**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 |
| **Chart, line chart  Description automatically generated** | **Chart, line chart  Description automatically generated** |

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 ~ 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 |
| Chart, line chart  Description automatically generated | Chart, line chart  Description automatically generated |
| **Figure 5**: BST insertions in ascending order | **Figure 6**: BST insertions in random order |
| Chart, scatter chart  Description automatically generated | Chart, scatter chart  Description automatically generated |
| **Figure 7**: Proof that LLRB BST is balanced | **Figure 8**: Iterative VS Recursive BST |
| **Chart, scatter chart  Description automatically generated** | **Chart  Description automatically generatedChart, line chart  Description automatically generated** |

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.

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

**Bloom Filter**

**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 O(5) for both searches and insertions, an improvement from the initial O(35). With those new parameters, we attained < 0.04. We experimentally tested the expected false positive rate using the equation , and got that = 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.