**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 Θ(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 O(log(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.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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) |

**Bloom Filter**  
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). 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 [3][4]. However, note that the number of hashes k itself is optimised according to the equation which derives = . [5] 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.