COMP20003 Assignment 1: Experimentation Report

Alan Guo, 831747

Abstract

This assignment serves to explore information retrieval using binary search trees (BST) through the implementation of a dictionary in C. In both inserting and searching the tree, one traverses through the nodes by the sorted nature of 'left' and 'right' leaves. It is this same feature which allows a significant reduction in search time, as elements which are significantly greater or lesser than the searched key are neglected early on.

A dictionary is a structure which holds a collection of objects, referred to by their 'key', which returns information on the object. The natural modification to the binary search tree to implement a dictionary is to put both key and information part of the node struct. Then, given the node, comparison can be made by accessing the key and information can be retrieved by accessing the data.

Introduction:

Two of the operations of this BST dictionary explored in this assignment is the insertion and search time complexity. Two factors which can impact this is:

1. The order in which the data is inserted (grouped, randomly, sorted)
2. The method of handling insertions of duplicate keys

Through theory, it is expected for randomised insertion order to perform best (with search having average case $O(\log n)$ operations), as it takes advantage of the tree structure with highest probability. On the other hand, it is expected for sorted insertion to be the most time complex/costly, as the structure is reduced to a linked list, (with search having $O(n)$ operations). The aim of the experiment is to verify and explore the significance of these factors, by varying datasets (for insertion) and key files (for searching).

Method:

Our main dataset which was used as a context to implement this dictionary is Olympics athlete dataset across the years, provided for the assignment. In this context, Olympians' name is the key, and duplicate keys arise from returning athletes or athletes which are involved in different events.

In the "athletes" dataset, three versions of it was tested as input:

1. Grouped: Athletes are grouped by ID
2. Random: The athletes are in random order.
3. Sorted: The athletes are sorted ascending alphabetical order.

In addition to this, a sample input had also been created, called "even numbers". It is a CSV file with the name column the even numbers 0002 to 2000 (formatted as thus), with five copies of each, reaching a total of 5000 inputs. There are three versions to this dataset:

1. Random: The numbers are randomised.
2. Ascending: The numbers are sorted in ascending order.
3. Descending: The numbers are sorted in descending order.

In addition to this, input sizes of a random selection of 500, 1000, 1500, 2000, 2500, 3000, 4000, 5000 points of the "even numbers" dataset was considered. A graph was created for each of the three versions of increasing-size "even numbers" datasets, which compared and details the growth of dict1 and dict2 mean number of comparisons.

For stage 1, the left side of each node consists of all key names less or equal to it in alphabetical order. Thus, all duplicate keys will be treated as a leaf that will traverse to the left when it is compared to an identical key. Likewise, during the binary search, the search function does not cease after it has found its first match but must continue to the left to search for duplicate keys. This dictionary will be denoted as dict1.

For stage 2, each node will be appended with a pointer to a linked list, which will consist of data with identical keys. In this BST dictionary, the search function can cease searching other nodes once a match is found, and simply traverse the list, one element after another, to retrieve the duplicate key data. This dictionary will be denoted as dict2.

A dict1 and dict2 program was written to insert the dataset in the given order to create a BST dictionary, read an input key file and output the number of comparisons required for each key search. The number of comparisons was used as a metric of time complexity/cost, as it represents the number of function calls to match a given key to a key of a node.

To yield comparable results, both dictionaries were subject to the same inputs. For each dataset ("athletes", "even numbers"), a key file was created for each name, including duplicates. This key file was used for each variation of its corresponding dataset, which was converted into a CSV file, then imported into Excel and RStudio for analysis.

For "even numbers" and "athletes" datasets, for each given version of the dataset, a histogram of the number of comparisons in dict1 and dict2 were overlayed in one, with vertical lines representing means. Tables of summary statistics is also provided. 5000 random samples of size 1% of the dataset was used to create sample means, and the mean("`Mean(2)`") of that was also reported.

Results:

Fig 1: Even Numbers, Random Insertion

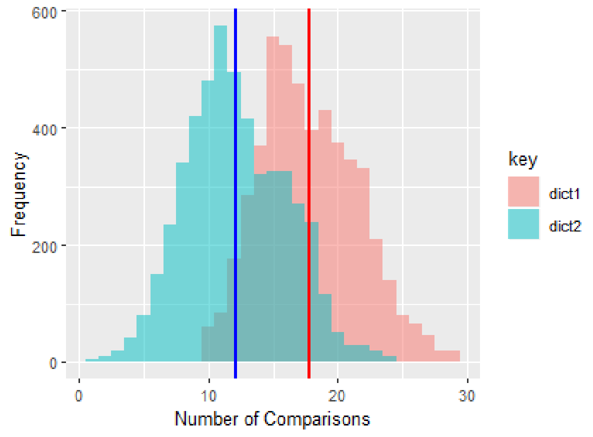


Fig 4: Athletes Data, Grouped Insertion

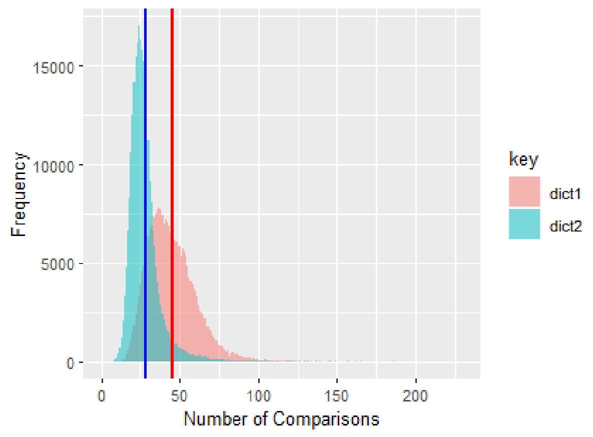


Fig 2: Even Numbers, Ascending Insertion

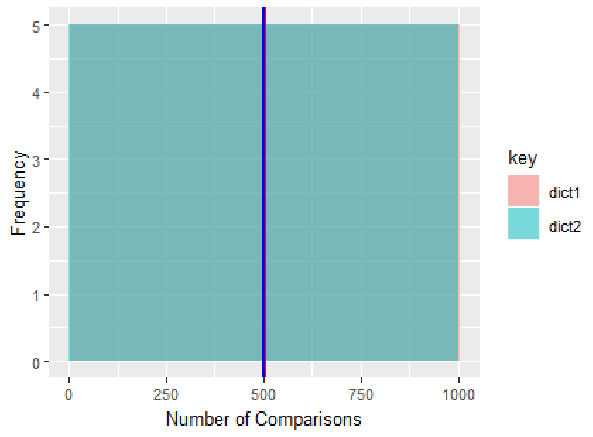


Fig 5: Athletes Data, Random Insertion

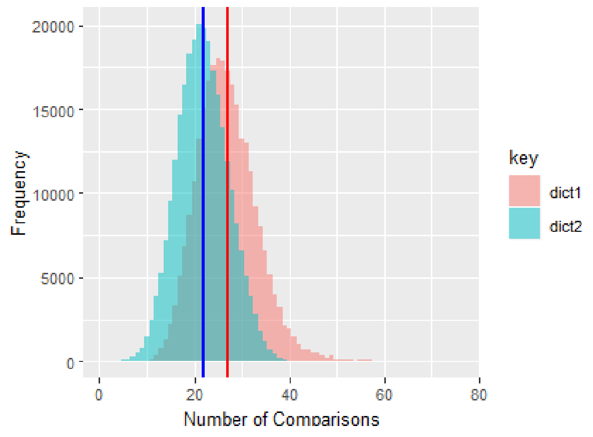


Fig 3: Even Numbers, Descending Insertion

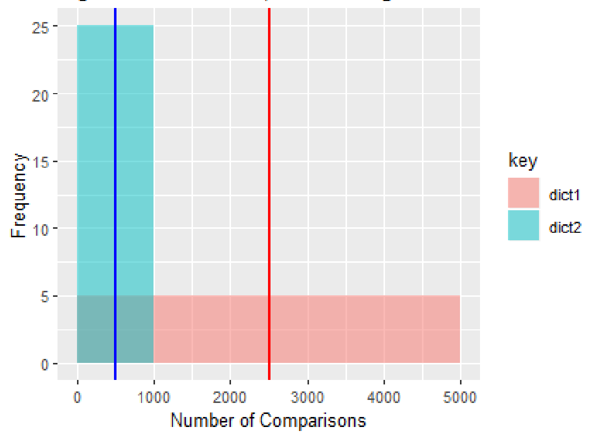
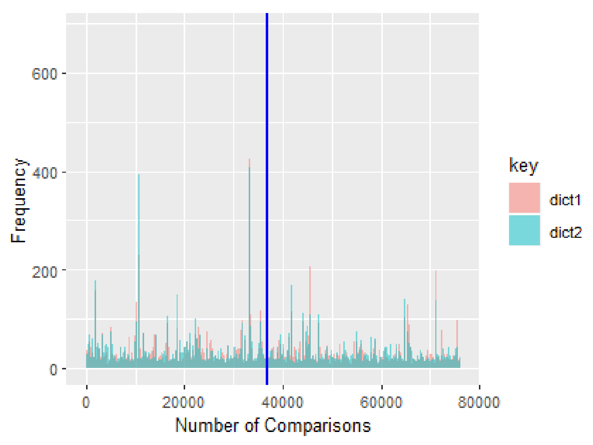


Fig 6: Athletes Data, Sorted Insertion



Figures 1-5 show that the number of comparisons for dict2 in each variation of data is distributed less than that of dict1 in some way. In Figures 1,2,4,5, the histogram distribution is shifted to the left. In Figures 3,4, the distribution is much more concentrated towards lower values than dict1, with higher peaks since the range of dict2 is less than dict1.

Figure 2 shows two almost identical uniformly distributed histograms, with dict2 minutely shifted to the left.

For Figure 6, there is no discernible difference between histogram of dict1 and dict2, as shown much the large overlap region and random peaks of both colours.

Table 1: Athletes	dict1	dict2

Grouped (by ID)	`Min.   :  1.00`
	`1st Qu.: 34.00`
	`Median : 43.00`
	`Mean(1): 45.67`
	`3rd Qu.: 54.00`
	`Max.   :229.00`
	`Mean(2): 45.67`
	`SD     : 18.29`
		`Min.   :  1.00`
		`1st Qu.: 22.00`
		`Median : 26.00`
		`Mean   : 28.54`
		`3rd Qu.: 31.00`
		`Max.   :221.00`
		`Mean(2): 28.54`
		`SD     : 14.10`

Random	`Min.   : 8.00`
	`1st Qu.:23.00`
	`Median :26.00`
	`Mean   :26.89`
	`3rd Qu.:31.00`
	`Max.   :76.00`
	`Mean(2):26.89`
	`SD     : 6.29`
		`Min.   : 1.00`
		`1st Qu.:18.00`
		`Median :22.00`
		`Mean   :21.85`
		`3rd Qu.:25.00`
		`Max.   :44.00`
		`Mean(2):21.85`
		`SD     : 5.38`

Sorted (Ascending)	`Min.   :    1`
	`1st Qu.:18539`
	`Median :35319`
	`Mean   :36726`
	`3rd Qu.:55489`
	`Max.   :76307`
	`Mean(2):36728`
	`SD     :21668`
		`Min.   :    1`
		`1st Qu.:18535`
		`Median :35315`
		`Mean   :36719`
		`3rd Qu.:55479`
		`Max.   :76305`
		`Mean(2):36728`
		`SD     :21668`

Table 2: Even Numbers	dict1	dict2

Random	`Min.   :10.00`
	`1st Qu.:15.00`
	`Median :17.00`
	`Mean   :17.82`
	`3rd Qu.:21.00`
	`Max.   :29.00`
	`Mean(2):17.82`
	`SD     : 3.78`
		`Min.   : 1.00`
		`1st Qu.: 9.00`
		`Median :12.00`
		`Mean   :12.16`
		`3rd Qu.:15.00`
		`Max.   :24.00`
		`Mean(2):12.16`
		`SD     : 3.89`

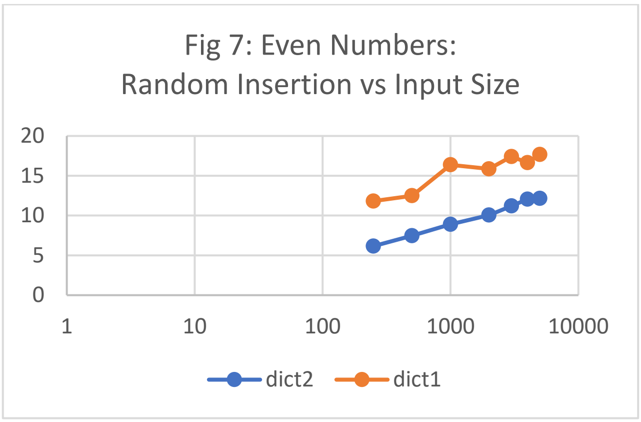
Sorted (Ascending)	`Min.   :   5.0`
	`1st Qu.: 254.8`
	`Median : 504.5`
	`Mean   : 504.5`
	`3rd Qu.: 754.2`
	`Max.   :1004.0`
	`Mean(2): 503.9`
	`SD     : 288.7`
		`Min.   :   1.0`
		`1st Qu.: 250.8`
		`Median : 500.5`
		`Mean   : 500.5`
		`3rd Qu.: 750.2`
		`Max.   :1000.0`
		`Mean(2): 500.7`
		`SD     : 288.7`

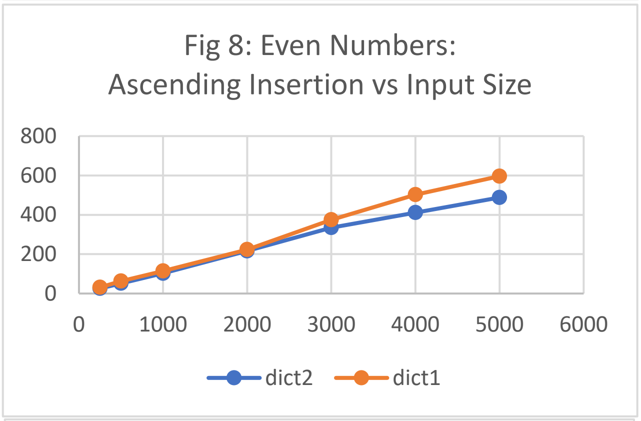
Sorted (Descending)	`Min.   :   6`
	`1st Qu.:1255`
	`Median :2504`
	`Mean   :2503`
	`3rd Qu.:3752`
	`Max.   :5000`
	`Mean(2):2508`
	`SD     :1444`
		`Min.   :   1.0`
		`1st Qu.: 250.8`
		`Median : 500.5`
		`Mean   : 500.5`
		`3rd Qu.: 750.2`
		`Max.   :1000.0`
		`Mean(2): 500.2`
		`SD     : 288.7`

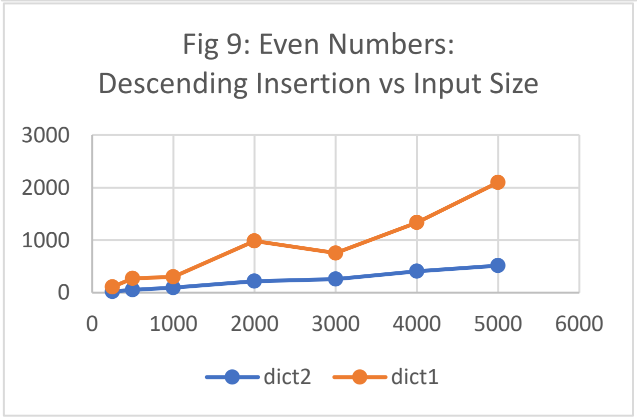
Shown by Table 1,2 and supported by Figures 1-6, is that the mean of number of comparisons for dict2 is at most dict1, for each varied dataset. In fact, the output showed that for a given key search in any of the datasets, the dict2 number of comparisons is 100% less than or equal to the dict1 number of comparisons.

The remainder of the summary statistics reinforce that dict2 is distributed with lower values than dict1. For both ascending variations however, they are distributed almost equally. In contrast, sorted descending insertion for "even numbers" had the number of comparisons for dict1 5 times greater than that of dict2.

In both Tables 1,2, random insertion has distribution with least number of comparisons, shown by smaller values and smaller standard deviation. Sorted insertion is greatest values, with table 2 showing that in the "even numbers" case, sorted descending distributed the greatest for dict1.







In Figures 8,9, number of comparisons in sorted insertion of "even numbers" shows linear growth with respect to sample size. In Figure 9, the multiplier of dict1 is apparent. In Figure 7, dict2 random insertion shows near perfect linear growth with respect to a logarithmic x-scale. Dict1 random insertion also shows a moderately strong linear growth, but with more variation and noise.

Discussion:

To start, the result that dict2 performs with lower (or equal) number of comparisons when insertion method and key is held constant is consistent with theory. When the key search function is called, in the case for dict2, the function ceases when it reaches the first time a match is found, as the rest of the key data must lie in the linked list that is in the node data. This contrasts to dict1, which, after a found match, must check the subtree that begins is rooted on the left leaf.

What was found was that the degree of benefit from the linked list structure of dict2 was significantly impacted by the insertion method, as well as the nature of the input. In ascending insertion, "even numbers" and "athletes" datasets showed that the difference in comparison number distribution between dict1 and dict2 is insignificant. However, this is due to the structure of the tree in this insertion method; it is a linked list of right nodes. The only left nodes will occur in dict1, which is precisely where the duplicate keys will be located. Thus, most of the comparisons are attributed to traversing through the linked list, and the number of comparisons that differs between dict1 and dict2 is insignificant due to the lower number of duplicates compared to number of unique names. Thus, this is dependent also on the nature of the dataset. It is most likely that a dataset with more duplicate keys than unique names will find that the benefit of dict2 will increase, although further experiments will be required to verify this. For descending insertion, the "even numbers" dataset showed that there was 5 times greater number of comparisons from dict1 to dict2. This is due to the 5 copies of each number, and since the BST is a linked list via the left node, each number in dict1 is 5 times deeper than the same number in the BST of dict2. However, if there were no duplicates, then the number of comparisons between dict1 and dict2 in this case will be equal. Again, we see that the dataset itself and how it is inserted are both important factors that determines the time complexity of the BST dictionary.

In a randomised order, dict2 tends to be significantly lower cost than dict1. This is likely due to the handling of duplicate keys once again. Consider when the search function reaches its first match. Regardless whether there is duplicate key, the search function of dict1 must traverse left, to search for potential duplicate keys. Unless the duplicate key appears again, the function must traverse to the rightmost leaf, since the key is larger than everything in that subtree, except for itself. This process will repeat if a match was found again. For dict2, the first match implies the search has ended.

Through Figures 7-9, we see that the theoretical time complexities are consistent with results, but not perfectly. This is due to the graph being dependent on randomisation in the samples, with the results of dict1 more erratic, due to the handling of duplicate keys in insertion and search, as mentioned before. For the random insertion case, both dictionaries exhibit the logarithmic ($O(\log n)$) tendencies, and for the sorted case, which are considered the worst case, does indeed exhibit linear ($O(n)$) growth. More samples can be taken and perhaps the average case will have stronger linearity in each of the scales. Perhaps it is interesting to note that initially, the first n points were chosen rather than a random sample of size n, to test for time complexity. However, due to the perfect, artificial nature of the data (5 copies of each unique key), the results were perfectly linear for the sorted insertion case. All in all, in the experimental results at least, sorted insertion is more costly than randomised insertion in search, as it tends towards a mean of $O(n)$ comparisons compared to $O(\log n)$ comparisons.

In terms of space complexity, `valgrind` showed that memory allocation was the same regardless which insertion method. The difference between the two dictionaries are insignificant for each of the datasets, with dict2 using marginally less memory. This is consistent with the $O(n)$ space-complexity of BSTs, as each node and its data require the same amount of memory, independent of location in the BST. Perhaps the difference in dict2 is due to linked list nodes requiring less memory, since there is only one pointer to the next element, compared to two for a BST node.

Conclusion:

Between dict1 and dict2, dict2 performs with less comparisons 100% of the time. The degree of difference is directly related to the number of duplicate keys the dataset has. The experimental results were consistent with the theoretical asymptotic time-complexity from the lectures, with sorted insertion being the worst case, appearing to have $O(n)$ behaviour, and random insertion appearing to have $O(\log n)$ behaviour.