There are a vast number of real-life applications to the Set Membership problem whether in databases, detecting malicious URLs, or while determining the uniqueness of a file name. Below, we have analyzed the time and space complexity of four different algorithms that implement search and insert operations for sets.

Sequential Search *Theoretical Analysis:*

Space Complexity: The space complexity of the Sequential Search Set class is determined by the size of the array containing the elements of the set. Thus, the space complexity is O(n), for both insertion and search methods, where n is the size of the set.

Time Complexity:

Search: We need to iterate through the elements of the array to find the search element, which gives a time complexity of O(n) in the worst (when the element being searched is at the end of the list or not present in the list) and average cases, but O(1) in the best case, where the search element is the first element.

Insert: Insertion of the element always has a time complexity O(1), since the element is inserted at the end of the array data structure. However, since we also perform a search before insertion to ensure uniqueness, the worst-case time complexity is O(n).

Experimental Analysis:

	InsertElement time (seconds)	Search_element.txt search time(seconds)
test1-mobydick.txt	3.165659	0.017063
test2-warpeace.txt	6.329319	0.015580
test3-dickens.txt	168.555525	0.048499

In our experiment, the time taken for search operations varied across different test files due to their varying sizes and word distributions. When discussing the big O notation, it is crucial to remember that it provides an upper bound for the growth rate of an algorithm's time complexity. Our experiment demonstrated that the actual performance of SequentialSearchSet might vary depending on the input data, even though the worst-case time complexity remains consistent. In conclusion, the experiment showed that SequentialSearchSet's performance depends on the characteristics of the input data. Stress tests highlight the correlation between larger sets and search times, with the Dickens and large synthetic test files (5 million elements) taking 3 times as long to perform search operations. Thus, the SequentialSearchSet algorithm is easy to implement for small sets, but structures like BalancedSearchTreeSet may offer more efficient solutions for specific use cases, particularly larger sets.

Binary Search Tree *Theoretical Analysis:*

Space Complexity: The space complexity of searching and inserting elements in our iterative implementation of the search tree is always O(1) as we only ever allocate one variable “element” which is overwritten in each iteration. This means the space complexity will always be O(1).

Time Complexity: Since each node can have 2 children, the height of the tree is proportional to the log of the number of nodes, and determines how many nodes need to be visited in search and insertion operations. The average time complexity for search and insert operations will therefore be O(logN). In the worst case where the tree is completely unbalanced and only one node is stored at each depth, the time complexity for search and insert operations will be O(N).

Experimental Analysis:

	InsertElement time (seconds)	Search_element.txt search time(seconds)
test1-mobydick.txt	0.178977	0.000502
test2-warpeace.txt	0.503637	0.000489

test3-dickens.txt	4.608968	0.000790

The table above displays data regarding the Search and Insert operations for Binary Search tree for each text file. The file "test1-mobydick.txt" contains 209329 words, whilst the file "test2-warpeace.txt" contains 564236 words and file "test3-dickens.txt" contains 5149661 words. These operations are much faster than the same operations performed on sequential search due to the average case insert and search time of O(log(N)) as previously mentioned.

The third file is different in that it consists of only one line of text and also has a much larger size. To ensure the prolonged runtime on file "test3-dickens.txt" was due to its size, and not due to any other external factor such as the file being 1 line or the file being sorted to a higher degree, we made functions to evaluate the file. The first function evaluates how sorted a function is to a percentage value(the more sorted the list the worse the time complexity) and the second function finds the height of the tree. The text file 'test3-dickens.txt' was 48.72% sorted whilst the other two text tiles were respectively 49.12% and 48.73% sorted. When the text files were limited to the first 100,000 words, insertion of the file 'test3-dickens.txt' resulted in a tree of depth 30, whilst insertion of the other two files resulted in trees of depths 34 and 32. Both of these tests suggest that the file 'test3-dickens.txt' is actually more suitable for insertion into a tree, and we can therefore attribute the extended element insertion time to the size of the file.

Although none of the files resulted in a skewed tree we still must consider cases where the tree is skewed. To account for this edge case we made two data generators one which produces a list of sorted strings and one which produces a list of randomly sorted strings. As seen in the graph the sorted data results in a degenerative tree with one node per level of depth and thus makes insert operations considerably slower, reducing the element insertion time complexity to O(N). The same trend is reflected with Search operations. With this in mind BST should not be used for data which may be sorted.

Balanced Search Tree *Theoretical Analysis:*

*Space Complexity:*The space complexity of the searchElement method in a LLRB balanced search tree is O(1) because it does not require any additional space to perform the search. The search is performed by traversing the tree in a top-down manner until the target element is found, and no additional data structures are used to store the traversal path. The space complexity of the insertElement in a LLRB balanced search tree is O(log n) because the method needs to create new nodes to insert the element, and the height of the tree is proportional to log n. Therefore, the maximum space required to insert an element in a LLRB balanced search tree is O(log n).

Time Complexity:

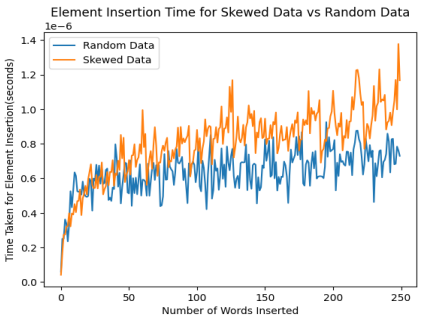
Search: The time complexity for searchElement() in a Red-Black Tree is O(log n) in both worst and average cases, where n represents the number of nodes in the tree. This is because each search operation requires traversing the tree from the root to a leaf node, and a well-balanced Red-Black Tree has a height of approximately log n.

Insert: The time complexity for inserting an element in a Red-Black Tree is O(log n) in both worst and average cases, where n is the number of nodes in the tree. This is because each insertion operation involves traversing the tree from the root to the appropriate leaf node, and since the tree is a type of Red-Black tree, the height of the tree is approximately log n.

Experimental Analysis:

	InsertElement time (seconds)	Search_element.txt search time (seconds)
test1-mobydick.txt	0.418644	0.000635
test2-warpeace.txt	0.781082	0.000626
test3-dickens.txt	7.256694	0.001198

From the experimental data, we can see that the insertion time for the LLRB tree implementation increases with the number of words in the test files. The result on the table is consistent with the expected O(log N) complexity, as the time complexity grows logarithmically with the size of the input. The search time, on the other hand, is relatively consistent across all test files. This is because the search operation in a balanced search tree, such as LLRB, also has a time complexity of O(log N) for the worst and average cases. In the case of a balanced search tree, the worst-case and average-case complexities for both insertion and search operations are O(log n). This holds true even for edge cases like sorted data structures, as the self-balancing mechanism maintains the tree's balance, preventing it from degenerating into a linked list. As a result, the search and insertion operations remain efficient even in scenarios where the input data has a specific edge-case order. As you can see from the graph on the left, even with a sorted data structure, the BST algorithm performs logarithmic and there is minimum time loss. If we did not implement the self-balancing mechanism, the insertion and search times might be lower in the average case, but a sorted list would make the time complexity linear instead of logarithmic.



In conclusion, the experimental results demonstrate that the LLRB tree implementation provides efficient and scalable performance for both insert and search operations, for both average and worst case scenarios, in line with the expected O(log N) complexity. This makes it a suitable data structure for applications that require fast and efficient search and insertion operations.

Bloom Filter

Bloom filters work by performing several hash functions on the element to be inserted and changing the value of the bitarray according to the outcome of each hash function. They are both space and time efficient, but have a tradeoff of false positives. For example, from the image, we can see that *w,* though not present in the set, would appear to be. Thus, it is incredibly important that the bit array is of appropriate size and the hash functions chosen produce evenly distributed values.

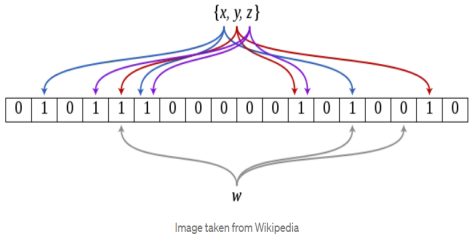


Image taken from Wikipedia

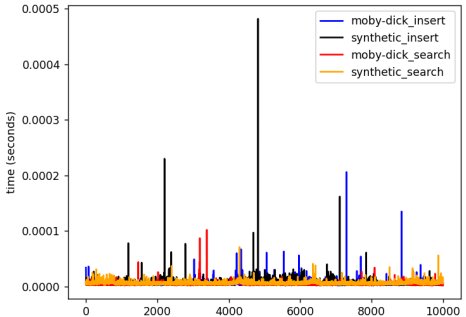
Theoretical Analysis:

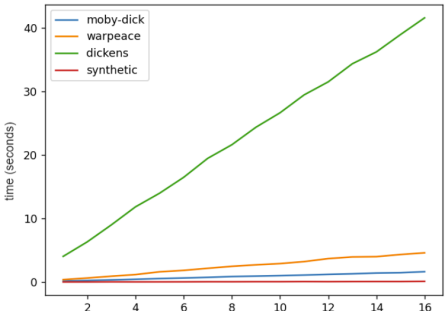
Space Complexity: We have constant space complexity in every possible case since the size of the bit array remains the same regardless of the numbers of elements inserted. No auxiliary memory is required for the search and insertion methods, so if the size of the bitarray is *n*, it has O(*n*) space complexity.

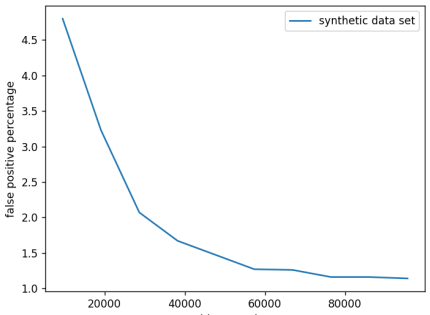
Time Complexity: Insert: To insert an element, *m* hash functions must be computed. If each hash function takes O(1) time, inserting an element takes O(m) time, and inserting *k* elements takes O(mk) time, in every possible case. *Search:* While searching for an element, if the value of any one hash function does not correspond with the bit array values, the element is definitely not in the set. So if the search element is a member of the set, all *m* hash functions need to be computed, thus giving a worst case of O(*m*). In the best case, the first hash function detects non-membership, giving O(1) time. The average case has a time complexity between O(1) and O(*m*).

Experimental Analysis:

From experimental analysis of the runtime, trends for the insert and search methods accurately reflect those from the theoretical analysis. Below are tests involving real and synthetic data (file of 10,000 random strings, each with a maximum length of 10 characters).





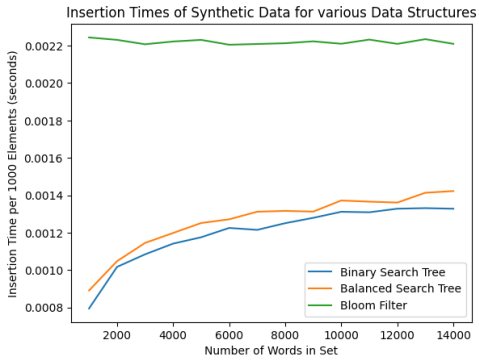


From Figure 1, when the number of hashes is constant, time taken to insert and search for nth element is roughly the same, for real and synthetic data, with variations of roughly 0.0002 seconds. While from Figure 2 the number of hashes against time taken to insert the entire file, is linear - as expected (more hashes require more computations for each element), with larger text files taking longer time. In figure 3, we stress tested the size of the bit array for synthetic data (a set of 10,000 elements) to find the optimum bit array size to achieve a false positivity rate of 1%. The synthetic data consisted of random strings so that by searching with the search_element.txt file, any found element was a guaranteed false positive. We concluded that to maintain a 1% rate of false positives, the optimum size of the bit array size is approximately 9.5 times the size of the set.
All the above trends stay consistent for real and synthetic data, and we can safely assume that the only factors affecting runtime for insert and search are the number of hash functions, and the efficiency of the hash function itself. There are no edge cases, since the element itself is not stored, so search and insert times remain constant for any element for a constant number of hash functions - which is seen from the table below.

	InsertElement time (seconds)	Search_element.txt search time(seconds)
test1-mobydick.txt	0.614399	0.001864
test2-warpeace.txt	1.623703	0.001852
test3-dickens.txt	15.845035	0.001934

However, average testing times show that Bloom is in fact slower for the insertion and search methods, so if limited storage is not a concern, Balanced and Binary Search Trees should be preferred.

Conclusion



Under different cases, each of the sets prove more useful. Though it is the least efficient, while dealing with a small set that does not require fast operations, a Sequential Search Set is easiest to implement because it has low overhead. However, that is almost never the case, and the need arises for faster solutions.
In cases where we have incredibly large sets with limited storage, and absolute precision is not a requirement, Bloom Filter Sets are most suitable. They have a significant advantage over the other three data structures in terms of space efficiency, since they don't store the actual element. However, since it is a probabilistic data structure - we can only tell if an element is possibly in the set, or definitely not in the set. Thus, bloom filters are useful when an imperfect set membership test can be used for large datasets, for example when checking if a username is unique.
Though Bloom filters are space efficient, from experimental analysis we've observed that Binary and Balanced Search Trees have faster insertion and search times, which can be seen from the graph. Analysis of search times also produced a graph of the same pattern, so we can conclude that if faster search and insertion operations are a priority, the Tree sets should be the preferred algorithms. Depending on the kind of data the set consists of, we can choose which tree to use. If the data being inserted is sorted, Balanced search trees should be used to ensure a runtime of log(n). On the other hand, if it is unlikely the data is sorted, Binary Search Trees should be preferred as they will also have a log(n) runtime, with a lower overhead cost due to absence of balancing operations present in Balanced Search Tree.
Thus, under different scenarios, different algorithms prove more useful, and according to the conditions required - having a large set, a limited amount of memory storage, and the need to have fast insert and search operations - the most appropriate algorithms should be selected.