Final Project

Insertion and Search Times for Hash Tables, Linked List, and Binary Search Tree.

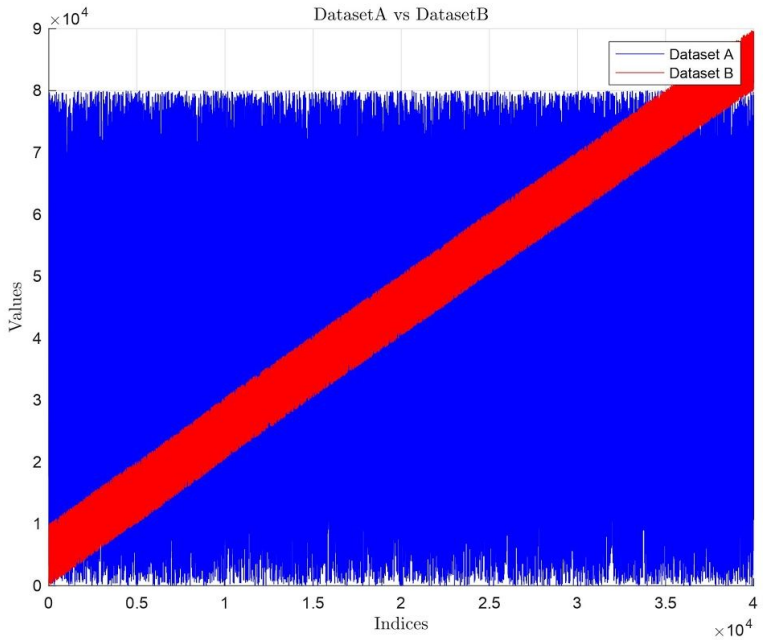
By:
David Wade
Kartik Sharma

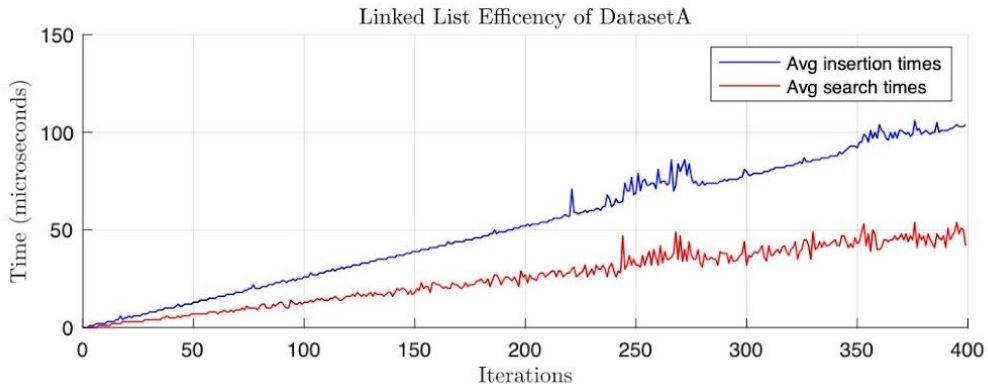
Summary:

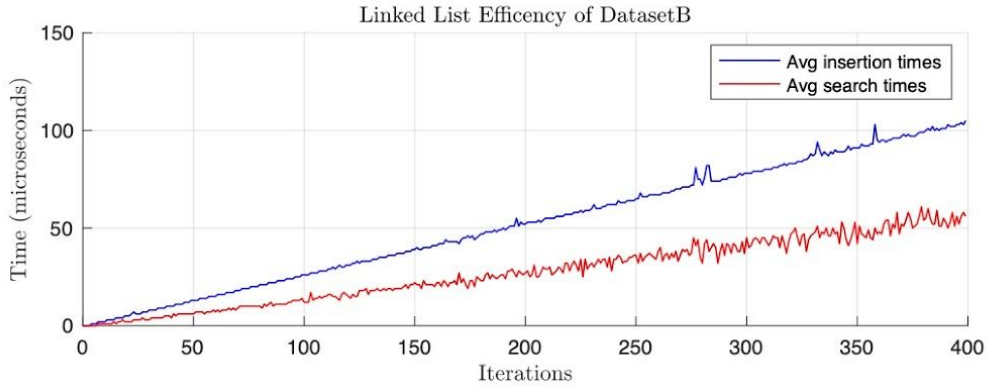
After testing the data structures, in essence, hash tables were overall superior than linked list and binary search trees with respect to its insertion and search times. In specific, hash tables with chaining collisions resolution had the lowest average insertion and search times. This is because for huge data sets such as this USPS package tracking application, in order to add an item to a linked list it will have to traverse the whole list until it finds the end of the list, and then adds the item to the list. This will take longer than the other data structures because as the list gets larger, it will have to traverse more nodes in order to find the end of the list. Thus, in the graphs linked lists insertions have a positive linear correlation between iterations and time, as the time it takes to insert an item will be larger when the number of elements increases. Similar to its insertion, linked list search times are also higher than the other data structures because it has to go through each element in the list, from start to end, to find the particular node.

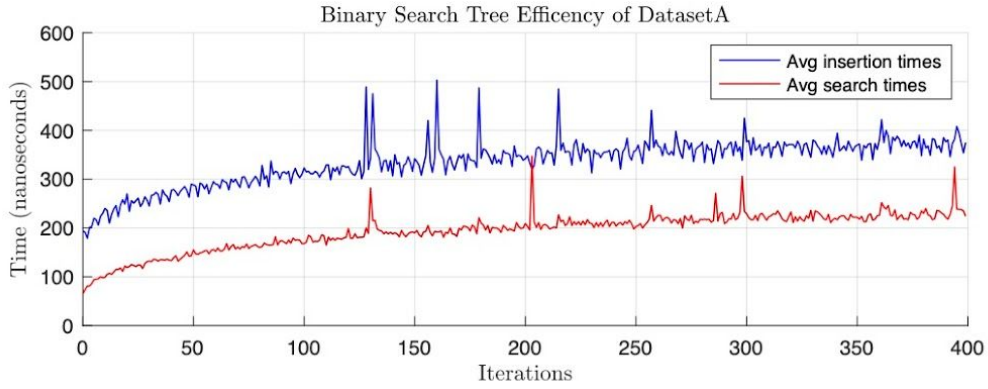
Furthermore, binary search tree insertion and search times are better than linked lists, but still worse than hash tables. Close to linked list, a binary search tree insertion also traverses the tree until it finds the correct position, however it's better than the linked list because it doesn't have to traverse the whole tree, due to its left and right node properties. Also, searching in a binary search tree is better than a linked list but still worse than hash tables with respect to time. This is because it still has to traverse the tree in order to find the value, but it doesn't need to search the whole tree since it can go to each node and see if it's larger or smaller, which will allow it to never traverse the whole tree.

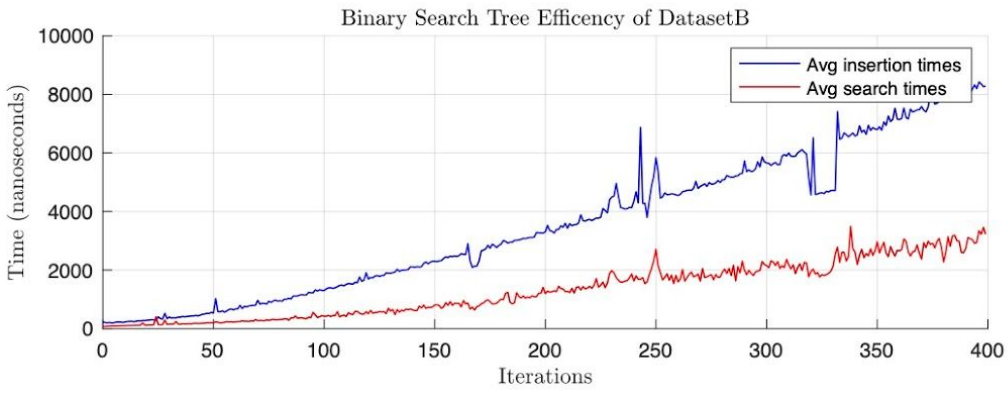
Moreover, hash tables have the lowest average insert and search times compared to linked list and binary search trees, due to its hash function attribute. The hash function allows for an item to be added at that specific index, unless there is a collision, and when searching for a particular value it can also be found at that index. However, when a collision results, chaining, linear, or quadratic probing can be used to fix that collision. After testing it with the two data sets, one with repetitive numbers and one with more unique numbers (as pictured below), it was apparent that the chaining collision resolution had the lowest search and insert timings. This could be because of the fact that in chaining, there is a linked list at every index of the hash table, which allows for the item to be inserted in that linked list if it results in the same hash function value. Thus, when searching for that item all that needs to be done is to go to that specific index and search the linked list, instead of traversing the hash table until finding an empty space. Evidence of our conclusions can be seen on the numerous graphs below.

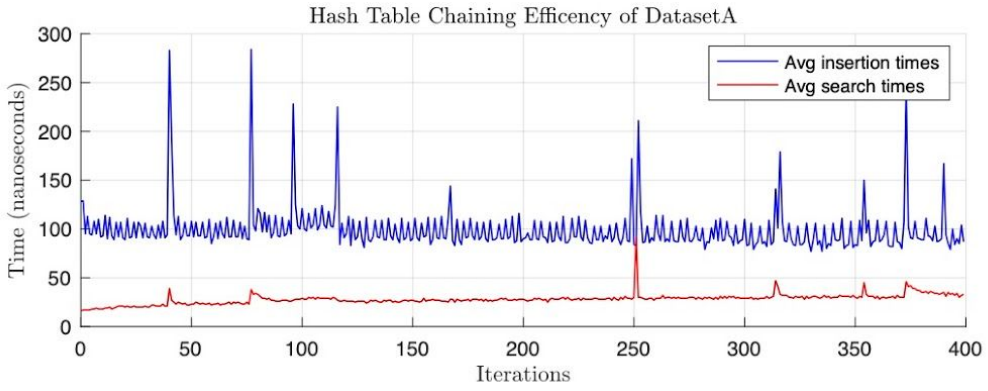


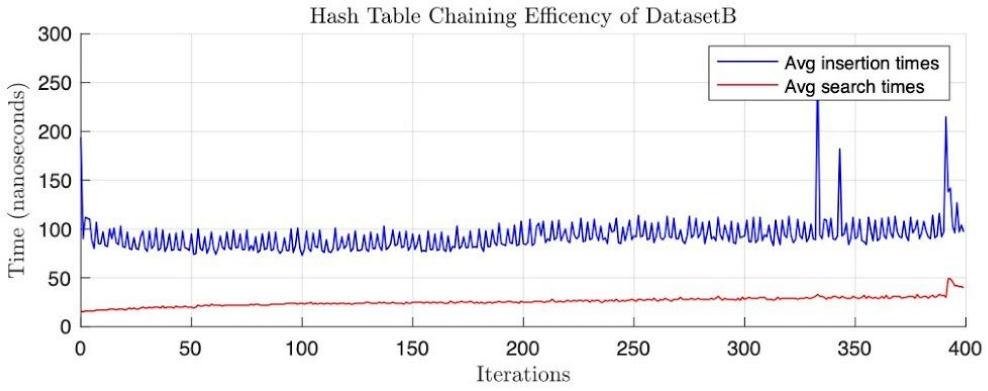

Linked List Efficency of DatasetA
Avg insertion times
Avg search times
Time (microseconds)
Iterations
0
50
100
150
200
250
300
350
400

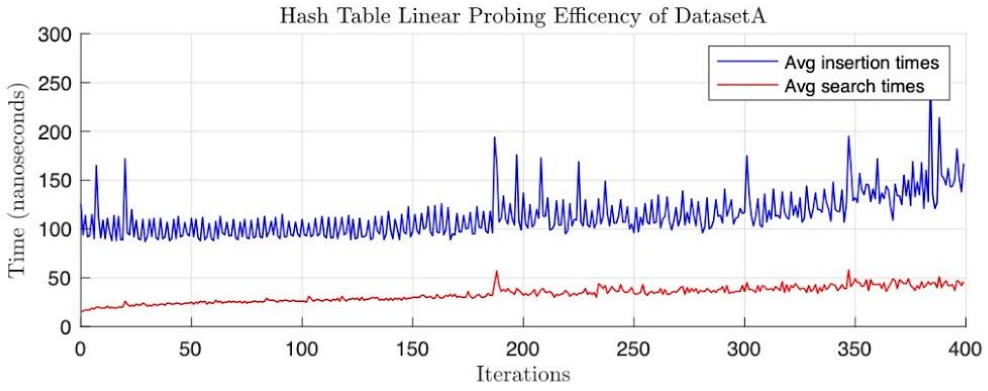

Linked List Efficency of DatasetB
Avg insertion times
Avg search times
Time (microseconds)
Iterations
0
50
100
150
200
250
300
350
400

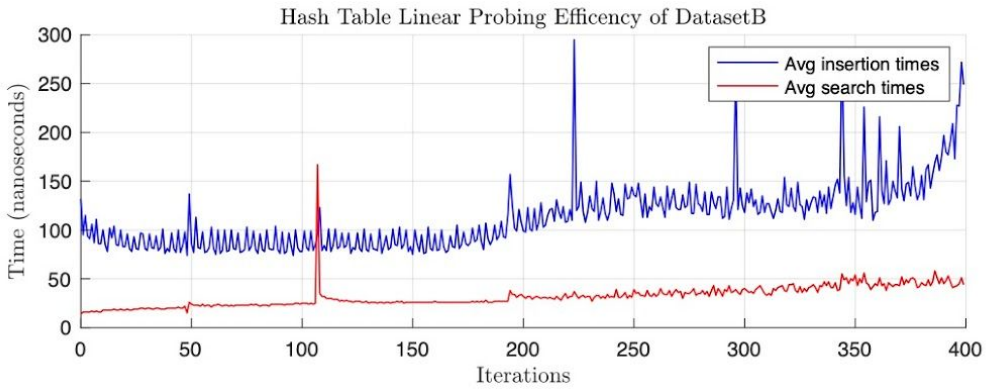

Binary Search Tree Efficency of DatasetA
Avg insertion times
Avg search times
Time (nanoseconds)
Iterations
0
100
200
300
400
500
600
50
150
250
350

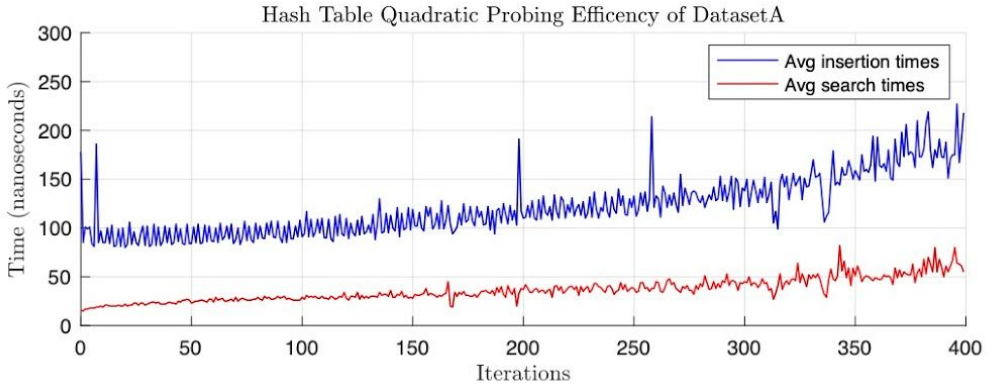

Binary Search Tree Efficency of DatasetB
Avg insertion times
Avg search times
Time (nanoseconds)
Iterations
0
2000
4000
6000
8000
10000
50
100
150
200
250
300
350
400

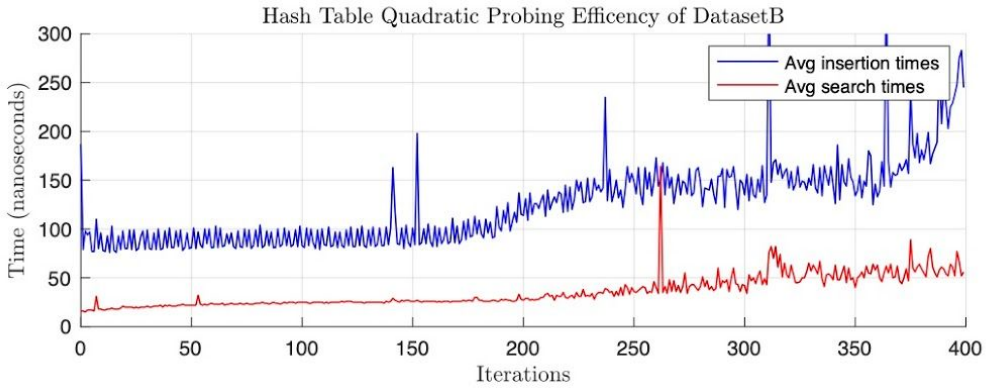

Hash Table Chaining Efficency of DatasetA
Avg insertion times
Avg search times
Time (nanoseconds)
Iterations
0
50
100
150
200
250
300
350
400

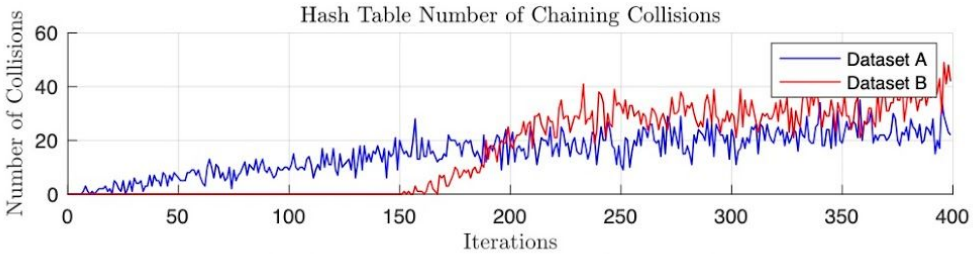

Hash Table Chaining Efficency of DatasetB
Avg insertion times
Avg search times
Time (nanoseconds)
Iterations
0
50
100
150
200
250
300
350
400

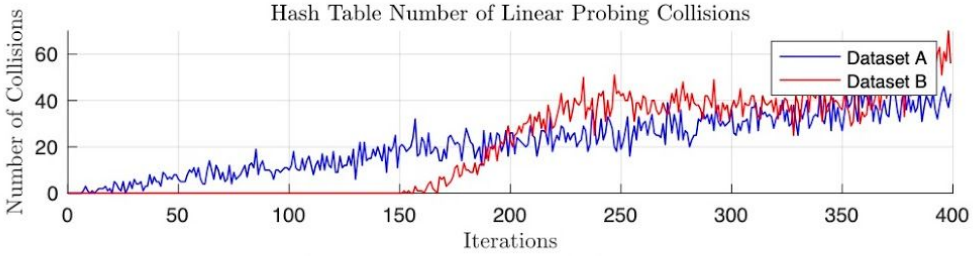

Hash Table Linear Probing Efficency of DatasetA
Avg insertion times
Avg search times
Time (nanoseconds)
Iterations
0
50
100
150
200
250
300
350
400

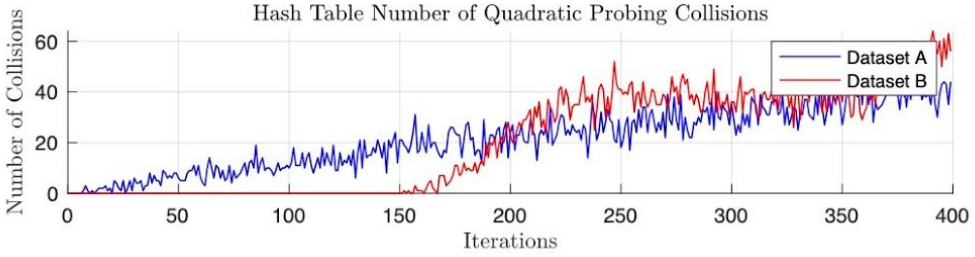

Hash Table Linear Probing Efficency of DatasetB
Avg insertion times
Avg search times
Time (nanoseconds)
Iterations
0
50
100
150
200
250
300
350
400

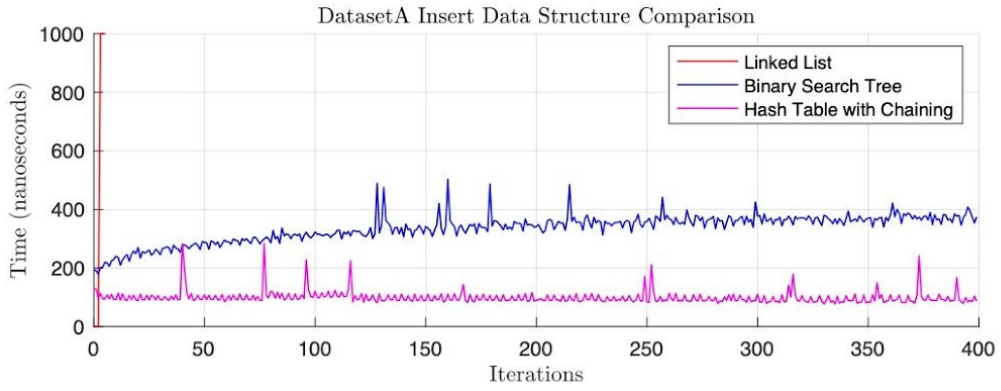

Hash Table Quadratic Probing Efficency of DatasetA
Avg insertion times
Avg search times
Time (nanoseconds)
Iterations
0
50
100
150
200
250
300
350
400

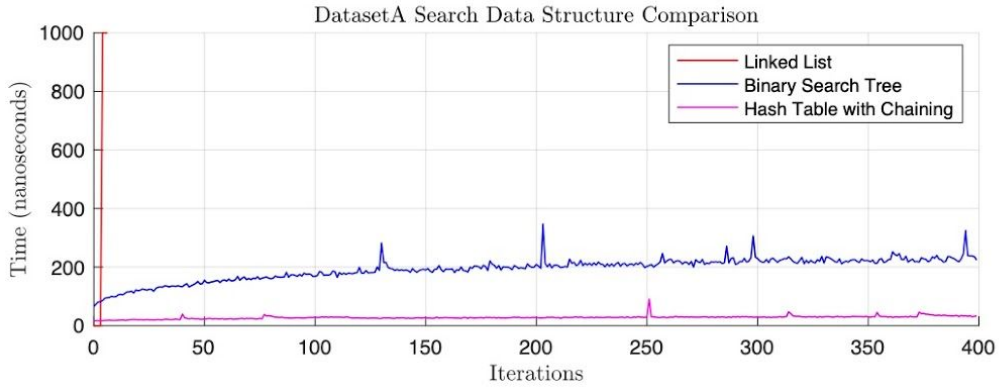

Hash Table Quadratic Probing Efficency of DatasetB
Avg insertion times
Avg search times
Time (nanoseconds)
Iterations
0
50
100
150
200
250
300
350
400

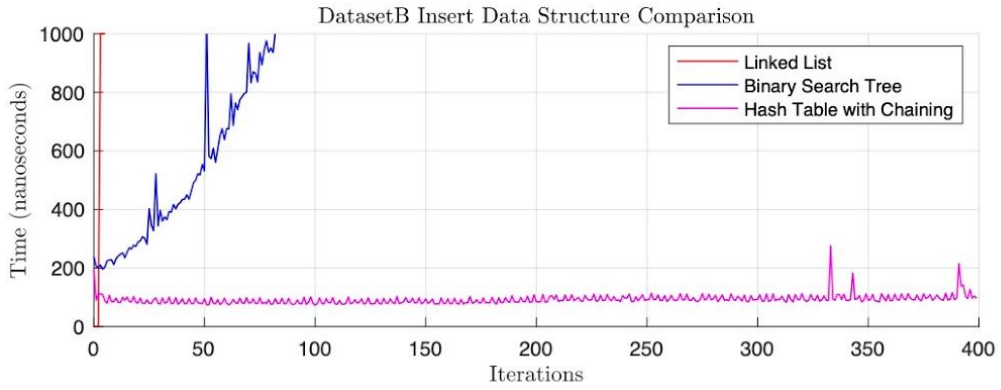

Hash Table Number of Chaining Collisions
Dataset A
Dataset B
Number of Collisions
Iterations
0
20
40
60
0
50
100
150
200
250
300
350
400

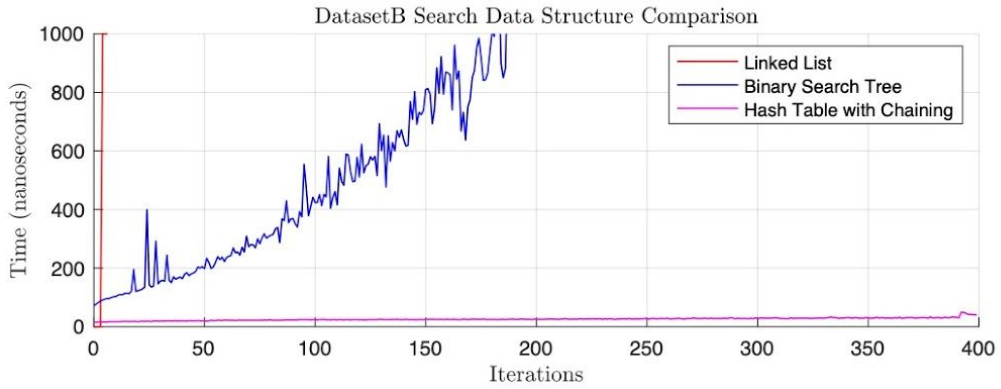

Hash Table Number of Linear Probing Collisions
Dataset A
Dataset B
Number of Collisions
Iterations
0
20
40
60
0
50
100
150
200
250
300
350
400


Hash Table Number of Quadratic Probing Collisions
Dataset A
Dataset B
Number of Collisions
Iterations
0
20
40
60
0
50
100
150
200
250
300
350
400


DatasetA Insert Data Structure Comparison
Linked List
Binary Search Tree
Hash Table with Chaining
Time (nanoseconds)
Iterations
0
200
400
600
800
1000
0
50
100
150
200
250
300
350
400


DatasetA Search Data Structure Comparison
Linked List
Binary Search Tree
Hash Table with Chaining
Time (nanoseconds)
Iterations
0
200
400
600
800
1000
0
50
100
150
200
250
300
350
400


DatasetB Insert Data Structure Comparison
Linked List
Binary Search Tree
Hash Table with Chaining
Time (nanoseconds)
Iterations
0
200
400
600
800
1000
0
50
100
150
200
250
300
350
400


DatasetB Search Data Structure Comparison
Linked List
Binary Search Tree
Hash Table with Chaining
Time (nanoseconds)
Iterations
0
200
400
600
800
1000
0
50
100
150
200
250
300
350
400