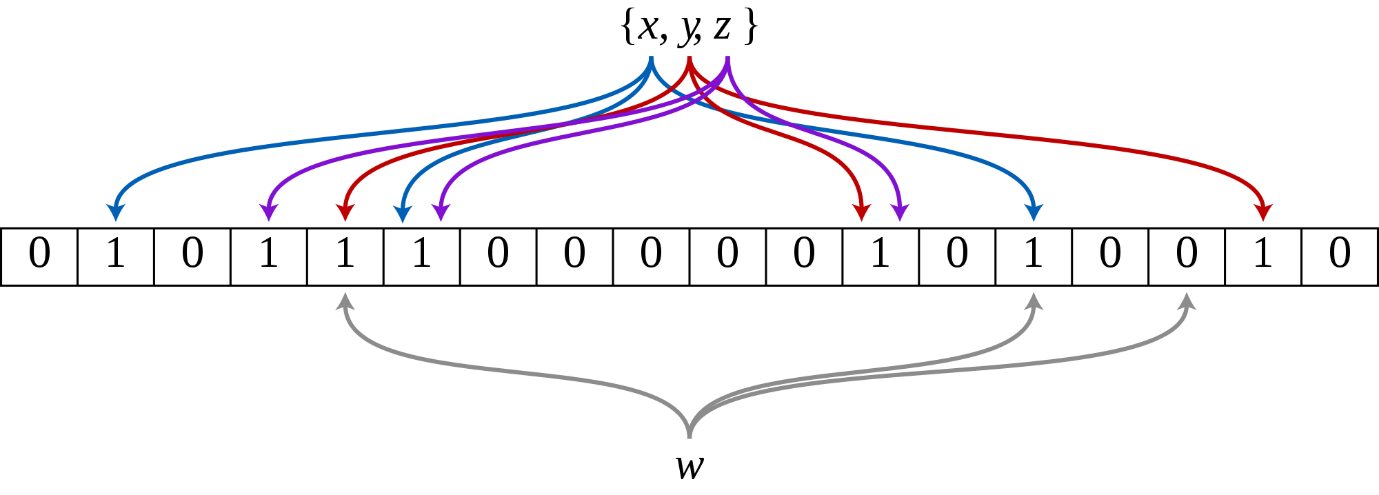
## Algorithms Coursework Group 11

**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**  
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.

**Experimental Analysis**

**Sequential Search**[Include graphs] Graph seems to be valid; it is linear for both insert and search for the real data which accounts for O(N+1) and O(N) from the theoretical analysis respectively. As expected, for synthetic data with no repeats and modified code for insertElement without the check, insertions were O(1) since it’s just appending to a Python list. Was the most time-consuming, included in a separate graph so to not scale down the others. We found that a moving average with 10 repeats was enough to get rid of noise.  
**BST & LLRB BST**  
[Include graphs] Both graphs are logarithmic and of the right shape. We plotted the median rather than the mean in order to lower the contribution of anomalies to the MA. A curio is that BST seems to be faster than LLRB BST for both insert and search, despite being unbalanced. Note that when we tested the cumulative time taken for 10,000 insertions on the synthetic data, BST was faster than LLRB for insertions but slower than it for searches- this makes a lot of sense considering the impact of rotations on processing time, compared to searches which are in-place and guaranteed to be faster if the tree is balanced. When running a recursive rather than an iterative implementation of BST, it ended up running 2 times slower, 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 intensive pointer memory due to rotations, especially for the larger values of N. Python is poorly optimised to deal with CPU cache jumps, namely it can cause poor cache locality and cache thrashing. BST has much better cache realization. Furthermore, LLRB could have an overhead due to having an additional layer of complexity that BST doesn’t (colours, rotations) which requires more intensive type checking, which, in a dynamically typed language like Python, could slow things down.

**Bloom Filter**  
[Include graphs] Graph shows constant time, as expected. [Talk about the false positive analysis and include graphs] We’ve fixed N = 5,400,000 to account for the Dickens file and chosen M/N = 50 to guarantee no false positives. This works out at around O(35). A good compromise is M/N = 11, attaining < 0.05 and O(8). This results in O(k) for searches not in the list (90% of the cases) but O(logN + k) searches in only 10% of the cases.