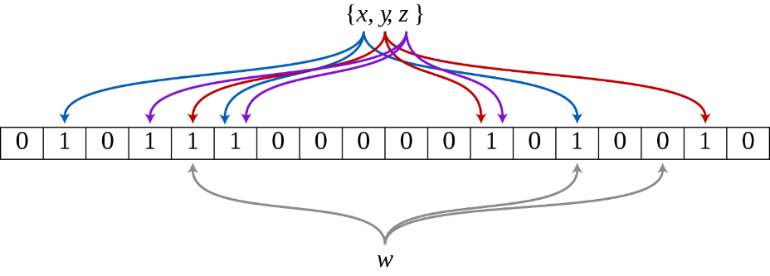
**Bloom Filter** – **Introduction**

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. A false positives rate of 10% is considered condonable for most situations e.g. URL filtering and network packet filtering. Bloom Filter is typically used in conjunction with a 2nd data structure to verify false positives, although we’ve omitted that part.

**Theoretical Analysis**

**Sequential Search**  
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, ~ on average. 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).

**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 log₂(N) where c is a constant of proportionality such that c ≥ 1 (due to being unbalanced). It is O(N) in the worst-case scenario (degrades to a linked list). We conveniently say those operations are bound by O(h), where h is the height of the 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 O(log(N)) for the same reason. Inserting has time complexity Θ(log(N)) just like searching. Auxiliary space and time complexity are ~ 2log₂(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 ~ log₂(N) + CR where CR 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**  
We used a bit-array and Python’s built-in hash function rather than FNV or Murmur, 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, resulting in amortised time complexity O(1). The F-stringing of 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). The larger the M, the lower the probability of collision, with the penalty of larger space requirements. Searching runs at Θ(k) time irrespective of N, since it just performs lookup in the bit-array. However, note that the number of hashes k is optimised according to the equation which derives = . Inserting runs at Θ(k) too, as it just assigns k bits in the bit-array to 1 in the bit-array, irrespective of their original value.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Set | Time Complexity | | | | Auxiliary Space | | | |
| Operation | SS | BST | LLRB | BF | SS | 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**It is a long-established fact that a reader will be distracted by the readable content of a page when looking at its layout. The point of using Lorem Ipsum is that it has a more-or-less normal distribution of letters, as opposed to using 'Content here, content here', making it look like readable English. Many desktop publishing packages and web page editors now use Lorem Ipsum as their default model text, and a search for 'lorem ipsum' will uncover many web sites still in their infancy. Various versions have evolved over the years, sometimes by accident, sometimes on purpose (injected humour and the like). Various versions have evolved over the years, sometimes by accident, sometimes on purpose (injected humour etc.)   
  
**Sequential Search**

|  |  |
| --- | --- |
| **Figure 1**: Sequential Insertions, Dickens | **Figure 2**: Sequential Search, Dickens |
| **Chart, line chart  Description automatically generated** | **Chart, line chart  Description automatically generated** |
| **Figure 3**: Sequential Insertions, Synthetic | **Figure 4**: Sequential Search, Synthetic |
| **Chart, line chart  Description automatically generated** | **Chart, line chart  Description automatically generated** |

Figures 1 and 3 are linear 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. Figures 2 and 4 show slight concavity in the execution time for search on both real data and a sample taken synthetic data, tested on a mixture of words from and outside the datafile. We conjectured that the graph starts at O(N) due to having to iterate through the entire array at first in the absence of words, then slowly shifting towards an average of O() as more words are inserted, increasing the likelihood of retrieving a word.

**BST & LLRB BST**

|  |  |
| --- | --- |
| **Figure 5**: Insertions for the other sets | **Figure 6**: Searches for the other sets |
|  |  |
| **Figure 7**: BST insertions in ascending order | **Figure 8**: BST insertions in random order |
|  |  |
| **Figure 9**: Proof that LLRB BST is balanced | **Figure 10**: Iterative VS Recursive BST |
|  |  |

Figures 5 and 6 show a clear logarithmic shape for BST and LLRB BST, as expected. Figures 7 and 8 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 9). We plotted the median rather than the mean in order to lower the contribution of anomalies to the moving average. A curio is that BST seems to be faster than LLRB for both insert and search on the random data, despite being unbalanced. 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. 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 10 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 BST’s outperformance over 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. Figure 7 also shows that LLRB is faster than BST when it comes to the edge case, however in reality a text being analysed is unlikely to be in ascending order.

**Bloom Filter**

Figures 5 and 6 show constant time for Bloom Filter, 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. This optimal number of hash functions works out at around 35. We then reduced N to 100,000 since this covers the number of *unique* words in the file, and fixed M/N = 7, attaining < 0.04. The new M/N ratio is meant to minimise the space complexity of the Bloom Filter. We experimentally tested the expected false positive rate using the equation , and got that = 0.034 for an expected 0.037. This results in O(5) for both searches and insertions, an improvement from the initial O(35).

**Summary**Overall, the experimental results fit with our predictions. Bloom Filter is the fastest, at constant time, followed by BST and LLRB BST. Sequential Search, at linear time, does not scale well for large datafiles. In situations where time is critical, and 4% false positive rate is tolerable, one would be better off using Bloom Filter. BST and LLRB BST are both logarithmic as expected, and we provided explanation for why we think the latter may have performed slower than the former. The edge case performance of BST is highly unlikely to be realized since text files don’t tend to be sorted in ascending order.