COMP26120 Lab 3 Report

Ayush Gupta

1 Experiment 1

Hypothesis The complexity of inserting a single value into my implementation of a hash set is O(1).

In theory, average case complexity of inserting an element in a hashset is O(1) which suggests constant time insertion for each word, and the observed linear relationship between the time and sample size, suggests that while the cumulative time for inserting multiple words follows a linear trend (O(n)) as the sample size grows, the amortized cost of inserting each individual word remains relatively constant.

Experimental Design To investigate the hypothesis regarding the insertion time of a hash set, the experimental design involved generating hash sets from randomly generated data sample of various sizes (sample size as our independent variable), including 50K, 100K, 150K, 200K, 250K, 300K, 350K, 400K, 450K and 500K items . Our dependent variable was the time taken to perform the insertion. Note, the query file remained empty to isolate the insertion time and avoid additional overhead associated with querying.

The speller_hashset program was executed thrice on each set size, and the average time was taken for the program to execute was measured using the UNIX time command. The sum of user and sys values output by the command provided an accurate representation of the overall execution time.

To maintain consistency and take into account the resizing factors affecting the running time, the initial size of the dynamic array was kept fixed at 509 along with the load-factor 0.7.

Following the execution of the hash set program on each dataset, the results were plotted using Matplotlib. Subsequently, a best-fit line was calculated for the function $f(x) = mx+q$ (because amortized time for inserting a word takes O(1)). The polyfit functionality in matplotlib was employed to determine values for the slope (m) and y-intercept (q) of the best-fit line. This analysis aimed to provide insights into the relationship between the size of the hash set and the corresponding insertion time, facilitating a quantitative evaluation of the hypothesis.

Results

Figure 1 clearly shows that the observations, obtained from plotting different times for different sample sizes, aligns with a linear curve 'y = mx + c', with values for 'm' and 'c' as 0.00000175 and 0.16006667 respectively.

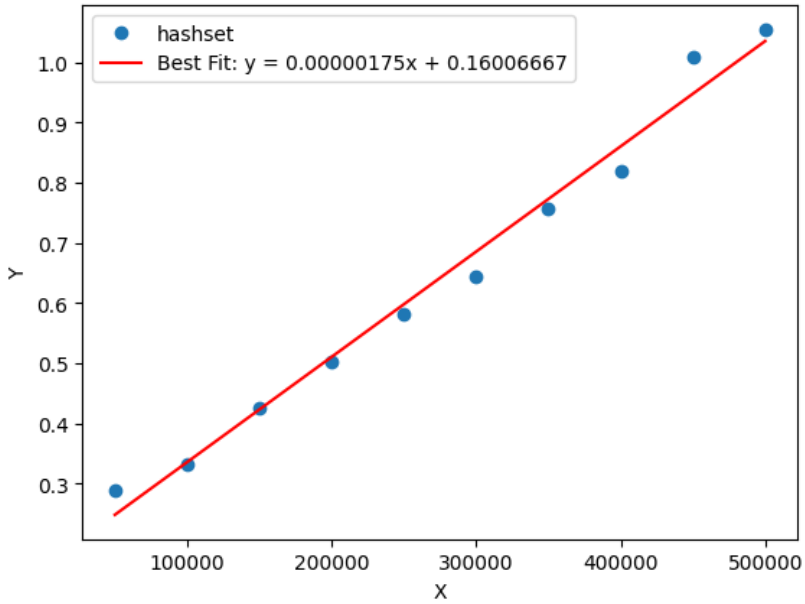

Figure 1: Graph showing the Best Fit Polynomial Curve

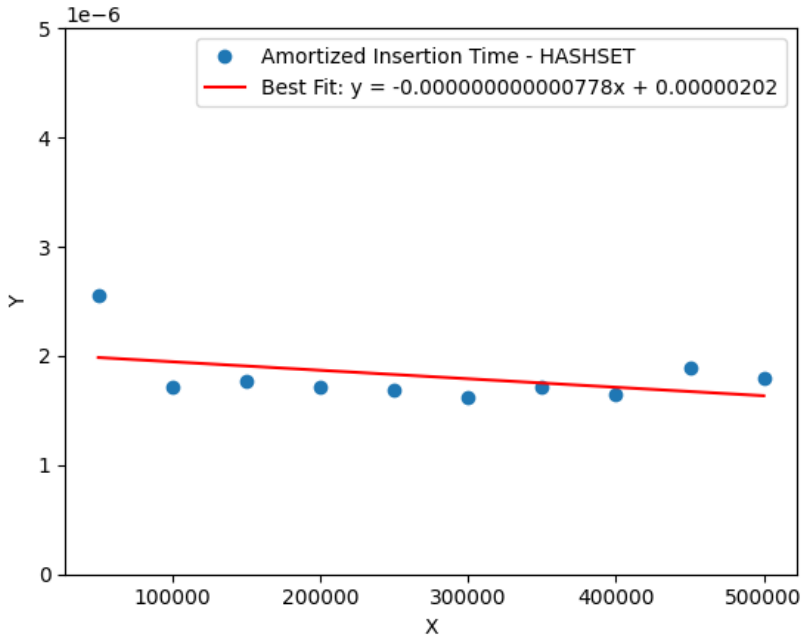

Figure 2: Graph showing the Best Fit Polynomial Curve For Amortized Time

Figure 1 and 2 together confirms the hypothesis as cumulative time for inserting multiple words follows a linear trend (O(n)), as the sample size grows, the amortized cost of inserting each individual word remains relatively constant (O(1)).

2 Experiment 2

Hypothesis The behaviour of insert for binary tree with unsorted input is O(n$log(n)$) for inserting 'n' number of words with O(log(n)) being the amortized time complexity for inserting single word.

In theory, average case complexity of inserting an element in a binary tree is O(log(n)) which suggests at most log(n) time insertion for each word, and while the the amortized time to insert each individual word remains relatively equal to log(n)),the asymptotic relationship between the time and sample size, the cumulative time for inserting 'n' words, follows a nearly O(n*log(n)) trend as the sample size grows.

Experimental Design To investigate the hypothesis regarding the insertion time of a binary tree, the experimental design involved generating hash sets of randomly generated data sample of various sizes (sample size as our independent variable), including 50K, 100K, 150K, 200K, 250K, 300K, 350K, 400K, 450K and 500K items. Our dependent variable was the time taken to perform the insertion. Note, the query file remained empty to isolate the insertion time and avoid additional overhead associated with querying.

The speller_bstree program was executed once on each sample size, and the time taken for the program to execute was measured using the UNIX time command. The sum of user and sys values output by the command provided an accurate representation of the overall execution time.

Following the execution of the binary tree program on each dataset, the results were plotted using Matplotlib. Subsequently, a best-fit line was calculated for the function $f(x) = p * n * \log(n) + q$ (because the amortized time to insert each individual word remains relatively equal to (log(n).). The polyfit functionality in matplotlib was employed to determine values for the constants 'a' and 'b' of the best-fit line. This analysis aimed to provide insights into the relationship between the size of the binary tree and the corresponding insertion time, facilitating a quantitative evaluation of the hypothesis.

Results

Figure 3 clearly shows that the observations, obtained from plotting different times for different sample sizes, aligns with a curve of the form 'y = a*x*log(x) + b', with values for 'a' and 'b' as 0.00000009 and 0.19016380 respectively.

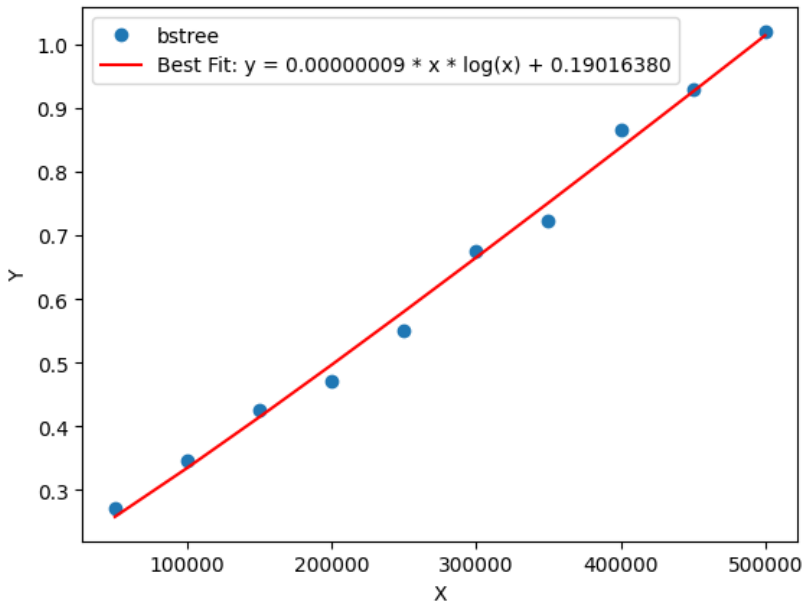

Figure 3: Graph showing the Best Fit Polynomial Curve

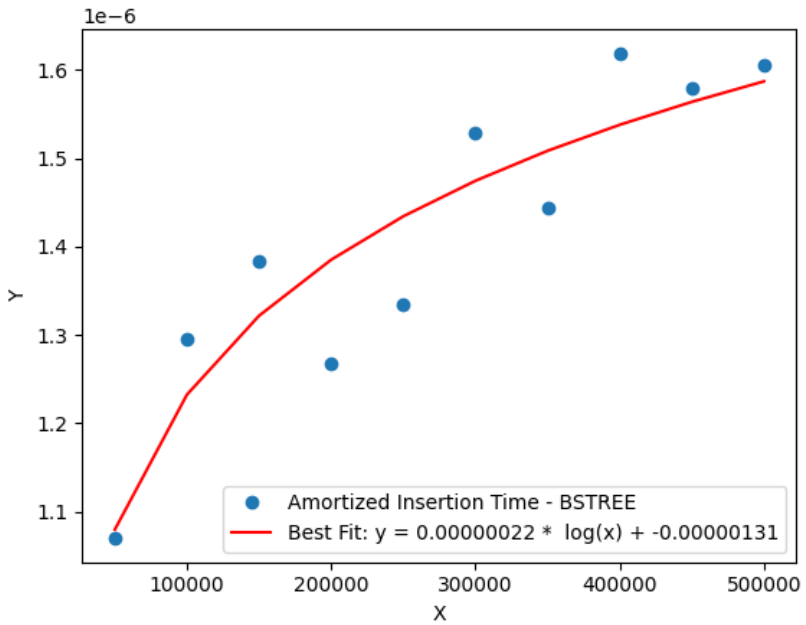

Figure 4: Graph showing the Best Fit Polynomial Curve For Amortized Time

Figure 3 and 4 together confirms the hypothesis as cumulative time for inserting multiple words follows a n$log(n)$) trend as the sample size grows, the amortized cost of inserting each individual word remains relatively equal to log(n)).

3 Conclusion

Figure 5 distinctly illustrates the asymptotic growth of the binary search tree (`binary tree`) surpassing that of the hash set (`hashset`). This observation aligns with the hypothesis that insertion in `binary tree` exhibits an asymptotic time complexity of $O(n \log n)$, contrasting the time complexity of O(n) for `hashset` insertion. The experimental findings, as depicted in Figure 5, reveal that `binary tree` surpasses `hashset` in terms of time taken after reaching a sample size of approximately 37.5K words.

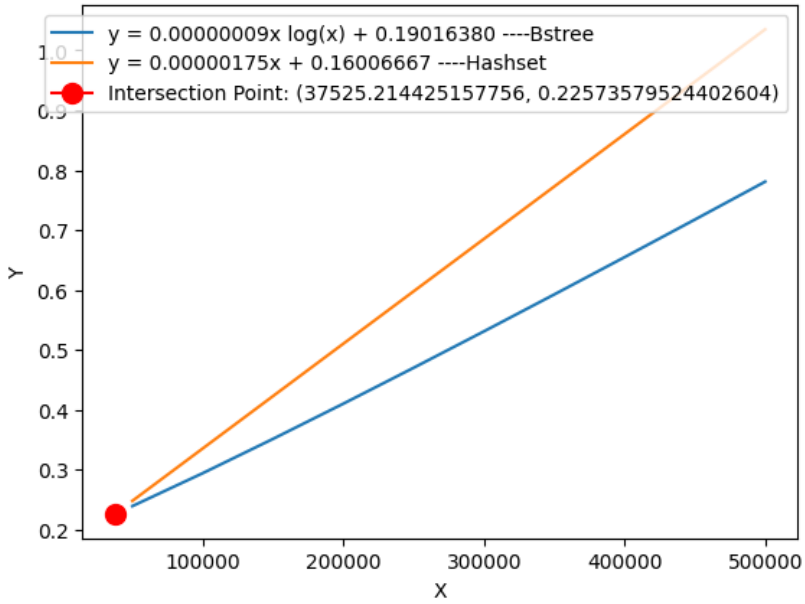

Figure 5: Graph showing the Cross-Over Point between the time taken by binary tree and hashset implementations, depicting the hashset becoming more efficient eventually after the intersection point

It is also important to note that the given cross-over point is specific to the used experimental setup, involving constraints like a dictionary containing all the words of same length, initial size of hashset, load-factor etc. Also in reality it also depends on machine, implementation detail, and other considerations. Although one thing can surely be concluded from any given experiment such as this, that all these factors do not affect the asymptotic nature of the both implementations. So it can clearly be concluded that based on this experimental results, hashset implementation is asymptotically more efficient than binary tree implementation.

4 Data Statement

The raw data for Experiment 1 is given in Appendix A. The process of generating dictionaries and computing the time for this experiment was done using the shell scripts given in Appendix B. The raw data for Experiment 2 is given in Appendix C. The process of generating dictionaries and computing the time for this experiment was done using the shell scripts given in Appendix D.

A Raw Data for Experiment 1

Unsorted Data	
Size	Time (s)
50000	0.288
50000	0.289
50000	0.287
100000	0.320
100000	0.331
100000	0.345
150000	0.420
150000	0.419
150000	0.436
200000	0.507
200000	0.510
200000	0.492
250000	0.582
250000	0.588
250000	0.576
300000	0.651
300000	0.654
300000	0.630
350000	0.761
350000	0.760
350000	0.753
400000	0.827
400000	0.833
400000	0.800
450000	1.013
450000	1.011
450000	1.003
500000	1.085
500000	1.069
500000	1.011

B Shell Commands for Experiment 1

B.1 Generating Dictionaries

```
python random_strings.py > dict
python generate.py dict test_data/${SIZE}/1/dict
```

```
test_data/${SIZE}/1/query random 0 10
```

B.2 Computing Run Times

```
time java comp26120.speller_hashset -d ./data/test_data/${SIZE}/
1/dict -m 0 -vv ../data/test_data/${SIZE}/1/query
```

C Raw Data for Experiment 2

Unsorted Data	
Size	Time (s)
50000	0.271
50000	0.273
50000	0.269
100000	0.368
100000	0.343
100000	0.330
150000	0.420
150000	0.419
150000	0.436
200000	0.487
200000	0.493
200000	0.433
250000	0.559
250000	0.568
250000	0.526
300000	0.676
300000	0.681
300000	0.671
350000	0.704
350000	0.712
350000	0.753
400000	0.892
400000	0.898
400000	0.805
450000	0.931
450000	0.934
450000	0.919
500000	1.037
500000	1.020
500000	1.003

D Shell Commands for Experiment 2

D.1 Generating Dictionaries

```
python random_strings.py > dict
python generate.py dict.txt test_data/${SIZE}/1/dict test_data/${SIZE}/1/query ra
```

D.2 Computing Run Times

```
time java comp26120.speller_bstree -d ./data/test_data/${SIZE}/
1/dict -m 0 -vv ../data/test_data/${SIZE}/1/query
```