**Theoretical Analysis**

**Sequential Search**  
Uses 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 insertElement appends to the end of the list only. searchElement: O(N) at worst case scenario, O() on average. insertElement: using append() rather than insert() reduced O(N) → O(1), but we have to use searchElement to check for duplicates at first, so that is O(N + 1) = O(N).

**BST**  
Uses an iterative implementation since insertions and deletions 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 recursive calls. Besides, we wanted to avoid exceeding the recursion depth limit with large files. searchElement: on average c log₂(N) where c is a constant of proportionality such that c ≥ 1 (due to being unbalanced). O(N) in worst case scenario (a linked list). insertElement: c log₂(N), same as searchElement. Those complexities are the same for auxiliary space.

**LLRB BST**  
Uses recursion rather than iteration because it’s more elegant and readable. The recursion stack wouldn’t be as much of a problem when dealing with traversals in logarithmic space, and besides, the iterative implementation uses a stack as well causing an overhead, such that recursion is favourable. searchElement: Θ(log₂(N)) due to O(log₂(N)) searches in both average and worst case scenario, owing to the logarithmic nature of traversal in a balanced tree. Auxiliary space is O(log₂(N)) for the same reason. insertElement: Time complexity of O(log₂(N)) on average. Auxiliary space and time complexity O(2log₂(N+1)) in the absolute worst case scenario that would require rotations for every single recursive call. In reality, as we’re going to see in experimental analysis, it’s O(log₂(N) + CR) where CR denotes the time taken for rotations, but we omit the smaller term as per convention. Works out at Θ(log₂(N)) for time complexity and auxiliary space.

**Bloom Filter**  
Uses a bitarray and Python’s built-in hash function. 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). We further F-string the word with the iterator in order 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 auxiliary space. searchElement: Θ(k) irrespective of N, since it just performs lookup in the bitarray. Hashing has a negligible amortised time complexity, as discussed above. However, note that the number of hashes k itself is optimised according to the equation which derives = . For our values, this works out at O(35) due to M/N ratio = 50. insertElement: Θ(k) once again, just assigns k bits in the bitarray to 1 in the bit-array, irrespective of their original value.