**Sequential Search**  
Uses a regular Python list due to the good optimisations it has. Its contingent memory access makes it faster than e.g. a linked list whereby the elements are stored in random places in the program’s memory. Space complexity: O(N).  
searchElement: O(N) at worst case scenario, 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 header stack from accumulative recursive calls. Space complexity: O(N). 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). Ω(1) for singleton tree, uninteresting.  
insertElement: same as searchElement, but constant time for an empty tree instead.

**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. Space complexity: O(N).  
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. Ω(1) for a singleton tree, trivial. insertElement: Time complexity of O(log₂(N)) on average. Auxiliary space O(2log₂(N+1)) in the absolute worst case scenario that would require rotations for every single recursive call. Ω(1) for an empty tree, once again trivial.  
Works out at Θ(log₂(N)) as well.

**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 as its slightly faster than concatenation. Modulus M ensures the hashes can all fit in the array but does not alter computation cost. 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. Mathematical calculations involved have negligible computational cost. However, note that the number of hashes k itself is optimised according to the equation  
 which derives = .  
insertElement: Θ(k) once again, just assigns k bits in the bitarray to 1 in the bit-array, irrespective of their original value.