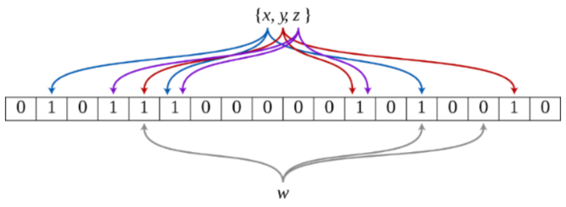
Bloom Filter – Introduction

A Bloom filter is a probabilistic data structure that is used to test whether an element is a member of a set or not. It works by hashing the elements and storing the hash values in a bit array. When checking if an element is in the set, its hash value is computed and compared to the bit array. If all the corresponding bits are set, then it's likely that the element is in the set. However, there's a chance of false positives, meaning that an element might be incorrectly identified as being in the set even though it's not. [1]



Theoretical Analysis

Sequential Search (LS)

We used a regular Python list due to the good optimisations it has. Its contiguous memory access makes it faster than e.g. a linked list whereby the elements are stored in random places in the program's memory. While linked lists don't require shifting the entire list to accommodate for insertions in the middle of the list, we don't need this feature since inserting appends to the end of the list only. Searching runs at O(N) time due to N accesses in the worst-case scenario, $\sim \frac{N}{2}$ on average. Auxiliary space is O(1) as it only needs constant additional memory for the current index. Inserting uses append() rather than insert() reducing O(N) → O(1), but we have to search for duplicates at first, so that is O(N) + O(1) = O(N).

Binary Search Tree (BST)

We used an iterative implementation since insertions and searches could then have auxiliary space O(1) compared to O(N) that would have been attained with recursion owing to the call stack from accumulative function calls. Besides, we wanted to avoid exceeding the recursion depth limit with large files. Searching and inserting both have average time complexity and auxiliary space O(log(N)) ~ c $\log_2(N)$ where c is a constant such that $c \geq 1$ (due to being unbalanced). They are O(N) in the worst-case scenario (tree degrades to a linked list). We conveniently say those operations are bound by O(h), where h is the height of the tree.

Left-Leaning Red-Black Binary Search Tree (LLRB BST)

We used recursion rather than iteration as it's more elegant and readable. Although recursion requires creating multiple frames on the call stack, this is not a significant issue when dealing with traversals that only require logarithmic levels of recursion. Besides, an iterative implementation with a stack would have incurred an overhead anyway, such that recursion is favourable. Searching has time complexity Θ(log(N)) due to O(log(N)) searches in both average and worst-case scenario, owing to the logarithmic nature of traversal in a balanced tree. Auxiliary space is Θ(log(N)) for the same reason. Inserting has time complexity Θ(log(N)) just like searching. Auxiliary space and time complexity are ~ $2\log_2(N+1)$ in the absolute worst-case scenario that would require rotations for every single recursive call. As we're going to see in experimental analysis, in practice the execution time is $O(\log(N) + C_R)$ where C_R denotes the time taken for rotations, but we omit the asymptotically smaller term as per convention. This works out at Θ(log(N)) for time complexity and auxiliary space.

Bloom Filter (BF)

We used Python's built-in hash function, as it is implemented in C and thus optimised for performance. It has worst-case time complexity O(N) for strings, but it efficiently reduces collisions with hash randomisation and XORing the hash with its string length [2], resulting in amortised time complexity O(1). F-stringing the word with the iterator aims to reduce the number of collisions. Modulus M ensures the hashes can all fit in the array but does not alter computation cost significantly. Space complexity is O(M) for M bits. The larger the M, the lower the probability of collision, at the cost of space. Searching runs at Θ(k) time and O(1) space irrespective of N since it just performs lookup in the bit-array [3][4]. However, note that the number of hashes k itself is optimised according to the equation $\left(1-(1-\frac{1}{m})^{nk}\right)^{k} \approx (1-e^{kn/m})^{k}$ which derives $k_{min} = \frac{m}{n} \cdot \ln(2)$. [5] Inserting runs at Θ(k) time and O(1) space, as it just assigns k bits in the bit-array to 1 in the bit-array, irrespective of their original value.

Set Operation	Time Complexity				Auxiliary Space			
	LS	BST	LLRB	BF	LS	BST	LLRB	BF
Insert	O(N)	O(h)	Θ(log(N))	Θ(k)	O(1)	O(h)	Θ(log(N))	O(1)
Search	O(N)	O(h)	Θ(log(N))	Θ(k)	O(1)	O(h)	Θ(log(N))	O(1)

Experimental Analysis

Experimental Framework

The framework comprises a base class that calculates insert and mean search time and two derived classes that stress test or manage synthetic data. The synthetic data consists of random strings of clamped length with a vowel between every two constants for better readability. Its associated test search file has an equal mix of words sampled from and outside the synthetic data, shuffled by storing it as a Python set. The main class is parameterised with a text file (data_file), a dictionary containing all data structures, the number of times timeit must run on insert or search for each data structure (operation_repeats), search intervals for executing the search function, and a search_test_file containing search words. Iterating over the words in data_file, the time taken to insert values and the time taken to search up every value in search_test_file in specified search_intervals is stored as the values input in the data structure increase. The average for search time is calculated and stored in a mean_search_time dictionary to improve the precision of search timing results. The RunTests class facilitates data manipulation, allowing one to sort data, remove duplicates or check for false positives, hence acting as the testing suite, based on which we produced graphs.

Sequential Search

Figure 1: Sequential Insertions, Dickens

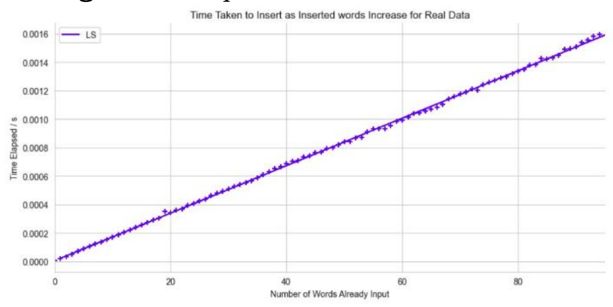
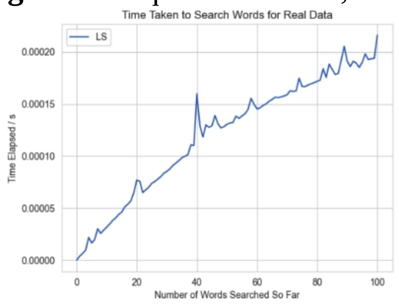


Figure 2: Sequential Search, Dickens



Figures 1 shows linear growth as predicted in the theoretical analysis. We found that an average of 10 repeats was enough to get rid of noise from system-dependent factors. Too many repeats caused overfitting. Figure 2 shows slight concavity in the execution time for search on real data. We conjectured that the graph starts at ~ N due to having to iterate through the entire array at first in the absence of words, then slowly shifting towards an average of ~ $\frac{N}{2}$ as more words are inserted, increasing the likelihood of retrieving a word somewhere in the middle. We chose to omit the graphs for both synthetic data and the other real data, namely Moby Dick and War and Peace, since the patterns observed were the same.

BST & LLRB BST

Figure 3: Insertions for the other sets

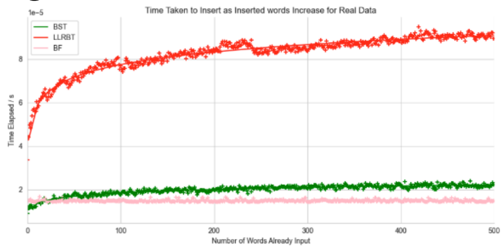


Figure 4: Searches for the other sets

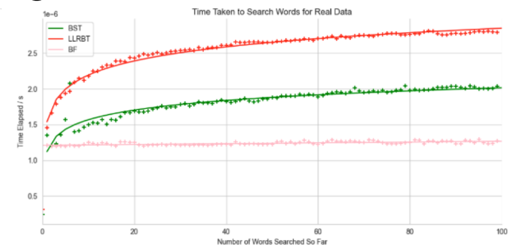


Figure 5: BST insertions in ascending order

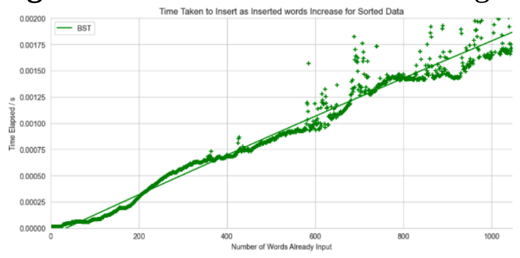


Figure 6: BST insertions in random order

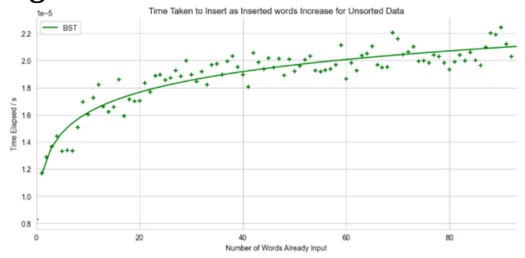


Figure 7: Proof that LLRB BST is balanced

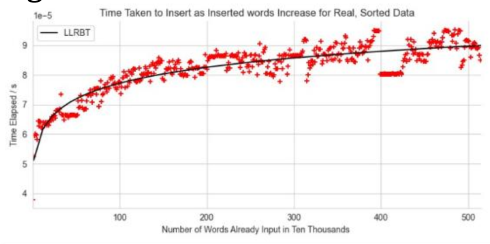
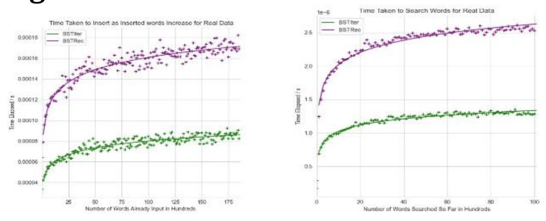


Figure 8: Iterative VS Recursive BST



Figures 3 and 4 show a clear logarithmic shape for BST and LLRB BST, as expected. Figures 5 and 6 portray the worst case and average case performances of BST respectively. We stress-tested the sets on the Dickens file. Notably, BST becomes linear as expected for the edge case (which we achieved by running it on a sorted version of the datafile), meanwhile LLRB maintains logarithmic growth (Figure 7). We plotted the median rather than the mean in order to lower the contribution of anomalies to the moving average. A curio is that LLRB seems to be slower than BST for both insert and search on the random data, despite being balanced. This is most likely due to higher processing time due to rotations in the case of insertions. Especially for large datafiles, this comes with a hefty computational cost. Additionally, BST turns out to loosely match its best-case profile in most cases [6]. We suggest that whilst searches don't involve rotations, there may be an overhead caused by the fact that accessing a red node takes the same amount of time as traversing a black node. Figure 8 shows that the iterative implementation of BST runs 2 times faster than the recursive one, as expected due to the call stack involved. However, this still can't account for the fact that recursive BST outperforms LLRB. We suggest it could be a case of more intensive pointer memory due to rotations, especially for large datafiles. Python's dynamic typing and memory management can lead to poor cache locality and cache thrashing, which can impact performance. On the contrary, BST has much better cache realization. Furthermore, LLRB could have an overhead due to colours and rotations amounting to an additional layer of complexity that BST doesn't, which requires more intensive type checking, again owing to dynamic typing. Figures 5 and 7 also show that LLRB is faster than BST when it comes to the edge case. However, unless previously sorted, text is unlikely to be in ascending order.

Bloom Filter

True Positives: 100,000	False Positives: 20,816
False Negatives: 0	True Negatives: 579,184

Figure 9: Confusion Matrix for Bloom filter searches on large synthetic data comprised of integers casted to strings, obtained from the median of 10 repeats on all possible elements up to M. We compared the results of each search with those of a BST to produce this. As expected, all elements up to N were correctly retrieved, with the rest being predominantly correctly retrieved.

Figures 3 and 4 show constant time for Bloom Filter (BF), as expected. Insertion took longer than search, possibly because setting bits in the array requires writing to memory which takes more time than checking bits, a read-only operation. We'd initially fixed N = 5,000,000 to account for the Dickens file and chosen M/N = 50 to guarantee no false positives. The optimal number of hash functions for this works out at around 35. We then reduced N to 100,000 since this suffices to cover all *unique* words in the file. In attempt to minimise space complexity, we reduced M/N to 7, resulting in $7 \cdot \ln(2) =$ ~ 5 time for both searches and insertions, an improvement from the initial ~ 35. With those new parameters, we attained $\mathcal{P}(false\ positive) < 0.04$. We experimentally tested the expected false positive rate using the equation $\varepsilon \approx (1 - e^{-kn/m})^k$, and got that ε = 0.03443 for an expected 0.03465. [7] Crucially, this assumes that the expected number of insertions, N, loosely matches with actual number of insertions. When we stress-tested the Bloom Filter with 5 times more insertions than expected, it yielded poor results with a false positive rate as high as 0.2891.

Summary

Overall, the experimental results fit with our predictions. Bloom Filter is the fastest, at constant time, followed by BST and LLRB BST, at logarithmic time. Sequential Search, at linear time, does not scale well for large datafiles. However, it may still be useful on a small scale, considering the overhead in BST and LLRB BST for small values of N. It is also by the far the easiest set to implement. When searching on sorted data e.g. a list of usernames in alphabetical order, BST should be avoided. LLRB BST is more robust in applications where edge cases are encountered. If the data is unordered, BST is preferrable to LLRB BST, due to factors discussed at great length in the experimental analysis. One would be better off using Bloom Filter in situations where time is critical and <10% false positive rate is condonable e.g. URL filtering or network packet filtering. It is worth mentioning that positive results can be validated; Bloom Filter is typically used in conjunction with a 2nd data structure to verify false positives, although we've omitted that part. Lastly, to secure a low false positives rate, BF should not be used in situations where data traffic can rise unexpectedly.

References

[1] Apache Doris. "Bloom Filter." Diagram taken from https://doris.apache.org/docs/data-table/index/bloomfilter/
[2] Holmes, Lee. "Efficiently Generating Python Hash Collisions." https://www.leeholmes.com/efficiently-generating-python-hash-collisions/
[3] Brilliant. "Bloom Filter." https://brilliant.org/wiki/bloom-filter/.
[4] Geeks for Geeks. "Bloom Filters: Introduction and Python Implementation." https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/.
[5] Wikipedia. "Bloom Filter." See section on optimising the number of hash functions: https://en.wikipedia.org/wiki/Bloom_filter#Optimal_number_of_hash_functions
[6] Sedgewick, Robert, and Kevin Wayne. Algorithms (4th Edition). Page 403.
[7] Wikipedia. "Bloom Filter." See section on probability of false positives: https://en.wikipedia.org/wiki/Bloom_filter#Probability_of_false_positives