Algorithms report (Group 35 members: Sindi Gjonaj, Nakul Goyal, Ivan Lee, Carolina Zhu)

Theoretical analysis:

Sequential search

This algorithm was implemented to search for the keywords iteratively using a list to hold the elements. An iterative approach was chosen over a recursive as the latter would require additional memory. In our chosen implementation, each element takes up a memory space, resulting in a space complexity of O(n). The time complexity for the insertElement and searchElement function of this algorithm is O(n) for both the average and worst case scenarios, with n being the size of the list. The worst case scenario occurs when the element which is being searched for is the last one or not present, as the whole list would need to be iterated through. The average case scenario would have a time complexity of O((n + 1)/2), which simplifies to O(n).

Binary Search Tree

The time complexity for insertElement and searchElement functions is O(n) in worst cases as it depends on the height of the tree and the tree could happen to be completely imbalanced, making the height of the tree n, where n is the number of nodes. Meanwhile, the time complexity would be O(log n) under average cases, where n is the number of nodes, as the height of the tree would be closer to log n when nodes could be more dispersed across the tree instead of all falling on one side.
In terms of space complexity, it would be O(n) for the insertElement function as a node would be allocated for each element in the set, while that for the searchElement function would be O(1) as it requires the same amount of memory in every iteration of the loop, hence why we chose to implement the BST iteratively as the space complexity would be lower than a recursive implementation.

Balanced Search Tree

To implement the balanced search tree, a left-leaning red-black (LLRB) tree approach was specifically taken. The code for searching was written so that the operation is carried out iteratively, which results in a constant space complexity of O(1), as opposed to recursively which would have had a space complexity of O(log n). In regards to time complexity for search, both in the average and worst-case scenario the performance is O(log n). Since every path from the root node to the null links has exactly the same number of black links, the tree maintains its balance and it will never be the case that it has an asymmetric structure (as the red links aren't counted). Furthermore, a node cannot have two red links connected to it. It is these properties that also guarantee a time complexity of O(log n) for the average and worst-case scenario when it comes to insertion with a LLRB tree. Specifically, in the worst-case scenario, the height of the tree will be at most 2 log n and so, after n inserts, the time complexity is O(log n). In the average case, the time complexity after n inserts is O(log n) as well.

Bloom Filter

To implement a bloom filter, a bit array of a specific length **m** is created with each bit initially set to False (or 0). **k** hash functions are applied to the element being inserted outputting values in index range (0 to m-1) of the bit array. The bit at this value is then set to True (or 1). While searching for an element we find its hash values (hash functions are deterministic) and if the bit at all the hash values is True then the element is possibly present, otherwise it is definitely not. There is a certain probability **p** we get a false positive from the Bloom Filter due to collisions. Let's say we are inserting **n** elements in the Bloom Filter set. Then the following theoretical formulas give values of m and k, given n and p values:
$m = -n * \ln(p) / (\ln(2))^2$ and $k = (m/n) * \ln(2)$
In our case: number of insertions, $n = 10^5$, acceptable false positive rate, p = 0.01 (1 in 100 false positives), number of bits in bit array, m = 958505, number of hash functions, k = 6.64 ≈ 7
The space complexity of the bloom filter only depends on m, the number of bits in the bit array. Thus **space complexity is O(m)**. The time complexity of the bloom filter only depends on k, the number of hashes being performed and is independent of n. Thus **time complexity is O(k)**. Bloom Filter functions the same for any string data given and in any order (since it just calculates hash values) and thus it has no worst case or best case scenario. The 2 hash functions used were: Fowler–Noll–Vo (or FNV) and Python inbuilt hash function. These are both non-cryptographic since safety isn't a concern here, they can be executed quickly and have a good avalanche effect (helps in preventing collisions for similar strings).

Real data from test files analysis

Average Insert/Search time (s) rounded to 5 significant figures					
Test file	Insert/Search	Algorithm			
		Sequential Search	Binary Search Tree	Balanced Binary Search Tree	Bloom Filter
Mobydick	Insert	8.2483	1.5492	0.83917	0.75503
	Search	0.27234	0.18628	0.00058651	0.0020861
Warpeace	Insert	43.984	5.5938	1.7635	1.7028
	Search	0.37810	0.34938	0.00058928	0.0013725
Dickens	Insert	896.79	31.566	19.076	13.855
	Search	0.74508	0.021	0.00078445	0.00182300

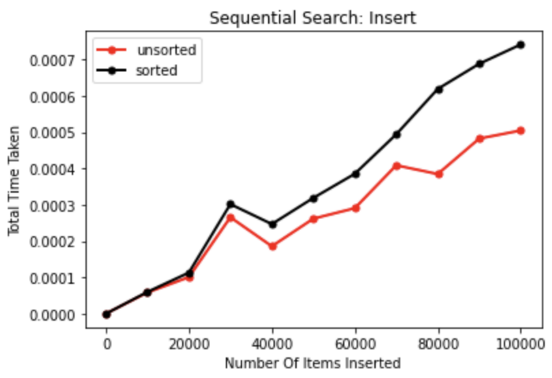
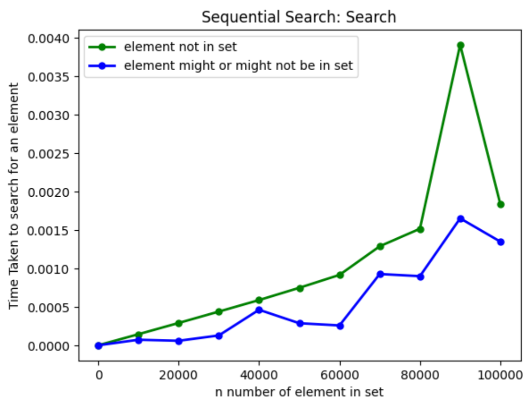
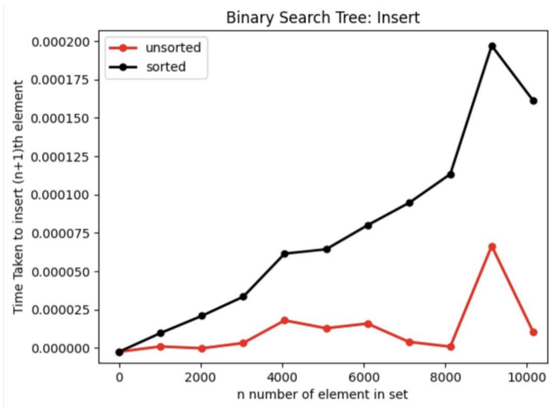
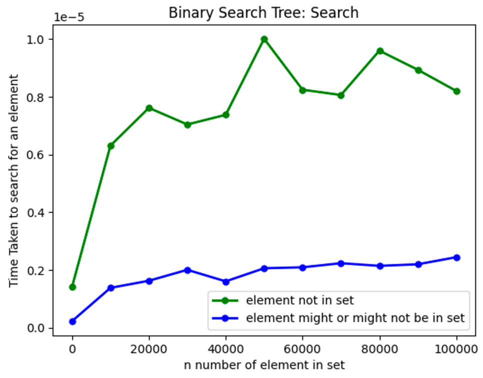
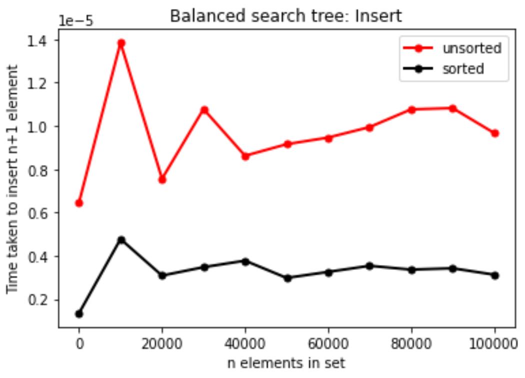
To calculate the average time for both Search and Insert, the time to run each test file with different algorithms was executed three times and a mean value was calculated. Through the values obtained, we can further prove our previous conclusions in the theoretical analysis. These values show that the bloom filter is the most time efficient in relation to the insert operation as it has consistently gotten the fastest speed for this and the Balanced Binary Search Tree is most efficient for the search operation, obtaining very small values for time taken, regardless of the size of the test file. The Bloom Filter acts similarly, with its values for search time also remaining constant regardless of the test file. This indicates how these algorithms perform this operation well irrespective of the size of the set. The Sequential Search and the Binary Search Tree are slower, with Sequential Search being the slowest out of all four algorithms with a considerable time difference compared to the Binary Search Tree. The time taken for this algorithm increases with each test file, showing time complexity of O(n), indicating it would be most suitable for smaller data sets as the time for Insert in Dickens is already very large. It is hard to fully predict what the pattern for values is with only three test files, however the Binary Search Tree indicates log n time complexity for the Insert operation. These results support what we suggested in our theoretical analysis.

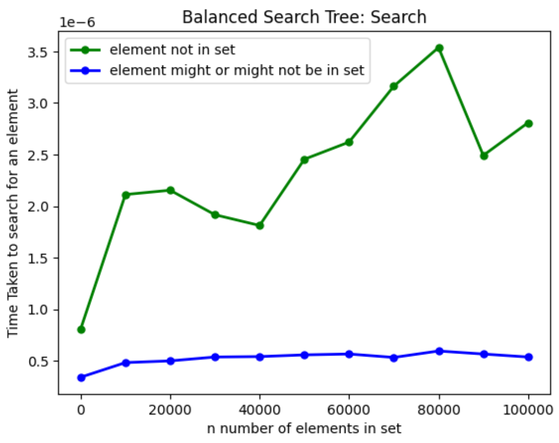
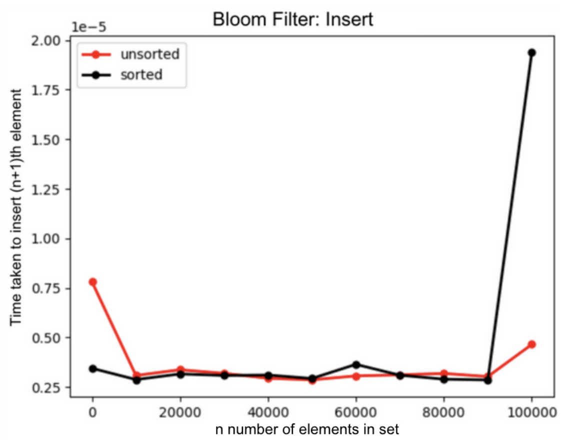
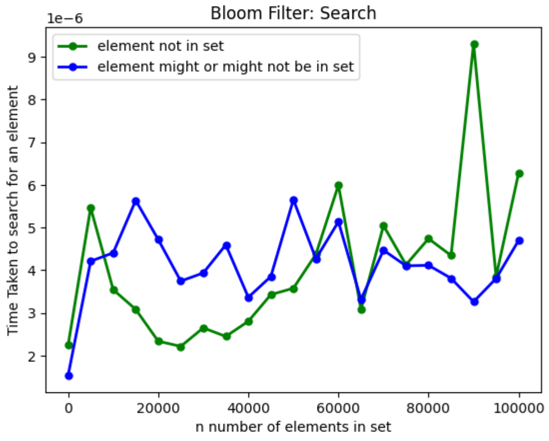
Experimental analysis:

Framework

For the experimental analysis, strings are being randomly generated from a combination of A-Z, a-z and 0-9. Their length is anywhere from 5-20 characters in order to properly stress test the data. For all insert graphs, we plotted the number of elements in set, i.e. n vs time taken to insert (n+1)th element (x and y labels not fully accurate for some). We plotted this because this is what gives us the expected corresponding graph to our theoretical complexities. Two cases for insert: inserting elements from a sorted sequence in one case and unsorted in another. This was done in order to see the effects it has on the algorithm since order in which you insert may impact the time of insertion. For search graphs, we plotted the number of elements in set, n vs average time taken to search one element. The average was taken by doing the process 20 times for a specific n and then dividing total time by 20. This gives rise to fewer outliers and more accurate results. Two cases for search: element being searched is definitely not in set and element being searched has 50% chance to be in set. This was done by using the random module (taking average from 20 helps here, since 10 cases could not be in set and 10 in set). For both insert and search we took n to 10^5 in order to test for a high range of values and see a clearer trend in graphs. We tested at 10 n values which are at intervals of 10000. For more points it took un an unreasonably high amount of time for the code to run.

Graph	Explanation

	As stated in the theoretical analysis, this graph also displays linear growth of O(n) in both scenarios, as the number of items inserted increases, the total time taken increases accordingly. The two lines also show how the total time taken does change when comparing its performance with sorted and unsorted lists, however, the difference is very minimal and there is close to no difference up until 20000 items inserted, indicating indifference in performance respective of the list being sorted or not. The graph shows again how this algorithm is most efficient on small lists.
	The same time complexity of O(n) that is mentioned in the theoretical analysis is proven in this graph. However, the growth of the average case scenario(element might or might not be in set) is not as smooth as it would be expected but this could be due to some other external factors that occurred during the experiment. Overall, the time taken to search for an element still increases as the number of elements increases. Then for the worst case scenario (element not in set), aside from one outlier, the line progresses in a smooth line, with the time taken increasing accordingly to the number of elements in the set.
	For sorted data sets (worst case), it takes much more time than that for unsorted data sets (average case) since the tree would become unbalanced if elements are being inserted in sorted order. We could see that the time needed for inserting elements increases with the number of elements in a logarithmic trend for unsorted data sets, meanwhile, the time needed increases almost linearly with the number of elements, supporting our theoretical analysis that the time complexity for average and worst cases is O(log n) and O(n) respectively.
	Although the graph suggests that the time taken doesn't increase in a logarithmic trend perfectly, the reason behind this is that there aren't enough data points. Meanwhile, we searched for elements that might be in the set and are not in the set to demonstrate average cases and worst cases respectively. We could see that even though searching for elements that are not in the set takes a longer time, the time taken to search for an element under both situations still increases logarithmically.
	From the experimental data, the graph highlights a logarithmic trend for a balanced search tree insertion operation both for sorted and unsorted input. With smaller values, there seem to be outliers and the trend is not as clear, but as the number of elements increases in the set, the values regulate and the pattern becomes apparent. The difference between the two functions is a constant value, but for the purpose of analysis, this value is omitted and so time complexity remains O(log n) for both cases.

	As mentioned in the theoretical analysis, this graph also displays what resembles log n functions for both the worst case (when the element is not in the set) and the average case (when element might or might not be in the set) scenario. When the element is not in the set, search takes a longer time because the entire tree needs to be traversed in an attempt to find the element. Nevertheless, the time complexity for LLRB search still remains O(log n) for both worst case and average case scenario.
	Aside from the one outlier point, it is evident from the nearly horizontal graph that the time complexity for insertion in bloom filter is O(constant) as is its theoretical complexity. The time taken is almost identical in case of both sorted and unsorted insertions since the hash functions takes identical time for elements and so the order of insertions doesn't matter. Bloom filter stays extremely fast (around 0.4 x 10^{-5} second) even around n =10^5.
	Although a jagged graph, we can see that it follows the trend for a horizontal line since the time values for higher n values are comparable (or even lower) to those for lower n values. Thus, time complexity for search in bloom filter is O(1). Also, in case of Bloom Filter, searching for an element that is not in the set can be faster since fewer hash functions can be performed if we get a False bit initially. Thus, as seen from the graph for n values between 10000 and 50000, the time taken for element not in set is lower.

Conclusion

To conclude, throughout our report, we have analysed the efficiency of the following four algorithms, ranked in order of their overall efficiency, from least to most efficient: Sequential Search, Binary Search Tree, Balanced Binary Search Tree and Bloom Filter. Sequential search only works well when the size of the data set is small as the time required for it to perform the insert and search functions increases linearly, while the same could happen to binary search trees if the tree is completely imbalanced. This algorithm can therefore be upgraded to a Balanced Binary Search Tree, which performs similarly to the previous algorithm, yet ensures the worst case scenario does not happen as it maintains the tree balanced. Then the last algorithm, the Bloom Filter proves to be the most efficient whilst holding the cost of losing accuracy. As the Balanced Binary Search Trees' speed values are so similar to the Bloom Filter's, we believe that this algorithm could be better for our scenario regarding the set membership because accuracy for many of these situations, such as determining usernames, is incredibly important. The bloom filter would be safer to use in an environment where potential false positives would not cause problems.