**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). In real life situations e.g. a website tracking visitors’ IP addresses, even < 0.1 might be fine, since the whole novelty of bloom filter as a probabilistic data structure is to be used in conjunction with a secondary data structure e.g. LLRB to validate positives. 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, when trying to ensure an IP address is not falsely flagged as malicious.