COMP26120 Lab 3 Report

Junle Yu

9 December 2022

Experiment 1 Hash Set Performance with linear probing

Hypothesis The complexity for Hash Set with linear probing on random, sorted or reversed sorted input is O(1). Theoretically, the best case and the average case for Hash Set is O(1) for any type of input. In the implementation of hash set, a hash function is used to generate a hash value for every input value to be inserted into the specific location in the hash set, without collision occur ideally, so the complexity of O(1). As the input values fill up the hash table, collision is likely to occur, then linear probing loops through the hash set from the position of collision, to find an empty space nearby to insert the value. The looping of linear probing will increase the complexity, thus, a load factor is used to keep the linear function performs efficiently, if the load factor is equal to or higher than 0.75, the hash set is resized and rehashed. By keeping a certain proportion of empty spaces in the hash set, linear probing should be able to find an empty by just a few loops, which will not increase the complexity from O(1) to O(n). Therefore, the complexity for Hash Set is O(1).

Design To test the hypothesis, two sets of dictionaries are generated. The first set has 5 dictionaries of sizes 20K, 40K, 60K, 80K and 100K, with values in random order, which are used to test the insert performance of the Hash Set when the size of the dictionary is large. The second set has 5 dictionaries of sizes 200, 400, 600, 800, and 1000, with values in sorted order, which are used to test the insert performance of the Hash Set when the values are sorted and are compared with the Binary Search Tree. The size of dictionary is the independent variable in this experiment. The spellcheck_hashset program runs on each dictionary with an input file containing a single word that is not in the dictionary, in order to minimise the impact of the look-up time on the comparative performance on each dictionary. To eliminate random errors, each dictionary is run 5 times, with run time recorded 5 times, then the average run time for each dictionary is calculated and is used to plot graph later in this experiment. The dependent variable of this experiment is the time taken for the program to execute (approximately the same as the total time taken for the program to insert dictionary since the impact of look-up time is minimised), it is measured using the UNIX time command summing the user and sys values output by the command. Finally, a scatter graph with x-axis to be the size of dictionary and y-axis to be the time taken per insert is plotted for each set of dictionaries. The UNIX time command only provides the **total time** taken for inserting the dictionary to the hash set, which the time will be affected by the size of the dictionary. In order to investigate the complexity of the hash set, the time taken per insert needs to be calculated, so in the scatter graph, the value of **the total time taken for inserting a given dictionary / the size of this dictionary** is plotted on the y-axis. The line of best fit of the graph is generated with its formula in the form of y = m*x + c, which m is the gradient of the line, and c is a constant value. The formula of the line of best fit in each graph can be used to determine the complexity of the Hash Set program, if the gradient m equals to 0, it means the time taken to insert dictionary is not affected by the size of the dictionary, but the constant c, then the deduction can be made that the complexity of the Hash Set program is O(1). The process of generating dictionaries and computing the time was automated using the shell scripts shown in Appendix B.

Results The results are plotted using Matplotlib and Numpy and Numpy's polyfit function is used to generate the best fit line y = m*x + c and values for m and c are calculated. The results are shown in Figure 1(random dictionary) and Figure 2 (sorted dictionary), and the raw data can be seen in Appendix A. The line of best fit y = m*x + c has values of 0.00000 and 0.03731 computed for m and c respectively in Figure 1, and it has 0.00000 and 0.004975 computed for m and c respectively in Figure 2. This confirms the behaviour of hash set on sorted input and random input and allows us to predict the time taken to per insert as 0.00000*x + 0.03731 for random input and 0.00000*x + 0.04975 for sorted input. The value 0.00000 of m in the line of best fit for both graphs deduces that the size of the dictionary does not affect the time of inserting the dictionary to hash set, so the complexity of hash set is O(1).

Discussion The results in both graphs correctly state that the hypothesis has been confirmed which the complexity of hash set of O(1). The formula of the line of best fit in both graphs are quite similar, and they have the same complexity, so the order of the input data will not affect the complexity of hash set. The time in graphs is plotted in milliseconds, and the scale of y axis in both graphs is quite small (0.002ms for Figure 1 and 0.0005ms for Figure 2), which makes the best fit line look visually not horizontal. However, such small scale over-exaggerates the visualisation of the gradient of the best fit line, which might give an illusion that the size of dictionary has an impact on the time taken per insert. Figure 3 shows the same plot as Figure 1 with a larger scale (0.005ms) on y-axis, and now the best fit line looks more horizontal. If we adjust the scale of y-axis to a more reasonable size (eg. 1ms), the line of best fit will be shown as horizontal. The value 0.00000 of m in the best fit line again confirms the hypothesis, which the complexity of hash set is O(1).

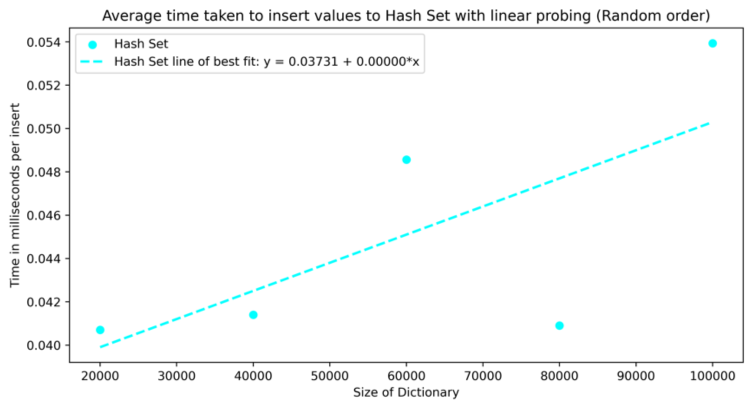


Figure 1: Time taken to insert values from a random dictionary to hash set, with best fit line

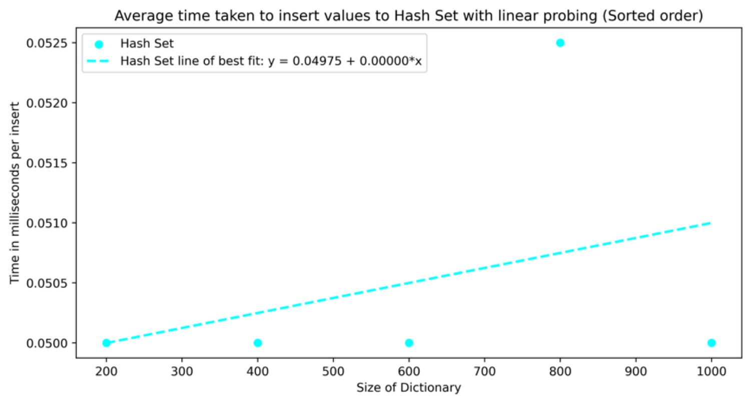


Figure 2: Time taken to insert values from a sorted dictionary to hash set, with best fit line

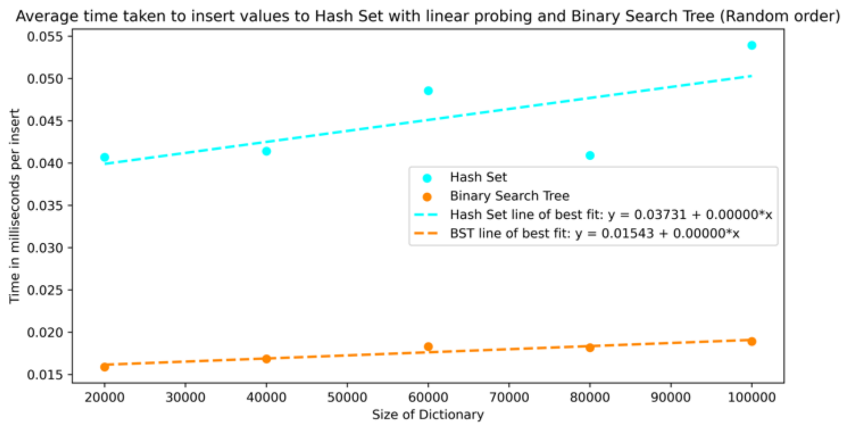


Figure 3: Time taken to insert values from a random dictionary to hash set, same as Figure 1

Experiment 2 Binary Search Tree Performance

Theoretical Average Case

Hypothesis The complexity of Binary Search Tree on random input (theoretical average case) is O(log n). In the implementation of binary search tree, when we insert a value, if the value is greater than the value of the root node, it goes to the right subtree else it goes to the left subtree, it then recursively compares with the value of the root node of the subtree it is in, until there is no more node to compare with. Simple binary search tree has no height balance property, in average case, the order of the data is not determined, the data might be well organised so they can construct a binary search tree that its left subtree and right subtree have approximately the same height. When inserting a value, only several comparisons with complexity O(1). Oppositely, there might be consecutive data that are greater or smaller than one another, so the height of the left subtree and the right subtree is not balanced, when inserting a value, it might need to compare with all existing nodes in the tree, the complexity is therefore O(n). Average case might include both scenarios, therefore the complexity is O(log n).

Design To test the hypothesis, 5 dictionaries of sizes 20K, 40K, 60K, 80K and 100K are generated, with values in random order, which are used to test the insert performance of the Binary Search Tree when the size of the dictionary is large and to compare with the performance of Hash Set. The size of dictionary is the independent variable in this experiment. The spellcheck_bstree program runs on each dictionary with an input file containing a single word that is not in the dictionary, in order to minimise the impact of the look-up time on the comparative performance on each dictionary. To eliminate random errors, each dictionary is run 5 times, with run time recorded 5 times, then the average run time for each dictionary is calculated and is used to plot graph later in this experiment. The dependent variable of this experiment is the time taken for the program to execute (approximately the same as the total time taken for the program to insert dictionary since the impact of look-up time is minimised), it is measured using the UNIX time command summing the user and sys values output by the command. Finally, a scatter graph with x-axis to be the size of dictionary and y-axis to be the time taken per insert is plotted for each set of dictionaries. The UNIX time command only provides the **total time** taken for inserting the dictionary to the binary search tree, which the time will be affected by the size of the dictionary. In order to investigate the complexity of the binary search tree, the time taken per insert needs to be calculated, so in the scatter graph, the value of **the total time taken for inserting a given dictionary / the size of this dictionary** is plotted on the y-axis. The line of best fit of the graph is generated, if the line of best fit shows the pattern of O(log n), then the deduction can be made that the complexity of the binary search tree program is O(log n). The process of generating dictionaries and computing the time was automated using the shell scripts shown in Appendix D.

Results The results are plotted as a scatter graph with x-axis to be the size of dictionary and y-axis to be the time taken per insert is plotted for each set of dictionaries using Matplotlib and Numpy. Numpy's polyfit function is used to generate the best fit line of this scatter diagram. The best fit line that matches the scatter diagram is in the form of y = m*x + c and values for m and c are calculated. The results are shown in Figure 4, and the raw data can be seen in Appendix C. The line of best fit y = m*x + c has values of 0.00000 and 0.01543 computed for m and c. The value 0.00000 of m in the line of best fit deduces that the size of the dictionary does not affect the time of inserting the dictionary to binary search tree, so the complexity of binary search tree is O(1). However, this disproves the behaviour of binary search tree on random input since the hypothesis of the complexity on random input is O(log n), and the results show that the complexity is O(1).

Discussion Based on the results of the graph, the complexity of binary search tree when inserting a random dictionary is O(1), which disproves the hypothesis. Although this situation should not happen, as we discussed in hypothesis, when a random dictionary is generated, the order of values is not determined, in this case, there might be values ordered in a way that each value is greater than the one above it and less than the one below it, so when we insert a value, a few of comparisons need to be made only, which the complexity becomes O(1). Since the random dictionary is generated by a program, this is a random error that is difficult to prevent. Although the results generated do not satisfy the hypothesis, it is still a valid answer based on the input data we have. In addition, Figure 5 shows the same plot as Figure 4 with a larger scale (0.005ms) on y-axis, and now the best fit line looks more horizontal. The value 0.00000 of m in the best fit line again confirms the hypothesis, which the complexity of binary search tree with random dictionary is O(1).

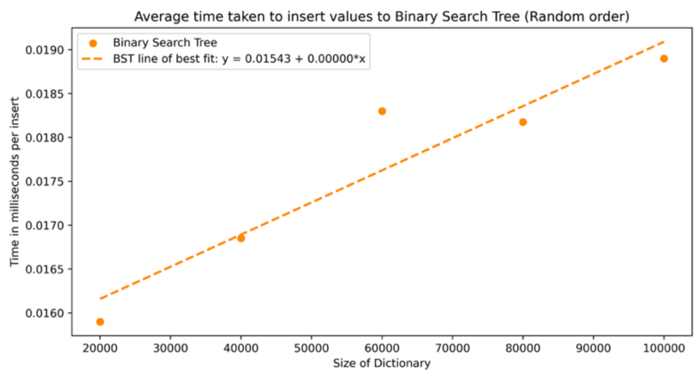


Figure 4: Time taken to insert values from a random dictionary to binary search tree, with best fit line

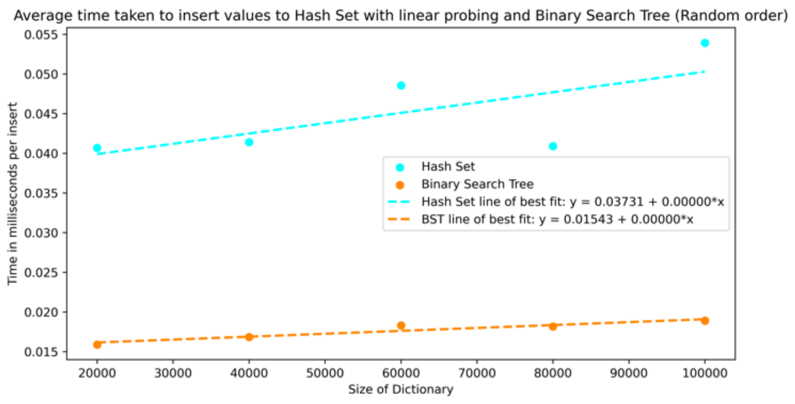


Figure 5: Time taken to insert values from a random dictionary to binary search tree, same as Figure 3

Theoretical Worst Case

Hypothesis The complexity of Binary Search Tree on sorted input (theoretical worst case) is O(n). In the implementation of binary search tree, when we insert a value, if the value is greater than the value of the root node, it goes to the right subtree else it goes to the left subtree, it then recursively compares with the value of the root node of the subtree it is in, until there is no more node to compare with. Simple binary search tree has no height balance property. In worst case, when inserting a value in the binary search tree, if the dataset is in descending order, the value will always be inserted to the left subtree; if the dataset is in ascending order, the value will always be inserted to the right subtree. The binary search tree will end up become a linked list, when inserting a new value in the tree, the new value must compare with all the existing nodes in the tree. Therefore, the complexity of Binary Search Tree on sorted input (theoretical worst case) is O(n).

Design If we insert an ordered list to a binary search tree, as we discussed above, the value will always be inserted to the right subtree or always be inserted to the left subtree, depends on how the data are ordered in the dictionary. Python limits the number of recursive calls on the right subtree and left subtree to 1000 times, so in this experiment the size of the dictionary cannot be greater than 1000. To test the hypothesis, 5 dictionaries of sizes 200, 400, 600, 800 and 1000 are generated, the values are in ascending order. The size of dictionary is the independent variable in this experiment. The spellcheck_bstree program runs on each dictionary with an input file containing a single word that is not in the dictionary, in order to minimise the impact of the look-up time on the comparative performance on each dictionary. To eliminate random errors, each dictionary is run 5 times, with run time recorded 5 times, then the average run time for each dictionary is calculated and is used to plot graph later in this experiment. The dependent variable of this experiment is the time taken for the program to execute (approximately the same as the total time taken for the program to insert dictionary since the impact of look-up time is minimised), it is measured using the UNIX time command summing the user and sys values output by the command. Finally, a scatter graph with x-axis to be the size of dictionary and y-axis to be the time taken per insert is plotted for each set of dictionaries. The UNIX time command only provides the **total time** taken for inserting the dictionary to the binary search tree, which the time will be affected by the size of the dictionary. In order to investigate the complexity of the binary search tree, the time taken per insert needs to be calculated, so in the scatter graph, the value of **the total time taken for inserting a given dictionary / the size of this dictionary** is plotted on the y-axis. The line of best fit of the graph is generated with its formula in the form of y = m*x + c, which m is the gradient of the line, and c is a constant value. The formula of the line of best fit in each graph can be used to determine the complexity of the Binary Search Tree program, if the gradient m does not equal to 0, it means the time taken to insert dictionary is affected by the size of the dictionary, then the deduction can be made that the complexity of the Binary Search Tree program is O(n). The process of generating dictionaries and computing the time was automated using the shell scripts shown in Appendix D.

Results The results are plotted using Matplotlib and Numpy and Numpy's polyfit function is used to generate the best fit line y = m*x + c and values for m and c are calculated. The results are shown in Figure 6, and the raw data can be seen in Appendix C. The line of best fit y = m*x + c has values of 0.00015 and 0.05883 computed for m and c respectively. This confirms the behaviour of binary search tree on sorted input and allows us to predict the time taken to per insert as 0.00015*x + 0.05883. The value 0.00015 of m in the line of best fit deduces that the size of the dictionary affects the time of inserting the dictionary to binary search tree, so the complexity of binary search tree given a sorted dictionary is O(n).

Discussion The results of the graph correctly state that the hypothesis which is the time complexity of binary search tree when inserting an ordered dictionary is O(n). The formula of the line of best fit y = 0.05883 + 0.00015*x indicates that as the size of dictionary increases by 1, the time in milliseconds per insert increases by 0.00015, so the size of dictionary will affect the time per insert. This is because the dictionary is ordered, all the value will always be inserted to the right subtree or always be inserted to the left subtree, depends on how the data are ordered in the dataset. The binary search tree becomes a linked list, as the size of the dictionary increases, more values need to be inserted and they need to compare with all the existing values in the tree. The values at the back of the dictionary will have more comparisons than the values at the front, so time per insert increases as more values are inserted in the binary search tree. The last value of the dictionary needs to make (size of dictionary – 1) comparisons in its insertion, the complexity is therefore O(n).

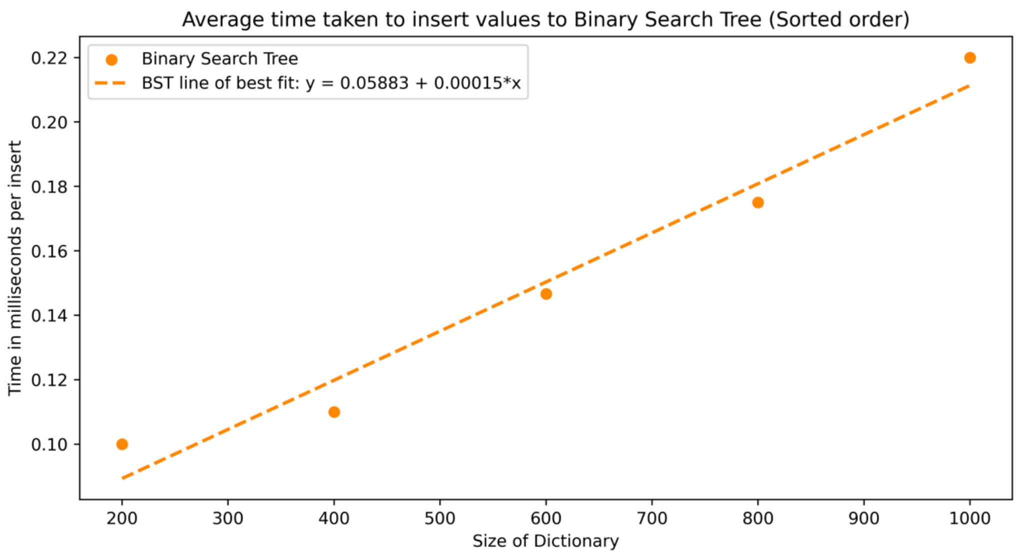


Figure 6: Time taken to insert values from an ordered dictionary to binary search tree, with best fit line

Conclusion

First of all, in experiment 1 we deduce that the complexity of Hash Set when inserting a dictionary is O(1), this applies to ordered dictionary and random dictionary. In experiment 2 average case analysis, in theory we expected that complexity of Binary Search Tree when inserting a dictionary which its values are randomly ordered is O(log n), but in reality we get complexity of O(1). Since the complexity of Hash Set and Binary Search Tree are the same when the values in dictionary are randomly ordered, we can use Figure 7 to determine which one is better to use. In Figure 7, the best fit line of Hash Set is above the best fit line of Binary Search Tree, which shows Hash Set takes longer to insert than Binary Search Tree. Hash Set on average needs at least 0.04 milliseconds per insert, while Binary Search Tree only needs 0.018 milliseconds per insert. For the same size of dictionary, Hash Set on average needs 0.025 milliseconds more to insert one value. Therefore, Binary Search Tree performs better than Hash Set for random dictionary.

Secondly, in experiment 2 worst case analysis, in theory we expected the complexity of Binary Search Tree when inserting a dictionary which its values are sorted is O(n), and we get the same answer. In terms of complexity, for sorted dictionary, Hash Set has complexity O(1), Binary Search Tree has complexity O(n), Hash Set is better. In Figure 8, the best fit line of Binary Search Tree is above the best fit line of Hash Set, which shows Binary Search Tree takes longer to insert than Hash Set. When the dictionary is sorted, for every size of the dictionary, Hash Set almost takes the constant time (0.05ms) per insert as its line of best fit is a horizontal line. The time taken per insert for Binary Search Tree increases as the size of dictionary increases, when the size of dictionary is 1000, Binary Search Tree takes 5 times longer than Hash Set to insert a value. Therefore, Hash Set Tree performs better than Binary Search Tree for sorted dictionary.

In conclusion, if the values in dictionary are randomly ordered, using Binary Search Tree will have a shorter insertion time; if the values in dictionary are ordered, using Hash Set will have a shorter insertion time.

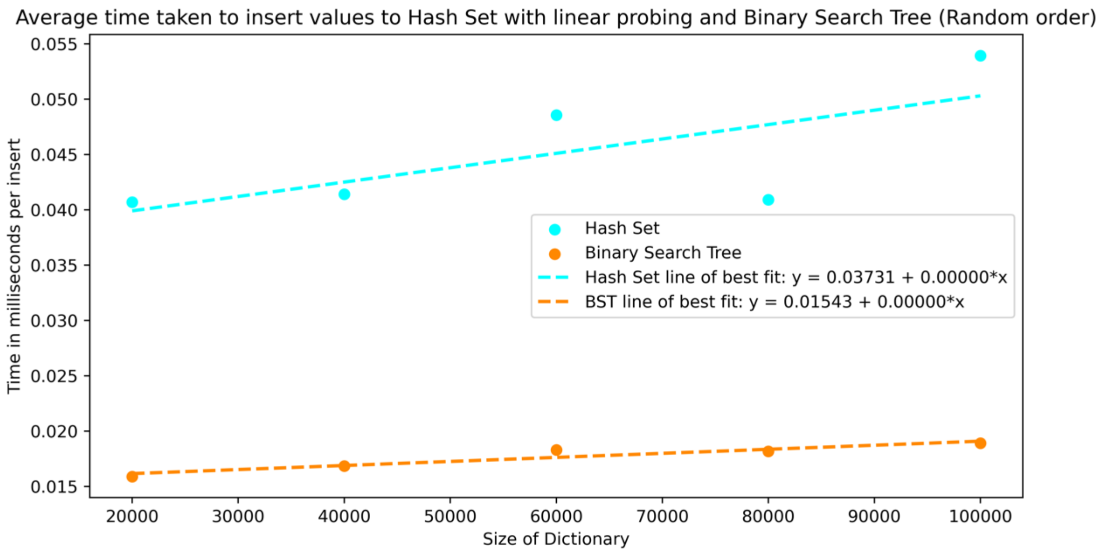


Figure 7: Graph compares the time taken to insert values from a random dictionary to Hash

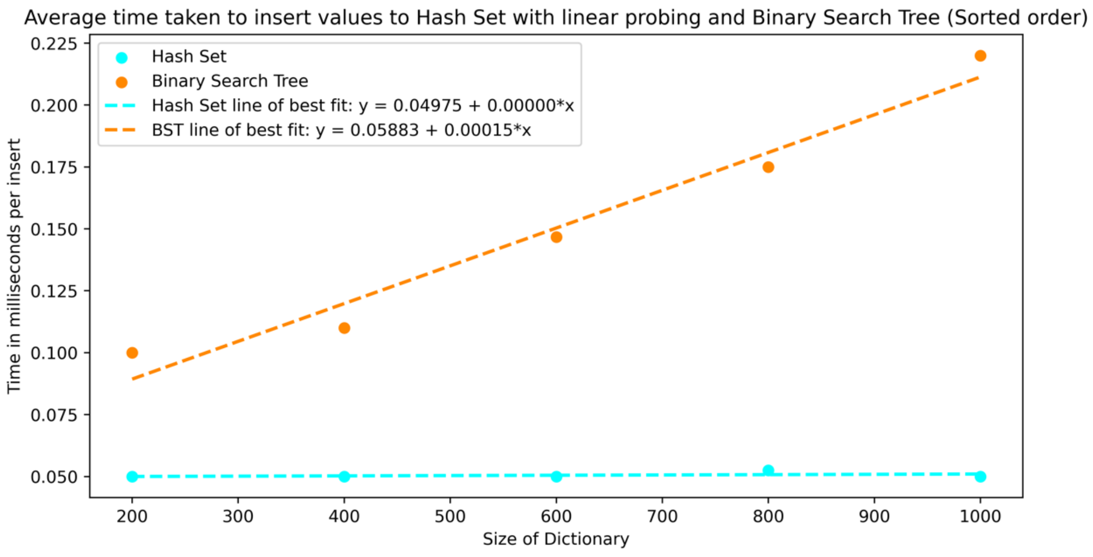


Figure 8: Graph compares the time taken to insert values from an ordered dictionary to Hash

Experiment 3 Hash Set with quadratic probing Performance

Appendix

A Raw Data for Experiment 1

Total time of insert recorded for HashSet with linear probing (Random dictionary)

	Size of Dictionary	Time(s)
0	20000	0.79
1	20000	0.82
2	20000	0.82
3	20000	0.79
4	20000	0.85
5	40000	1.66
6	40000	1.68
7	40000	1.69
8	40000	1.66
9	40000	1.59
10	60000	2.83
11	60000	3.05
12	60000	2.88
13	60000	2.93
14	60000	2.88
15	80000	3.36
16	80000	3.22
17	80000	3.24
18	80000	3.20
19	80000	3.34
20	100000	5.45
21	100000	5.27
22	100000	5.39
23	100000	5.45
24	100000	5.41

Total time of insert recorded for HashSet with linear probing (Sorted dictionary)

	Size of Dictionary	Time(s)
0	200	0.01
1	200	0.01
2	200	0.01
3	200	0.01
4	200	0.01
5	400	0.02
6	400	0.02
7	400	0.02
8	400	0.02
9	400	0.02
10	600	0.03
11	600	0.03
12	600	0.03
13	600	0.03
14	600	0.03
15	800	0.04
16	800	0.04
17	800	0.05
18	800	0.04
19	800	0.04
20	1000	0.05
21	1000	0.05
22	1000	0.05
23	1000	0.05
24	1000	0.05

B Shell Scripts for Experiment 1

B.1 Generating Dictionaries

Command Line

Random

1. python3 generate.py random dict_200 query_200 200 1 random 0
2. python3 generate.py random dict_400 query_400 400 1 random 0
3. python3 generate.py random dict_600 query_600 600 1 random 0
4. python3 generate.py random dict_800 query_800 800 1 random 0
5. python3 generate.py random dict_1000 query_1000 10000 1 random 0
6. python3 generate.py random dict_20000 query_20000 20000 1 random 0
7. python3 generate.py random dict_40000 query_40000 40000 1 random 0
8. python3 generate.py random dict_60000 query_60000 60000 1 random 0
9. python3 generate.py random dict_80000 query_80000 80000 1 random 0
10. python3 generate.py random dict_100000 query_100000 100000 1 random 0

Sorted

11. python3 generate.py random dict_200 query_200 200 1 sorted 0
12. python3 generate.py random dict_400 query_400 400 1 sorted 0
13. python3 generate.py random dict_600 query_600 600 1 sorted 0
14. python3 generate.py random dict_800 query_800 800 1 sorted 0
15. python3 generate.py random dict_1000 query_1000 10000 1 sorted 0
16. python3 generate.py random dict_20000 query_20000 20000 1 sorted 0
17. python3 generate.py random dict_40000 query_40000 40000 1 sorted 0
18. python3 generate.py random dict_60000 query_60000 60000 1 sorted 0
19. python3 generate.py random dict_80000 query_80000 80000 1 sorted 0
20. python3 generate.py random dict_100000 query_100000 100000 1 sorted 0

B.2 Script for Computing Run Times

Sorted

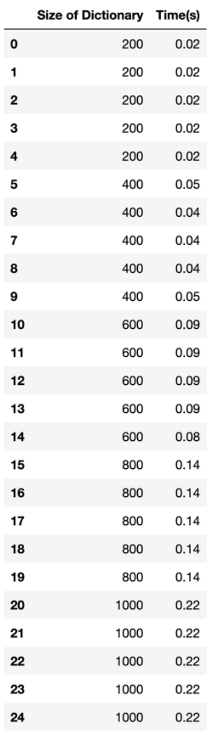
```
SIZES="20000 40000 60000 80000 100000"

for SIZE in $SIZES
do

for COUNT in 1 2 3 4 5
do

    # Debugging command to check program call actually works
    python3 ./python/speller_hashset.py -d ./sorted/dict_${SIZE} -m
4 ./sorted/query_${SIZE}

    ALL_TIME=`(time -p python3 ./python/speller_hashset.py -
d ./sorted/dict_${SIZE} -m 4 ./sorted/query_${SIZE}) 2>&1 | grep -E "user
|sys " | sed s/[a-z]//g`

    RUNTIME=0
    for i in $ALL_TIME;
    do RUNTIME=`echo $RUNTIME + $i|bc`;
    done

    echo $SIZE $RUNTIME >> data/sorted/data_hashset_python.dat
    echo $SIZE, $RUNTIME >> data/sorted/data_hashset_python.csv

done

done
```

Random

```
SIZES="20000 40000 60000 80000 100000"

for SIZE in $SIZES
do

for COUNT in 1 2 3 4 5
do

    # Debugging command to check program call actually works
    python3 ./python/speller_hashset.py -d ./Random/dict_${SIZE} -m
4 ./sorted/query_${SIZE}

    ALL_TIME=`(time -p python3 ./python/speller_hashset.py -d ./Random
/dict_${SIZE} -m 4 ./Random/query_${SIZE}) 2>&1 | grep -E "user |sys " |
sed s/[a-z]//g`

    RUNTIME=0
    for i in $ALL_TIME;
    do RUNTIME=`echo $RUNTIME + $i|bc`;
    done

    echo $SIZE $RUNTIME >> data/ Random /data_hashset_python.dat
    echo $SIZE, $RUNTIME >> data/ Random /data_hashset_python.csv

done

done
```

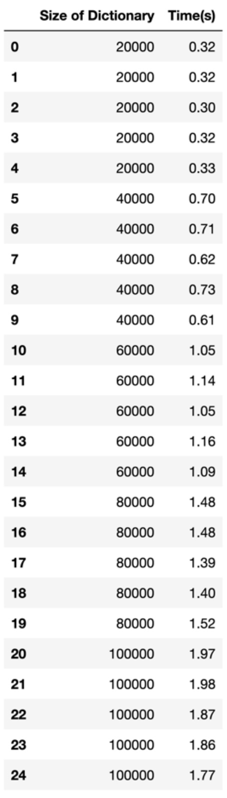
C Raw Data for Experiment 2

Total time of insert recorded for Binary Search Tree (Sorted dictionary)



	Size of Dictionary	Time(s)
0	200	0.02
1	200	0.02
2	200	0.02
3	200	0.02
4	200	0.02
5	400	0.05
6	400	0.04
7	400	0.04
8	400	0.04
9	400	0.05
10	600	0.09
11	600	0.09
12	600	0.09
13	600	0.09
14	600	0.08
15	800	0.14
16	800	0.14
17	800	0.14
18	800	0.14
19	800	0.14
20	1000	0.22
21	1000	0.22
22	1000	0.22
23	1000	0.22
24	1000	0.22

Total time of insert recorded for Binary Search Tree (Random dictionary)



	Size of Dictionary	Time(s)
0	20000	0.32
1	20000	0.32
2	20000	0.30
3	20000	0.32
4	20000	0.33
5	40000	0.70
6	40000	0.71
7	40000	0.62
8	40000	0.73
9	40000	0.61
10	60000	1.05
11	60000	1.14
12	60000	1.05
13	60000	1.16
14	60000	1.09
15	80000	1.48
16	80000	1.48
17	80000	1.39
18	80000	1.40
19	80000	1.52
20	100000	1.97
21	100000	1.98
22	100000	1.87
23	100000	1.86
24	100000	1.77

D Shell Scripts for Experiment 2

D.1 Generating Dictionaries

Command Line

Random

1. python3 generate.py random dict_200 query_200 200 1 random 0
2. python3 generate.py random dict_400 query_400 400 1 random 0
3. python3 generate.py random dict_600 query_600 600 1 random 0
4. python3 generate.py random dict_800 query_800 800 1 random 0
5. python3 generate.py random dict_1000 query_1000 10000 1 random 0
6. python3 generate.py random dict_20000 query_20000 20000 1 random 0
7. python3 generate.py random dict_40000 query_40000 40000 1 random 0
8. python3 generate.py random dict_60000 query_60000 60000 1 random 0
9. python3 generate.py random dict_80000 query_80000 80000 1 random 0
10. python3 generate.py random dict_100000 query_100000 100000 1 random 0

Sorted

11. python3 generate.py random dict_200 query_200 200 1 sorted 0
12. python3 generate.py random dict_400 query_400 400 1 sorted 0
13. python3 generate.py random dict_600 query_600 600 1 sorted 0
14. python3 generate.py random dict_800 query_800 800 1 sorted 0
15. python3 generate.py random dict_1000 query_1000 10000 1 sorted 0
16. python3 generate.py random dict_20000 query_20000 20000 1 sorted 0
17. python3 generate.py random dict_40000 query_40000 40000 1 sorted 0
18. python3 generate.py random dict_60000 query_60000 60000 1 sorted 0
19. python3 generate.py random dict_80000 query_80000 80000 1 sorted 0
20. python3 generate.py random dict_100000 query_100000 100000 1 sorted 0

D.2 Script for Computing Run Times

Sorted

```
SIZES="20000 40000 60000 80000 100000"

for SIZE in $SIZES
do

for COUNT in 1 2 3 4 5
do

    # Debugging command to check program call actually works
    python3 ./python/speller_bstree.py -d ./sorted/dict_${SIZE} -m
4 ./sorted/query_${SIZE}

    ALL_TIME=`(time -p python3 ./python/speller_bstree.py -
d ./sorted/dict_${SIZE} -m 4 ./sorted/query_${SIZE}) 2>&1 | grep -E "user
|sys " | sed s/[a-z]//g`

    RUNTIME=0
    for i in $ALL_TIME;
    do RUNTIME=`echo $RUNTIME + $i|bc`;
    done
    echo $SIZE $RUNTIME >> data/sorted/data_bstree_python.dat
    echo $SIZE, $RUNTIME >> data/sorted/data_bstree_python.csv

done

done
```

Random

```
SIZES="20000 40000 60000 80000 100000"

for SIZE in $SIZES
do

for COUNT in 1 2 3 4 5
do

    # Debugging command to check program call actually works
    python3 ./python/speller_bstree.py -d ./Random/dict_${SIZE} -m
4 ./Random/query_${SIZE}

    ALL_TIME=`(time -p python3 ./python/speller_bstree.py -d ./Random
/dict_${SIZE} -m 4 ./Random/query_${SIZE}) 2>&1 | grep -E "user |sys " |
sed s/[a-z]//g`

    RUNTIME=0
    for i in $ALL_TIME;
    do RUNTIME=`echo $RUNTIME + $i|bc`;
    done

    echo $SIZE $RUNTIME >> data/ Random /data_bstree_python.dat
    echo $SIZE, $RUNTIME >> data/ Random /data_bstree_python.csv

done

done
```