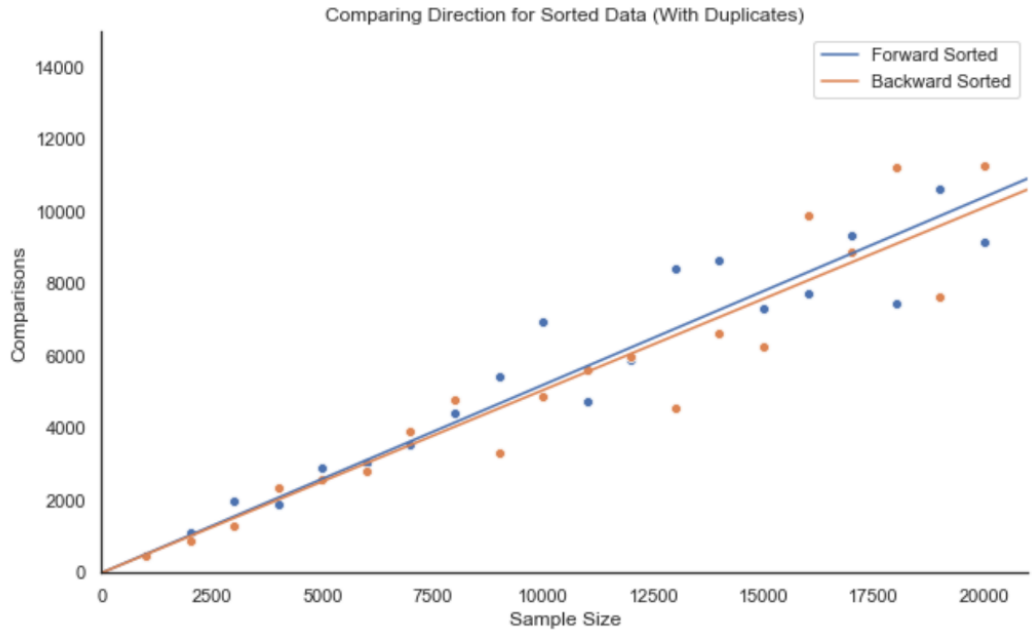

Figure 1

The kind of key only appeared to matter for **shuffled data, where keys that existed in the data set were found with less comparisons that those that did not exist in the dataset** (see Figure 2). This is to be expected as when a key is non-existent in a binary search tree, a search for the key must go to a leaf node of the tree (the worst case for a balanced binary search tree). However, when a key does exist in a binary search tree, a search can stop **earlier (hence fewer key comparisons)** and in fact this is the average case for a binary search tree. Either way, these searches are both O(logn) in theory, and this is confirmed empirically by observing how well the log curve fits to the data.

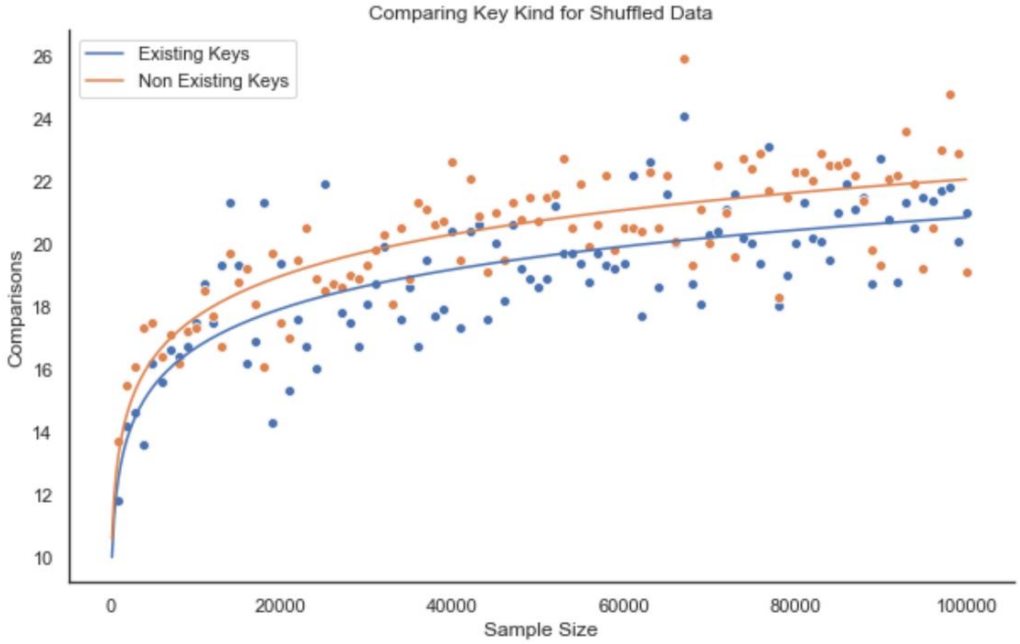

Figure 2

The kind of key did not matter for the sorted datasets. This is because the binary search tree is a stick for sorted inputs, hence can be treated like a linked list. The slope of the line in Figure 3 is approximately ½, and this is because on average it takes n/2 comparisons to find the position of a key in a sorted linked list regardless of whether the key is in the list. So again, the theory is confirmed by the graph with a search complexity of O(n)

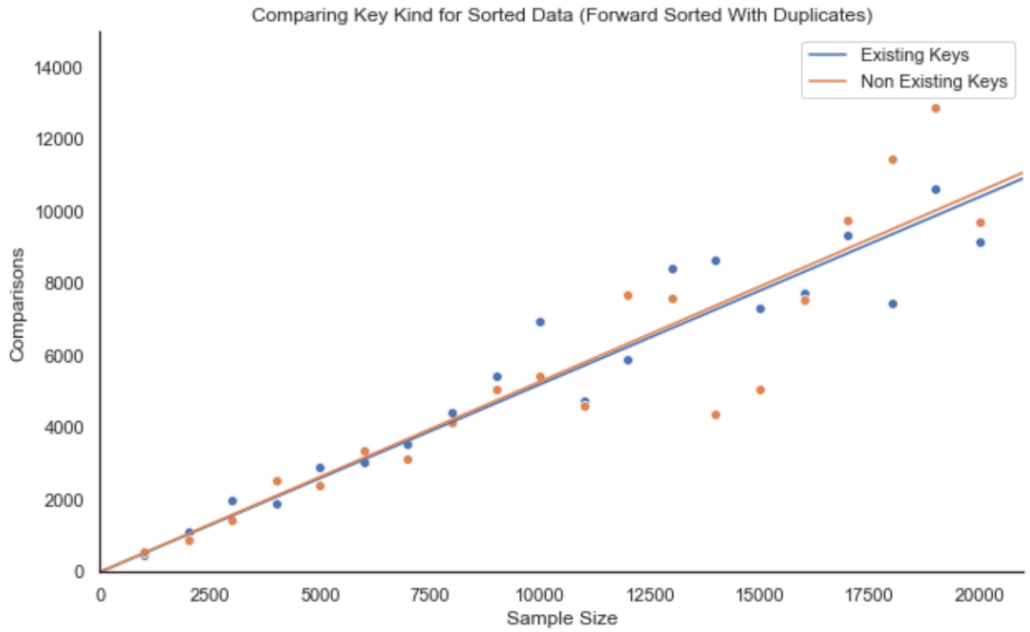

Figure 3

Whether or not duplicates existed in the dataset appeared to make no difference to any tests. This is because duplicates had nothing to do with the string comparisons, since once a matching key had been found in the binary search tree, it didn't matter how long the corresponding linked list was; No more key comparisons were made. However, in the extreme case (i.e. all data is a duplicate), O(1) behaviour would be observed since only one key comparison would need to be made. Similarly, it is expected that for a fixed data size, increasing the number of duplicates would reduce the number of key comparisons purely because the tree would be shorted but each linked list would be longer.

Stage 2

Method

1. Locate the datasets created for stage 1
2. For each of these kinds of dataset, randomly choose Pickup location IDs (PIDs) of two kinds:
    a. 10 PIDs that are in the sample dataset – named ***existing PIDs***
    b. 10 PIDs that are in the full-sized dataset, but not in the sample dataset – named ***non-existing PIDs*** (If one cannot be found, pick random numbers between 1 and 265 until one of these numbers is not in the sample dataset)
3. Now, for each of these keys and associated datasets, run the dict2 program with the PID and dataset and count the number of string comparisons
4. Store this data for later aggregation and analysis

Data & Comparisons to Theory

Regardless of duplicates or not, whether the PID existing in the dataset or not, or even the kind of dataset chosen, **a dataset of size n would require n PID comparisons** to be made in order to locate all corresponding data (see Figure 4). This makes sense as the PIDs were found with an in-order traversal of the entire binary search tree, which will always require n string comparisons and thus be O(n). This is necessary because the dictionary was built on a different key, thus was no help in locating required PIDs.

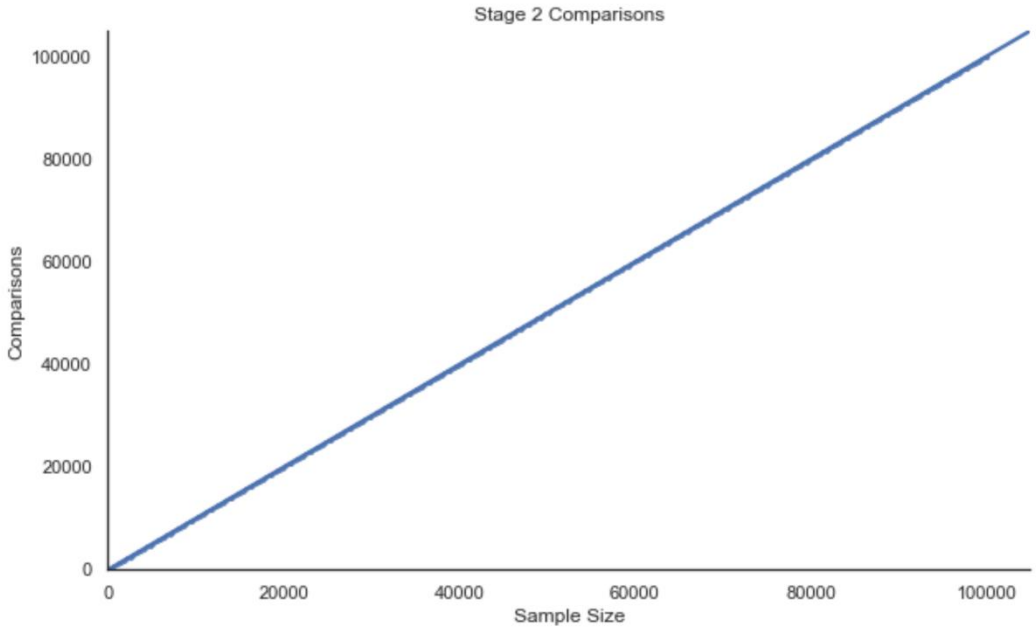

Figure 4

Conclusion

It was found that the theoretical complexities for searching both by key and via an in-order tree traversal were matched by empirical evidence as expected. The searches on shuffled data were completed in O(logn), whilst the searches on pre-sorted data were completed in O(n). The number of key comparisons did not change between input data that was sorted in ascending or descending order since both produced a stick. The kind of key did not matter for sorted inputs since the sticks are essentially sorted linked lists, but keys that existed in the dataset were found more quickly than those that did not for the shuffled data. Whether or not duplicates existed in the dataset made no observable difference to any tests which supports the theory. For stage 2, the only variable that effected the number of string comparisons was the size of the dataset since a full in-order tree traversal will look at every single node.

Recommendations

For future studies, it might be interesting to consider the best and worst cases for the algorithm i.e. searching for a key that is at the root of the binary search tree, or searching for a key at the end of a stick binary search tree. However, nothing surprising should come out of such endeavours as the theory is very well understood already. As mentioned earlier, algorithms can never be proved to be of any given complexity via empirical analysis and instead must be analysed theoretically. Hence, any further research should be focused into theoretical analyses since no amount of datasets, sampling, computing power or time will prove the Big O complexity of searches on a binary tree.