Group 38 - Ahmed Ansari, Kerem Eroglu, Andy Wu, Ninaad Kulkarni

The set membership problem, albeit primitive, has been essential in the modern field of Computer science, as it forms the building blocks for languages, such as automata and Higher-Level languages. The aforementioned problem allows for any computational challenge, where the answer is true or false, to be expressed as a query wherein the language is the set of strings where it yields True. There are numerous ways to decipher this problem, each with varying ease, results, and complexity. In this investigation, 4 dynamic data structures were put to the test – Sequential Search, Binary Search Trees, Balanced Search Trees, and Bloom Filters. Each of these abstract structures was programmed in Python, before being rigorously tested for inserting and searching using real world, as well as synthetic data. The best-case, worst-case, and average-case results were analysed, as were the time and space complexity for each. The results were evaluated as each datatype was compared, and suitable scenarios were determined, the results of which are elaborated in this report

Synthetic data:

The data structures were thoroughly investigated by means of real-world, as well as synthetic data. The latter was established with the primary purpose of stress-testing using a myriad of inputted values, alongside testing the worst and best theoretical cases for each data structure. The examinations were conducted with differing amounts of values such that trends could be established. Furthermore, multiple trials were performed for each operation and data structure; and the average was used to reduce random error and increase precision. As sequential search is a linear data structure, no edge cases were implemented. However, for binary and balanced trees, the worst case was judged to be a sorted list of elements, in either ascending or descending order. This was since each node in the binary tree would have only one child and it would effectively turn into a linked list, diminishing its efficiency. Moreover, the LLRB tree would frequently have 2 neighbouring red nodes, causing operations to occur regularly. Contrarily, the best case was determined to be a median case, wherein the median value of the input set was entered first as the root; followed by the median of the left subtree, and the median of the right subtree. This pattern was continued and predicted to yield the quickest results as every node in the BST has 2 children and no operations are required in the LLRB tree. Additionally, a multitude of additional tests were conducted to determine the efficacy of the bloom filter with varying size, hash functions, false positives etc.

Real Data Results:

Table 1: The times taken for searching and inserting data into the real-world files for each data structure

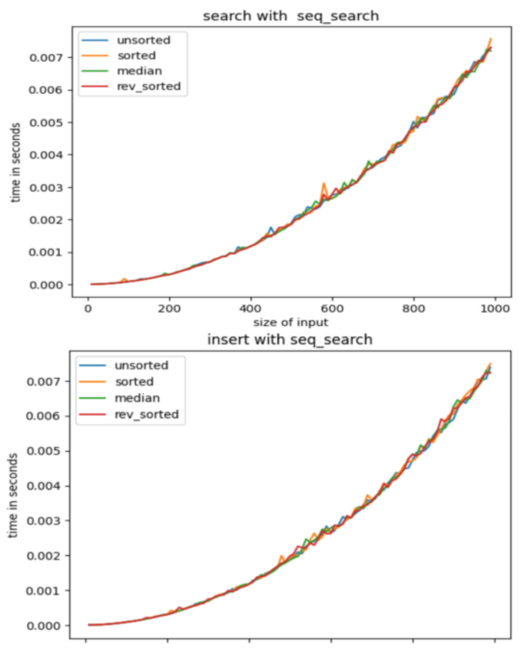
	File 1 – Moby Dick	File 2 – Warpeace	File 3 - Dickens
Sequential Search Insert	7.325 s	15.603 s	519.913 s
Sequential Search Search	0.041 s	0.038 s	0.188 s
Binary tree Insert	0.352 s	0.919 s	8.369 s
Binary tree Search	0.001 s	0.001 s	0.001 s
Balanced tree Insert	0.787 s	2.256 s	27.684 s
Balanced tree Search	0.002 s	0.002 s	0.003 s
Bloom filter Insert	0.313 s	0.823 s	7.930 s
Bloom filter Search	0.001 s	0.001 s	0.003 s

Sequential search

Sequential search is a searching algorithm that checks each element in a list until the item is found. It is an easy algorithm to implement with only for loops, but its time complexity is very slow for extremely large lists because to search for an element, it needs to check each element one by one to see if it matches, resulting in an average/worst case of O(n) time complexity where n is the number of elements in the list. Therefore, sequential search should only be used if the list is very small, where the time complexity is not a significant problem. However, in the best case, the element to be found could be first in the list. When using sequential search for real data such as the text files in *Table 1*, each string will need to be inserted into a list. Since the list is essentially a set of elements, the space and time complexity for the insert operation will be O(n) where n is the number of distinct and unduplicated elements in the text file.

Experimental Analysis

For both search and insert operations in sequential search, the time of execution for all four different synthetic data is very similar as the size of output increases *(Graph 1a, 1b)*, producing a linear time complexity that supports the theoretical analysis of the time complexity being O(n) stated above. This shows that <u>regardless of how the data is arranged (sorted, unsorted, median and reverse sorted), the execution time will roughly be the same</u> due to there being no extreme edge cases for sequential search.



Graph 1a, 1b: times for insert and search for sequential search for all cases

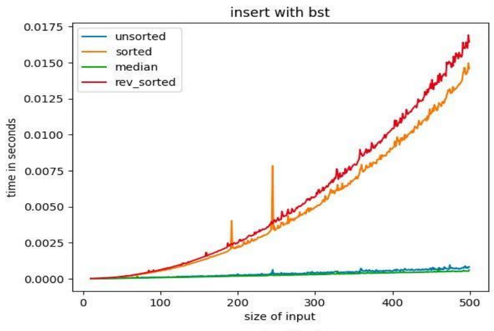
Binary Search Tree:

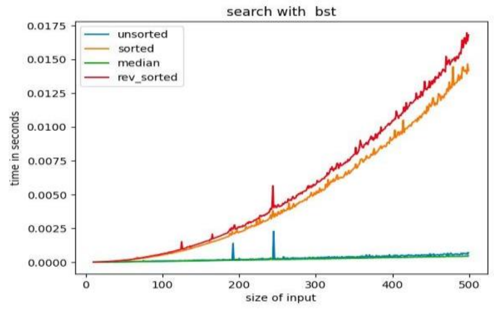
Binary search tree (BST) is a type of tree data structure that is made up of nodes and children, with each value encompassing a node, and branching downwards to a maximum of 2 children; such that the left child is smaller than the node and the right child is always larger.

In terms of time complexity, when compared to most linear data structures, it outperforms in most fields and excels at searching. The most significant advantage of any tree data structure is searchability, as the data is organised in such a way that the value can be compared with other data, and based on that comparison, the data can be easily traced. In the average case, the binary search tree is O(log(n)), which is better than all linear data structures while still being a simple data structure. Moreover, BST is also advantageous in terms of space, as the space complexity is also O(n), the best possible space complexity achievable in any data structure. However, if the input is already sorted, BST can behave exactly like a list, resulting in a tree that is linearly leaning to one side, making it no different than a linked list.

Experimental Analysis:

In *Graphs 2a & 2b*, the insertion and search to BST in best (median sorted) and average case is similar with O(log(n)) complexity, with best case being a bit faster, while the worst-case scenarios (sorted and reverse sorted) create a linear data (a linked list) structure instead of a tree data structure, which makes its time complexity O(n). In summary, except in the worst-case scenarios where the whole idea of tree structure goes off, BST is a simple but effective data structure that can provide efficiency and versatility in many aspects.





Graph 2a, 2b: times for insert and search for binary search tree for all cases

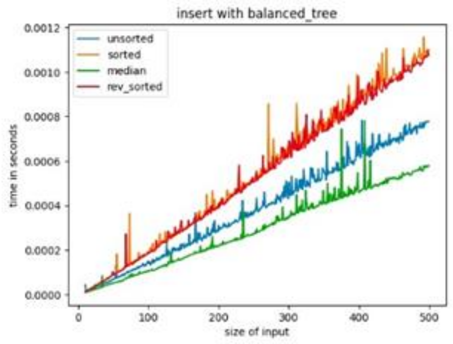
Left Leaning Red Black Tree:

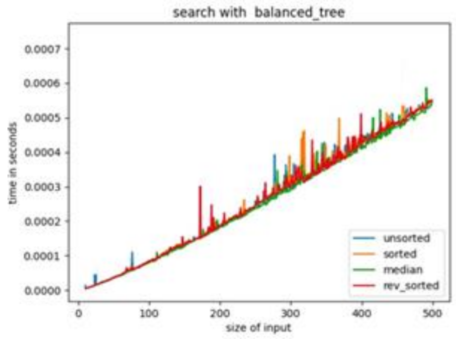
A LLRB tree is a variant of balanced search tree which relies on assigning each node a colour (Red/Black). The insertion of an element into said tree consists of 3 phases. Firstly, the element is inserted in an identical manner to a binary search tree, before the new node is coloured red if it isn't a root. In the third stage, if two or more red nodes are adjacent to one another, then the tree balances itself by performing at least one of three operations – colour swapping, rotating left or rotating right. In terms of time complexity, the BST style insertion is consistent for all trees at the height of the tree, which is O(log N) as the tree is balanced. The initialising of the colour is also uniform at O(1). However, the third part varies in different situations. The best case is when the insertion generates only one red node, thus no further operations are needed. The worst-case scenario is achieved when two consecutive pairs of red branches are adjoining, and multiple operations are needed. At any given time, only one operation is occurring, spanning the height of the tree, thus the upper bound is O(log n). The algorithm is tightly bound, with θ(log n), thus the average case complexity is also O(log n).

In all such trees, the properties of a binary tree as well as a balanced tree are maintained; and each node only consists of 1 bit of data – the character containing its current colour. Thus, the space complexity will always be O(n). As the search function operates similarly to that of a binary tree, it also has a time complexity of O(log n) and a space complexity of O(1)

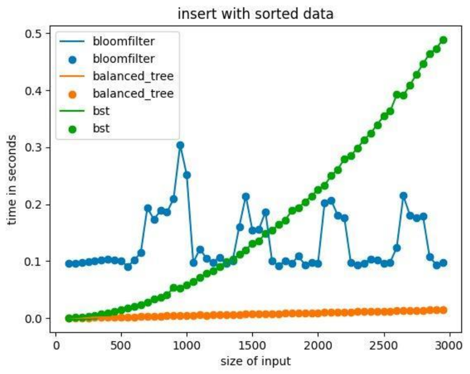
Experimental Analysis:

In this investigation, multiple real and synthetic data sets of varying lengths were fed into a LLRB tree, and the time required for each operation was measured. The adjoining graphs show the results for the latter, while Table 1 illustrates the same for the former. From the real data, which contain significantly more strings, it is perceptible that a balanced tree is inefficient with large datasets as it is 3 times slower than the BST and bloom filter for 'Dickens' – a file encompassing 5 million words.





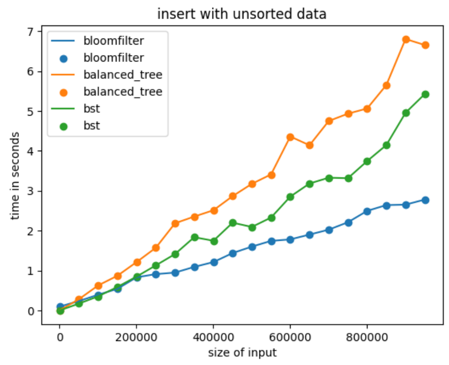
Graph 3a, 3b: times for insert and search for balanced search tree for all cases

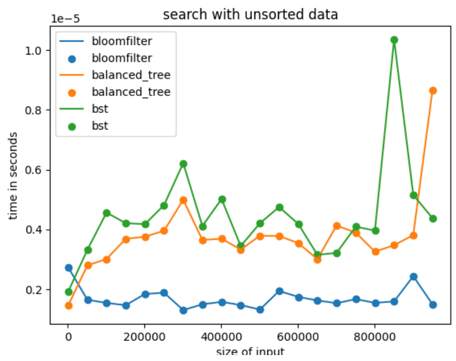


Graph 4: Time taken for insertion of 3000 sorted elements into 3 data structures

The most noticeable result from these graphs is that insertion takes longer than searching, as for 500 sorted values, insertion elapses 1.2 ms while searching takes 0.5. This is attributable to the significant number of rotations, and the reduced height of the tree during searching. Prior to research, it was hypothesised that the plot would resemble a logarithmic graph; and that the median data would be the quickest, with sorted data slowest. The results obtained reinforce this, as the shapes of the graphs for inserting median and unsorted data look like a log function. Moreover, as the data values increase, the median data expends the least time; while sorted and rev-sorted require the most (1.1 vs 0.5 ms for 500 values).

Additionally, it was deduced that the LLRB tree had the highest floor out of the 4 data structures – i.e.: It has the best worst-case performance. As evidenced by *Graph 4*, it has the best functionality for both insertion and searching, starting at a value arbitrarily close to 0 sec. It continues to be the best until a large number of values, hence showing for small datasets, balanced search trees will always be capable. However, it is less efficient than binary trees and bloom filters for unsorted data, as seen in *Graph 5a*, denoting that the average performance is relatively inferior to other data structures.



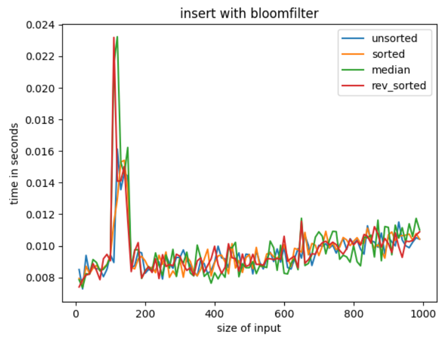


Graph 5a, 5b: Time taken for insertion and search for 800,000 unsorted elements into 3 data structures

Bloom Filter:

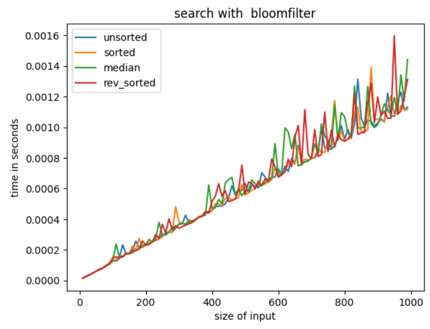
The Bloom filter is a probabilistic data structure that consists of a bit array - initialised to zeros- and a set of hash functions. The number of hash functions used by the bloom filter is often denoted by **k** and the size of the bloom filter is given by **m**. For our implementation, we choose a Python list data structure as our bit array. The hash functions were generated using the built in Python hash function and to ensure that each hash function was unique, every hash function hashed the value with a unique seed.

Elements to be inserted into the Bloom filter are first hashed using the set of hash functions to obtain indices of the bit array which are to be set to 1. During the search operation, the element to be searched is hashed using the same set of hash functions and the hashes are treated as the indices of the bit array. If any of the values at these indices is 0, it can be concluded with certainty that the element is not in the Bloom filter. Conversely, if all values at the indices are 1 then it is possible that the element is present in the Bloom filter since some other element may have generated the same exact set of hash values as the element being searched - giving rise to the probabilistic nature of the data structure. The probability of obtaining a false positive when searching the bloom filter is denoted by the false positive rate.



The time complexity for searching and inserting elements into the Bloom filter is constant at O(**k**) and its space complexity is O(**m**). As the Bloom filter hashes every element being inserted or searched using a constant number of hash functions, it shows consistent and very fast search and insertion performance making it the overall fastest data structure among the linear and tree data structures under consideration.

Experimental Analysis:



The Bloom filter also has the fastest search performance across all three files of the real data. This is most clearly evident in the search time of 7.93 seconds for book 3 *(Table 1)* which is 0.439 faster than the search time with Binary trees - which is the second fastest data structure for search operations. The Bloom filter also shows the fastest insertion times with test file 1 (Moby Dick) and test file 2 (War peace) against all the data structures by a margin of at least 0.3 milliseconds.

Graph 6a, 6b: times for insert and search for bloom filter for all cases

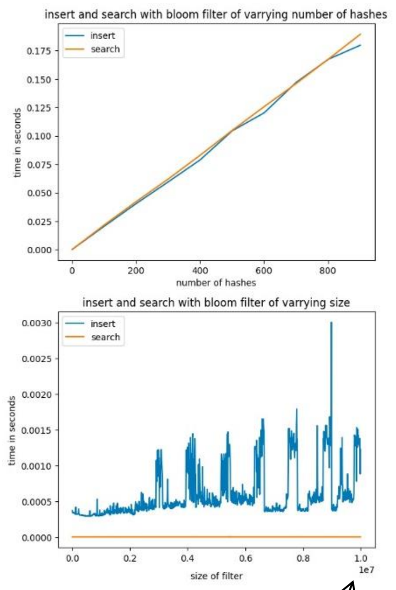
The bloom filter shows consistent and fast performance across synthetic data. To test the search and insert times of the Bloom Filter, test data varying in size from 10 to 1000 in length and type was generated. Each string in the test data was 10 characters long. For these experiments the number of hash functions(**k**) was set at 3 and the size of the Bloom filter(**m**) was set at 100000000. The results from these experiments are presented in the accompanying graphs – *Graphs 6a & 6b*.

Overlapping plots for insertion and search time for the different types of data in the graphs demonstrates that the Bloom filter performs consistently across all types of data. Running 3 hash functions on different types of strings will take roughly the same amount of time and varying the number of such strings inserted and searched for in the Bloom filter should result in the search and insertion times increasing proportionally with the input size. This strong linear relationship is visible in the constant gradient of the plots across all input types.

To reduce the false positive rate of the Bloom filter, larger values for the size of the Bloom filter (**m**) and number of hash functions (**k**) can be used. We independently varied the values for **k** and **m** to analyse their impact on the search and insertion performance.

The following graphs show the linear relationship between the insertion and search time and the number of hash functions being used *(Graph 7a)*. For this experiment the size of the Bloom filter was set to 100000 and the 1000 random elements were inserted and searched from the Bloom filter. As the number of hash functions increases the time taken to search and insert data into the Bloom filter scales linearly as these operations are proportional to the number of hash functions being used.

To test the impact of bit array size on search and insertion performance 1000 random elements were inserted and searched for Bloom filters of varying sizes. *(Graph 7b)*. The number of hash functions was set to 1. From the graphs, it can be concluded that the insertion and search time remain constant across the different sizes of the filters. The large intermittent spikes for the insertion time could be attributed to initialising larger Bloom filters. Thus to decrease the false positive rate of the Bloom filter without degrading its performance, the size of its bit array should be increased.



Graph 7a,7b: times for insert and search for bloom filter with varying number of hashes and varying size of bloom filter

Conclusion:

During this investigation, four data structures were implemented to test their efficacy in various factors to unravel the set membership problem. The bloom filter had the best performance in terms of time complexity. The bloom filter is useful for confirming if a value is not present, making it suitable for tasks such as checking for malicious URLs or preventing expensive database queries. However, its probabilistic nature means that accuracy is not guaranteed, so it is often used alongside an auxiliary data structure or database to verify potential matches. Bloom filters are one of the few data structures wherein performance is independent of the size of the input data, and if it confirms that an element is not present then expensive queries to the database or an auxiliary database can be avoided. However, in eliminating the chance of a false positive, many issues may arise. These include implementation and querying an auxiliary data structure or database, increasing the size of the bloom filter resulting in an increase in memory usage, and an increase in hash functions. Meanwhile, sequential search had the worst time complexity compared to other algorithms for insert operation when tested with very large real data files such the dickens.txt file. Therefore, the only suitable use case for sequential search is when there is a very small list of elements where the poor time complexity is not a significant problem.

The Binary and Balanced search trees both had similar outcomes, in that the results were logarithmic, and the median test yielded the quickest results, with the sorted data being the slowest in terms of time complexity. There was considerable inconsistency between the worst case and best case, with the worst case being linear for BST and over twice as slower for balanced tree. However, the foremost difference is that the BST had a competent average case and an incompetent worst case, while it was vice versa for the LLRB tree. The most suitable case for BST would be related to searching, such as autocorrect since the tree structure will allow to find all known valid strings promptly. For a LLRB tree, the most feasible scenario would be any wherein the worst case can be achieved frequently, such as implementation of Dijkstra's and other graph algorithms as well as indexing in file systems and databases. Balanced trees are easy to visualize and implement into code; and have a relatively rapid worst-case scenario. Conversely, they become difficult as the number of nodes inserted increases and the cumulative number of auxiliary operations introduced also increases, and the extra bit of data for the colour results in it being less space efficient compared to the others.

Another implementation of sequential search is using a linked list where the program iterates through a linked list of values by following the references of each node to the next element. However, this was not implemented in the code because it requires extra memory to store the pointers/references, resulting in a worse space complexity. Additionally, inserting a single element would require O(n) time complexity each time it is performed, making it extremely slow to insert every distinct string in the linked list. The LLRB tree could be replaced by a variety of other trees, such as AVL which is quicker and easier to execute, and B-trees which can have multiple children and are thus more shallow; as well as 2-3 and splay trees. However, these are more difficult to implement and aren't a vast improvement in either space or time complexity, hence weren't employed.

In conclusion, each data structure has numerous pros and cons, and is best suited for a definite circumstance; but it can be generalized that the bloom filter has the best overall performance, the balanced tree possesses the best worst-case operation, and the BST exhibits decent average-case performance. The set membership problem is integral to the field of computer science, and multiple data structures can all be implemented to assist in solving it.

References:

“Time and Space Complexity analysis of Red Black Tree,” *OpenGenus IQ: Computing Expertise & Legacy*, Dec. 28, 2021. https://iq.opengenus.org/time-and-space-complexity-of-red-black-tree/

“automata - The importance of the membership problem,” *Computer Science Stack Exchange*. https://cs.stackexchange.com/questions/57154/the-importance-of-the-membership-problem (accessed Mar. 18, 2023).

“Applications, Advantages and Disadvantages of Red-Black Tree,” *GeeksforGeeks*, May 16, 2022. https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-red-black-tree/

Geeksforgeeks, “Breadth first search or BFS for a graph,” *GeeksforGeeks*, 21-Feb-2023. [Online]. Available: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/. [Accessed: 15-Mar-2023].

“Data Structure - binary search tree,” *Tutorials Point*, 2018. [Online]. Available: https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm. [Accessed: 15-Mar-2023].

Geeksforgeeks, “Binary search tree,” *GeeksforGeeks*, 15-Feb-2023. [Online]. Available: https://www.geeksforgeeks.org/binary-search-tree-data-structure/. [Accessed: 15-Mar-2023].

“Bloom Filters by Example,” *llimllib.github.io*. https://llimllib.github.io/bloomfilter-tutorial/

E. van Baaren, “Bloom Filter in Python: Test If An Element is Part of a Large Set,” *Python Land*, Mar. 16, 2021. https://python.land/bloom-filter (accessed Mar. 18, 2023).