Kareem Saleh

README: Data Structure Performance

Introduction

This document provides an overview of the Data Structure Performance, specifying its components, walkthrough and functions.

Features

1) Create multiple data structures to store integers

2) Insert random integers in each data structure

3) Search random integers in each data structure

4) Delete random integers in each data structure

5) Repeat steps 2,3,4 for 1000 elements, 10000 elements, 100000 elements

6) Measuring time taken to carry out operations in each data structure

7) Measuring memory used to carry out operations in each data structure

8) Comparing different data structure performances for each data structure through time and memory usage

9) Exploration: Trying one of the operations for 1 million elements (not included in requirements)

Components

1. AvlTree (BST invariant)
   • BST invariant
   • Implements insertion, deletion, and search of integers
   • Preserves a balanced tree after each insert or delete (a tree is balanced when each parent’s left and right subtrees has a height difference <= 1. Also, it has to be the case for all levels of the tree)
   • If an imbalance occurs, then the Avl tree self balances using a series of single left, single right rotations and/or double left, double right rotations.

2. LinkedList (A class employing a Doubly Linked List)
   • Implements insertion, removal, and iteration.
   • Used to solve collisions in the Hash Table through chaining

3. SplayTree (BST invariant)
   • Implements insertion, deletion, and search of integers
   • Doesn’t immediately balance tree after each insert or delete, but splays most recent elements to the root of the tree
   • A splay is a series of zig-zig and/or zig-zag rotations aiming to transport the recent nodes to the root

4. HashChaining (Hash Table)
   • Child class of LinkedList.
   • Provides hash function of modular divison by the capacity of the table (array)
   • Each input integer is mapped to a position (index) in the table
   • Implements insertion, deletion and search
   • When a collision (2 inputs having the same output index), it is solved by creating a linked list of colliding elements at their common index

5. HashQuad (Hash Table)
   • Provides hash function of modular divison by the capacity of the table (array)
   • Each input integer is mapped to a position (index) in the table
   • Implements insertion, deletion and search
   • When a collision (2 inputs having the same output index), it is solved by quadratic probing which adds i^2 (where i starts from 1 to capacity of table)

6. Main.java
   • Driver program that runs and tests the data structure performance

---

Usage Instructions

Run the program

Main inserts n random integers into each data structure

Main searches n random integers in each data structure

Main deletes n random integers in each data structure

Main calculates time taken for each of those operations

Main calculates memory used for each of those operations

Main prints table of these values, evaluating performance

Order of Growth (worst case) :-

Linked List Class:

- **insert(Integer a)**:

- insert is of order O(1) since we have last node and we are adding to the end by just incrementing last and placing element, which takes constant time and doesn’t depend on the size of the list.

- **remove(Integer y)**:

- remove is of order O(n) since are crawling over the whole list using a while loop. But the removal as soon as we get to the element is O(1)

- **contains(Integer y)**

- contains is of order O(n) since are crawling over the whole list using a while loop.

- **LLdoClear()**:

- LLdoClear() is of order O(1) since we just make first and last nodes point to null, which takes constant time and doesn’t depend on the size of the list.

- **iterator()**:

- iterator is of order O(1) since we just return a new LLiterator object that doesn't require traversal of the list , which takes constant time and doesn’t depend on the size of the list.

- **isEmpty()**:

- isEmpty is of order O(1) since we are just checking if first is null, which takes constant time and doesn’t depend on the size of the list.

- **hasNext()** (from LLiterator):

- hasNext is O(1) since we are just checking if the current node is null, which takes constant time and doesn’t depend on the size of the list.

- **next()** (from LLiterator):

- next is O(1) since we are just accessing the current node's data and moving to the next node, which takes constant time and doesn’t depend on the size of the list.

AvlTree Class:

- **Avl_height():**

- Avl_height() is O(1) since it is just checking whether a statement is true or false and returns an int accordingly

- **rotateWithRightChild():**

- rotateWithRightChild() is O(1) since it is just a left rotation with the right chid and only includes pointer reassignment which is O(1).

- **rotateWithLeftChild():**

  - rotateWithLeftChild() is O(1) since it is just a right rotation with the left chid and only includes pointer reassignment which is O(1).

- **doubleWithRightChild():**

  - doubleWithRightChild() is O(1) since it is just a double rotation with the right child, which only includes pointer reassignment and other single rotate functions which are O(1).

- **doubleWithLeftChild():**

  - doubleWithLeftChild() is O(1) since it is just a double rotation with the right child, which only includes pointer reassignment and other single rotate functions which are O(1).

- **Avl_balance():**

  - Avl_balance() is O(1) since it includes rotations which are all O(1)

- **Avl_insert (int x, AvlNode t):**

  - Avl_insert (int x, AvlNode t) is O(log n) : it has two parts

  - Avl_balance() is O(1)

  - Avl_insert (int x, AvlNode t) recursive calls that split the tree into half every call which adds up to O(log n)

- **Avl_delete (int x, AvlNode t):**

  - Avl_delete (int x, AvlNode t) is O(log n) : it has two parts

  - Avl_balance() is O(1)

  - Avl_delete(int x, AvlNode t) recursive calls that split the tree into half every call which adds up to O(log n)

- **Avl_contains (int x, AvlNode t):**

  - Avl_contains (int x, AvlNode t) is O(log n) : Avl_contains (int x, AvlNode t) recursive calls that split the tree into half every call which adds up to O(log n)

- **findMin (int x, AvlNode t):**
  - findMin (AvlNode t) is O(log n) : it has findMin (int x, AvlNode t) recursive calls that split the list into half every call which adds up to O(log n)

SplayTree Class:

- **l_zig(SplayNode s1) – single left rotation**
  - SplayNode l_zig(SplayNode s1) is O(1) since it only includes pointer assignments
- **r_zig(SplayNode s1) – single right rotation (zig)**
  - SplayNode r_zig(SplayNode s1) is O(1) since it only includes pointer assignments
- **l_rots_zigzig(SplayNode s1) – left-left (zig-zig) double rotation**
  - l_rots_zigzig (SplayNode s1) is O(1) since it only includes pointer assignments
- **r_rots_zigzig(SplayNode s1) – right-right (zig-zig) double rotation**
  - r_rots_zigzig (SplayNode s1) is O(1) since it only includes pointer assignments
- **lr_rots_zigzag(SplayNode s1) – left-right (zig-zag) double rotation**
  - lr_rots_zigzag (SplayNode s1) is O(1) since it only includes pointer assignments
- **rl_rots_zigzag(SplayNode s1) – right-left (zig-zag) double rotation**
  - rl_rots_zigzag (SplayNode s1) is O(1) since it only includes pointer assignments

- **splay(SplayNode s1):**

- splay is O(n) since it contains a while loop with is O(n) and inside the loop are O(1) rotations
- rotateWithRightChild() is O(1) since it is just a left rotation with the right chid and only includes pointer reassignment which is O(1).

- **Splay_insert (int x, AvlNode t):**

  - Splay_insert (int x, SplayNode t) is O(n) : it has two parts
  - splay is O(n)
  - Splay_insert (int x, SplayNode t) recursive calls that split the tree into half every call which adds up to O(log n)
  - So O(n) dominates

- **Splay_delete (int x, AvlNode t):**
  - Splay_delete (int x, SplayNode t) is O(n) : it has four parts
  - splay is O(n)
  - SplayMax is O(n)
  - Splay_contains is O(log n)
  - Splay_delete (int x, SplayNode t) recursive calls that split the tree into half every call which adds up to O(log n)
  - So O(n) dominates

- **Splay_contains (int x, AvlNode t):**
  - Splay_contains (int x, SplayNode t) is O(n) : it has two parts
  - splay is O(n)
  - Splay_contains (int x, SplayNode t) recursive calls that split the tree into half every call which adds up to O(log n)
  - So O(n) dominates

- **SplayMax (AvlNode t):**
  - SplayMax is O(n): has two parts
  - While loop accounting to O(n)
  - Splay_contains is O(log n)
  - So O(n) dominates

HashChaining Class:

- **HashCh(int x)**

- HashCh(int x) is O(1) since it is just has an assignment

- **HashCh_insert(int x)**

- HashCh_insert(int x) is O(1): it has two parts

- HashCh(int x) is O(1)

- insert from LinkedList is O(1)

- So O(1) dominates

- **HashCh_contains(int x)**

- HashCh_contains(int x) is O(n): it has two parts

- HashCh(int x) is O(1)

- contains from LinkedList is O(n)

- So O(n) dominates

- **HashCh_delete(int x)**

- HashCh_delete(int x) is O(n): it has two parts

- HashCh(int x) is O(1)

- remove from LinkedList is O(n)

- So O(n) dominates

- **nextPrime(int o)**

- nextPrime is O(n) since it has a for loop

HashQuad Class:

- **HashQ(int x)**

- HashCh(int x) is O(1) since it is just has an assignment

- **resize()**

- Quadratic probing improves the efficiency of the resizing by not looping over every single position in the inner loop and just rehashes the elements that are already there, making resize() O(n)

- resize() is O(n) even though the 2 for loops make it look like O(n^2)

- nextPrime is O(n)

- 2 nested for loops look like O(n^2) but inner loop is rehashing old elements so O(1), so total O(n)

- So O(n) dominates in worst case and O(n) dominates in avg case

- **HashQ_insert(int x)**

- HashQ_insert(int x) is O(n)

- HashQ(int x) is O(1)

- resize is O(n)

- So O(n) dominates

- **HashQ_contains(int x)**

- HashCh_contains(int x) is O(n) since it is based for loop is O(n)

- So O(n) dominates

- **HashQ_delete (int x)**

- HashQ_delete(int x) is O(n) since it is based on a foor loop

- **nextPrime(int o)**

- nextPrime is O(n) since it has a for loop

Main Class:

• main(String[] args)

- printTable(double[][] times, int[] n)

• HashCh_ops_time(int index, int m)

- HashCh_ops_time is O(n) since it is based on 3 single for loops.

• HashQ_ops_time(int index, int m)

- HashQ_ops_time is O(n) since it is based on 3 single for loops.

• Avl_ops_mem(int index, int m)

- Avl_ops_mem is O(n) since it is based on 3 single for loops.

• Splay_ops_mem(int index, int m)

-Splay_ops_mem is O(n) since it is based on 3 single for loops.

• HashCh_ops_mem(int index, int m)

- HashCh_ops_mem is O(n) since it is based on 3 single for loops.

• HashQ_ops_mem(int index, int m)

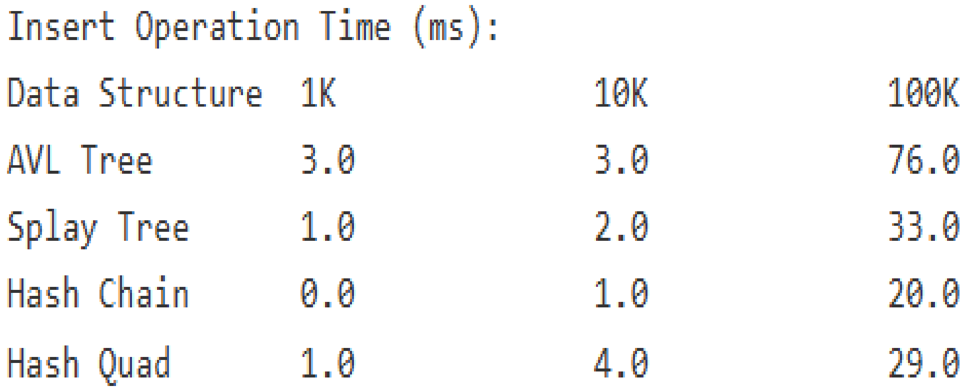
- HashQ_ops_mem is O(n) since it is based on 3 single for loops.

Time Test Cases:-

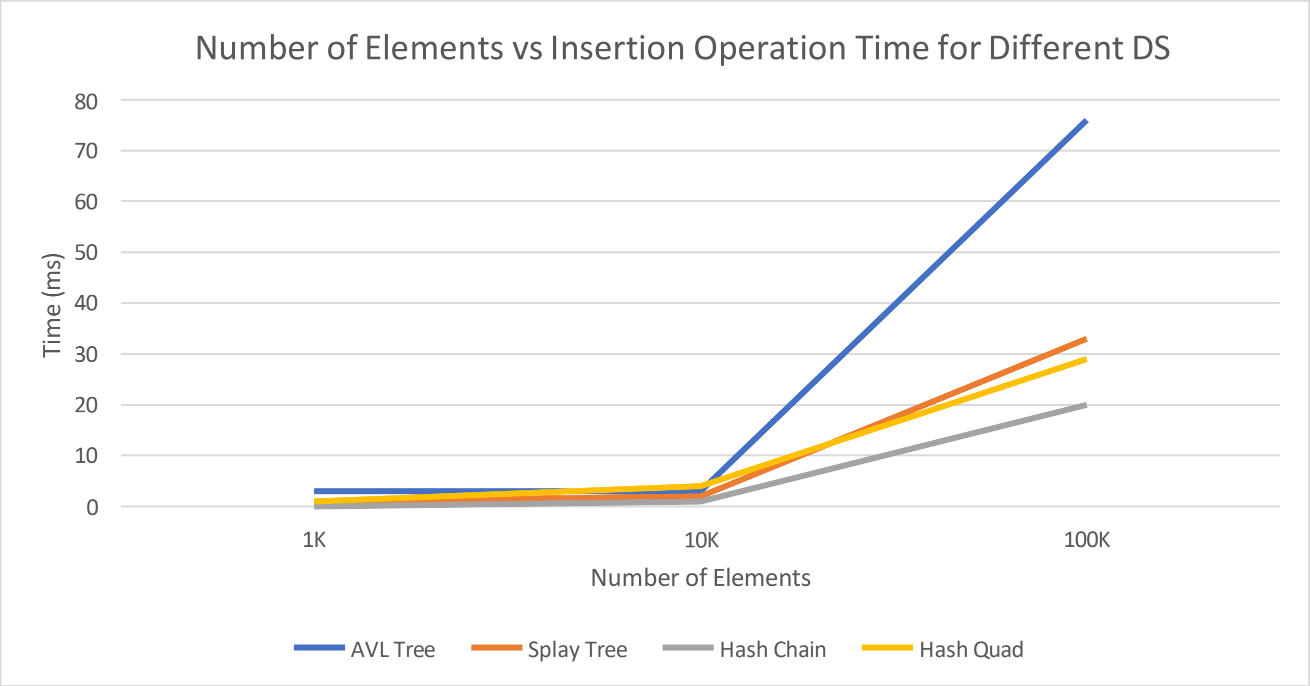
Insert Operations (Time):

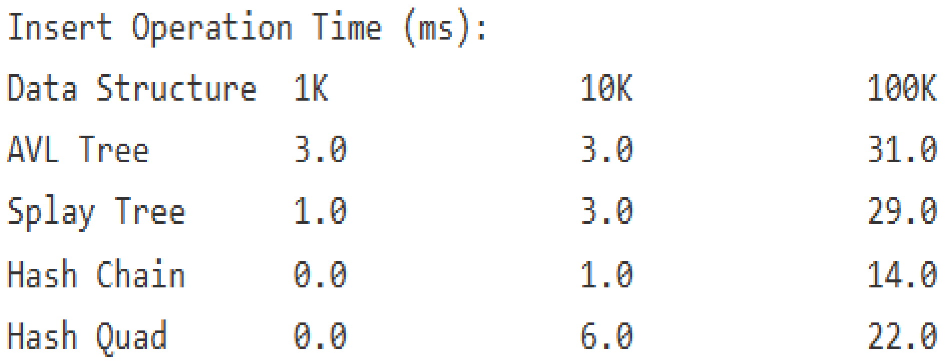
Hypothesis:

Based on the order of growth comparisons between functions in the data structure classes, we can attempt to predict the outcomes of the Insert operations times. Since the Hash Table (Chaining) utilizes the linked list insert which is O(1) and a hash function which is O(1), then it should be O(1). Since other DS inserts are O(log n) or O(n), then the Hash Table using Chaining has the fastest insert.



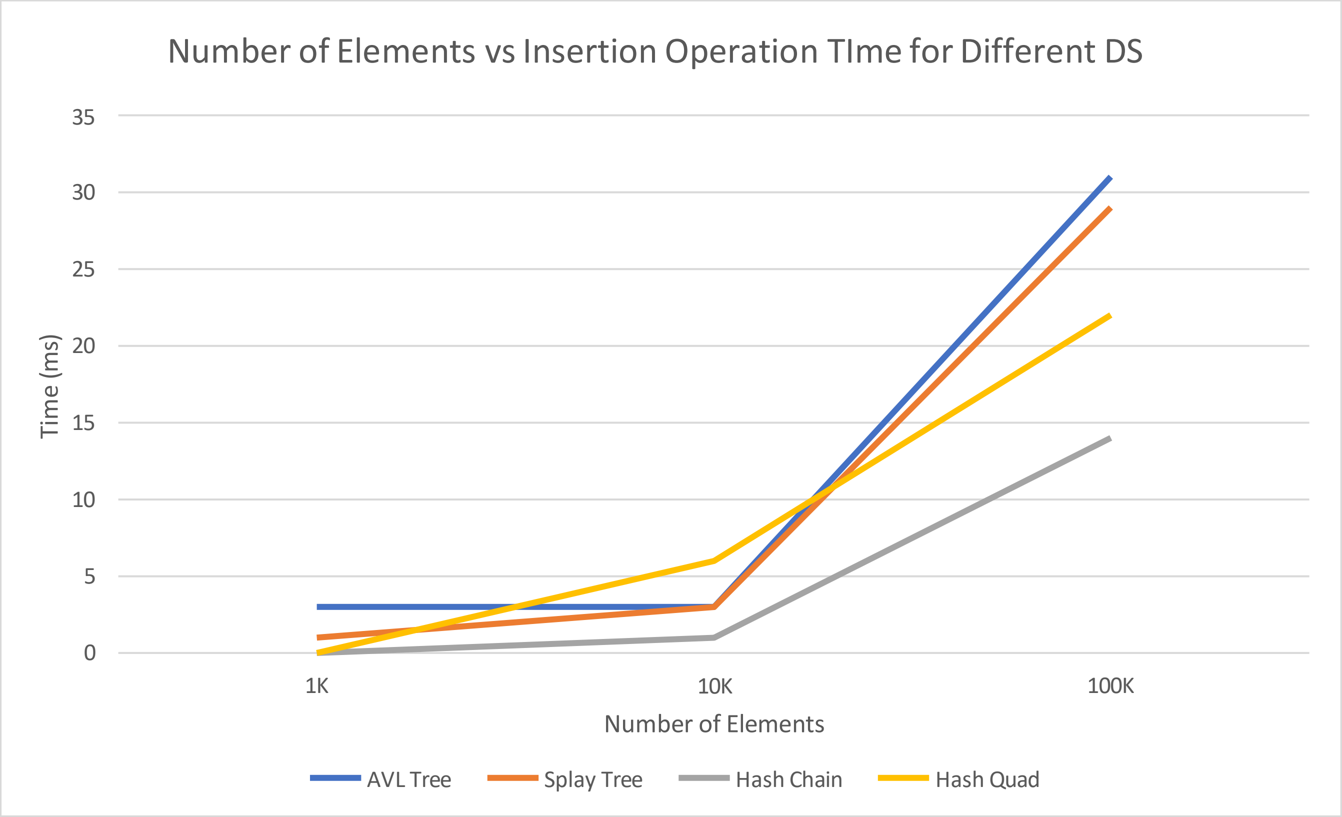
```
Insert Operation Time (ms):
Data Structure  1K              10K             100K
AVL Tree        3.0             3.0             76.0
Splay Tree      1.0             2.0             33.0
Hash Chain      0.0             1.0             20.0
Hash Quad       1.0             4.0             29.0
```

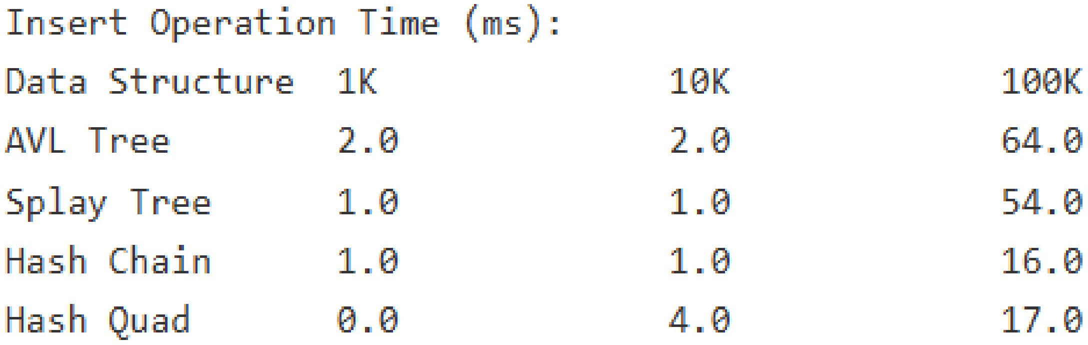




```
Insert Operation Time (ms):
Data Structure  1K              10K             100K
AVL Tree        3.0             3.0             31.0
Splay Tree      1.0             3.0             29.0
Hash Chain      0.0             1.0             14.0
Hash Quad       0.0             6.0             22.0
```

Kareem Saleh





```
Insert Operation Time (ms):
Data Structure  1K          10K         100K
AVL Tree        2.0         2.0         64.0
Splay Tree      1.0         1.0         54.0
Hash Chain      1.0         1.0         16.0
Hash Quad       0.0         4.0         17.0
```

Actual:

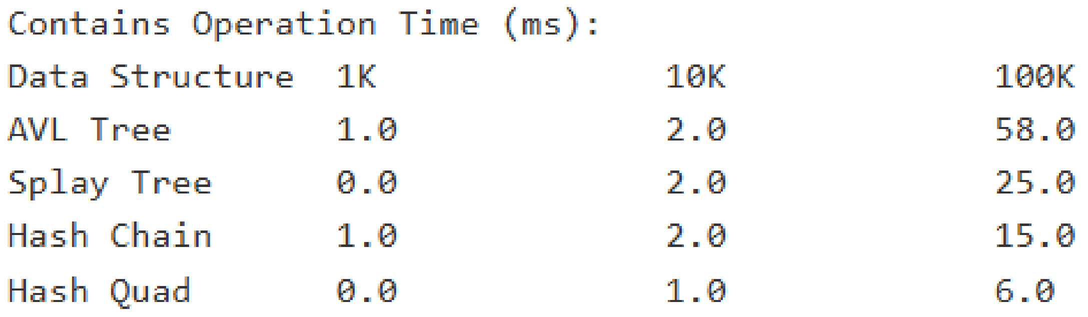
Based on the graphs and tables, the general trend indicates that Hash Table using Chaining has the fastest insert amongst the data structures for the small, medium and large datasets. Although we have some zeros, this does not mean it was done in no time, it just means that it was less than 1 ms . For the 10K elements, the speeds of inserts for splay and avl trees were similar, supporting the fact that they are both O(log n). The Hash Table (Quadratic Probing) insert is slower and it is justified through the O(n) complexity (caused by the resizethat it had (because O(log n) < O(n)). As for the 100K elements, Hash Table employing Chaining is still the fastest but, the Hash Table using Quadratic Probing has insert faster than the avl and splay trees' inserts even though they are O(log n). This could have happened because the Quadratic probing lead to a well-distributed table with fewer collisions due to spars population of the table.

Note:- 0 value here represents minimal or negligible time as opposed to no time at all.

Contains Operations (Time):

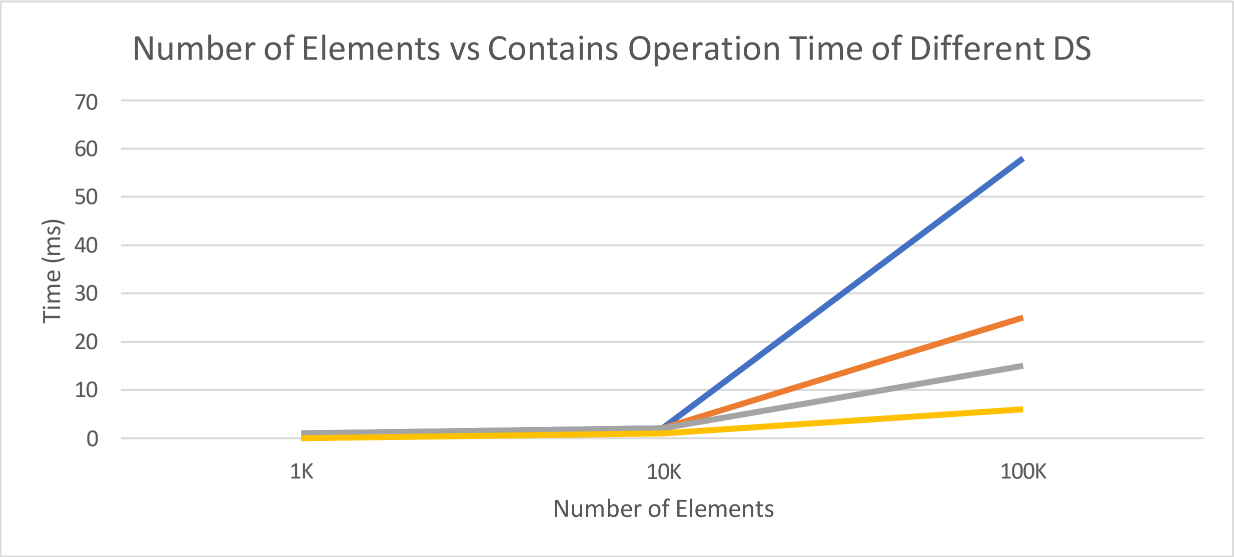
Hypothesis:

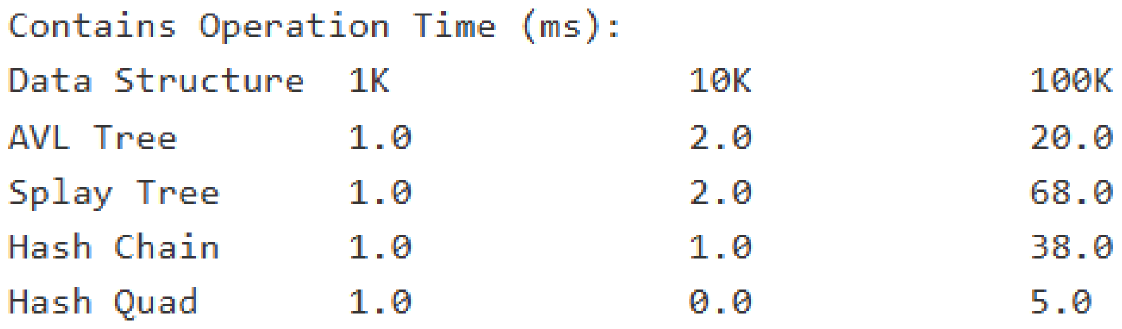
Based on the order of growth comparisons between functions in the data structure classes, we can attempt to predict the outcomes of the Contains operations times. Since the Avl Tree utilizes recursion splitting the tree in half every call, contains is O(log n). Even though splay tree also utilizes recursive calls splitting the tree in half every time, it also has a splay function to transport the most recently searched node to the root. Since splay has O(n) complexity, the O(n) complexity dominates the O(log n) of the recursive calls making the whole Splay Tree's contains be O(n).Since other DS contains are O(n), then the Avl Tree has the fastest contains.

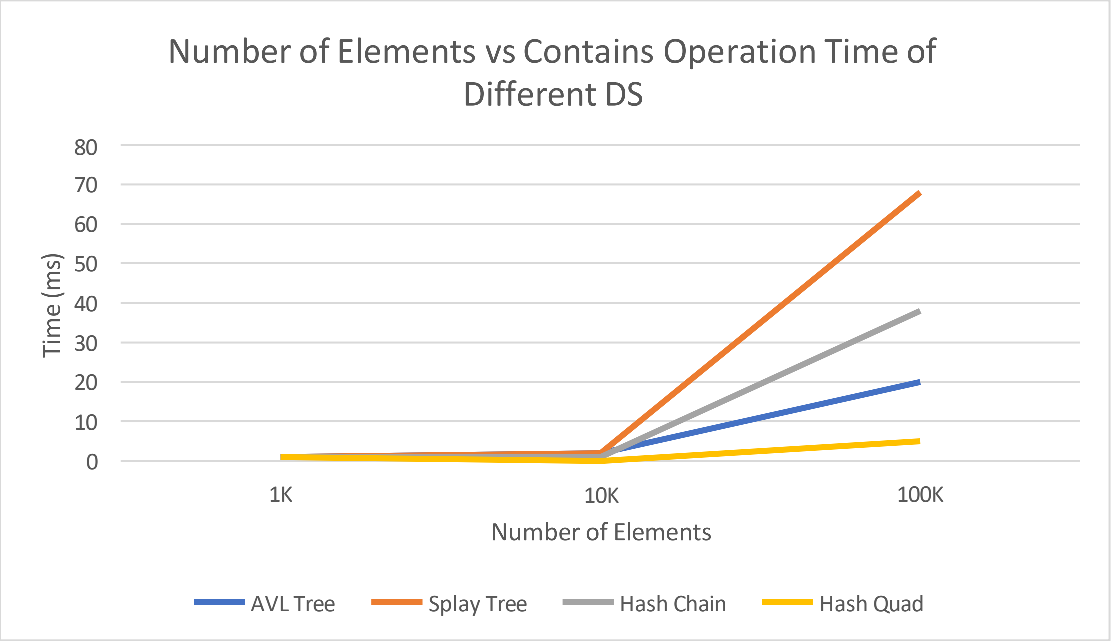


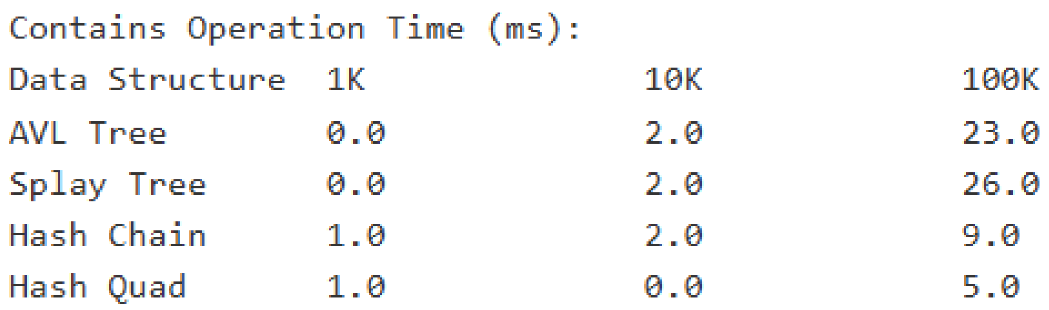
```
Contains Operation Time (ms):
Data Structure   1K              10K             100K
AVL Tree         1.0             2.0             58.0
Splay Tree       0.0             2.0             25.0
Hash Chain       1.0             2.0             15.0
Hash Quad        0.0             1.0             6.0
```

Kareem Saleh

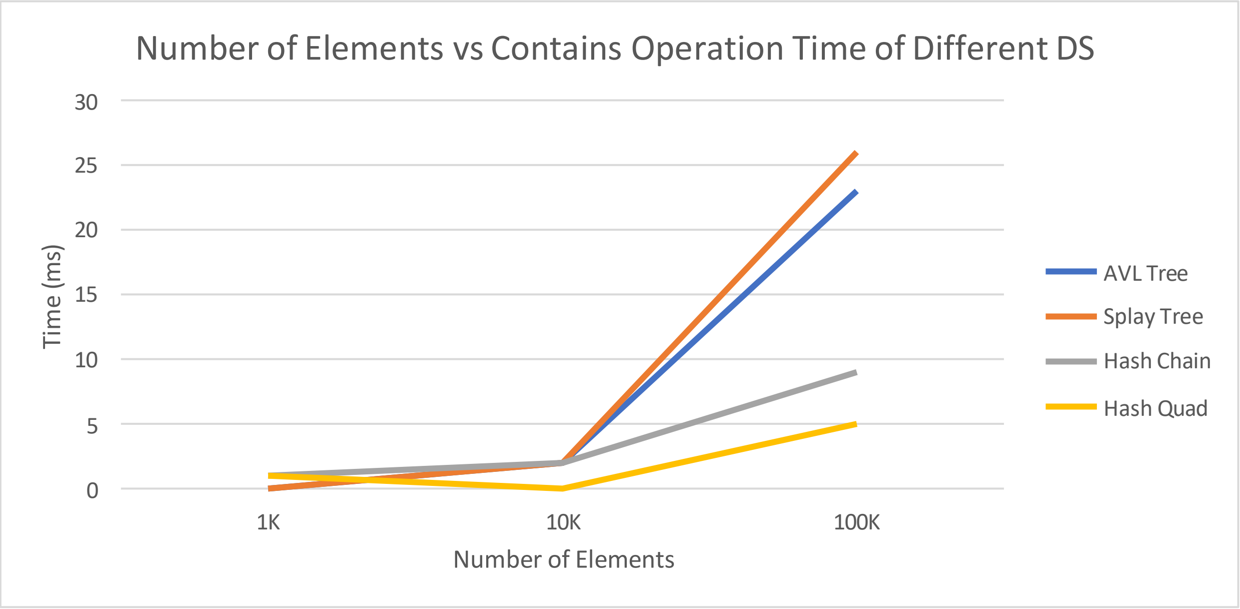



```
Contains Operation Time (ms):
Data Structure  1K              10K             100K
AVL Tree        1.0             2.0             20.0
Splay Tree      1.0             2.0             68.0
Hash Chain      1.0             1.0             38.0
Hash Quad       1.0             0.0             5.0
```





```
Contains Operation Time (ms):
Data Structure  1K              10K             100K
AVL Tree        0.0             2.0             23.0
Splay Tree      0.0             2.0             26.0
Hash Chain      1.0             2.0             9.0
Hash Quad       1.0             0.0             5.0
```



Actual:

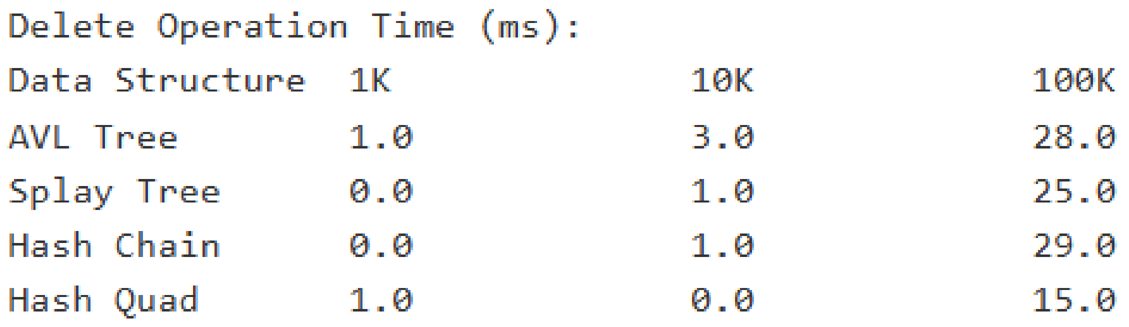
Based on the graphs and tables, the general trend indicates that Hash Table using Quadratic Probing has the fastest contains (search) amongst the data structures for the medium and large datasets. Although we have some zeros, this does not mean it was done in no time, it just means that it was less than 1 ms . For both the medium and large datasets, we expected Avl to have the fastest search operation. However, we see that the general trend points towards the efficiency of Hash Tables that use quadratic probing. This could have happened because of several reasons. Amongst those is the fact that the Quadratic probing leads to a well-distributed table with fewer collisions, making it optimized for high numbers of searches, which gets us to an average case of O(1) < O(log n). In addition, this sparse distribution leads to immediate access of the data without long probes.

Note:- 0 value here represents minimal or negligible time as opposed to no time at all.

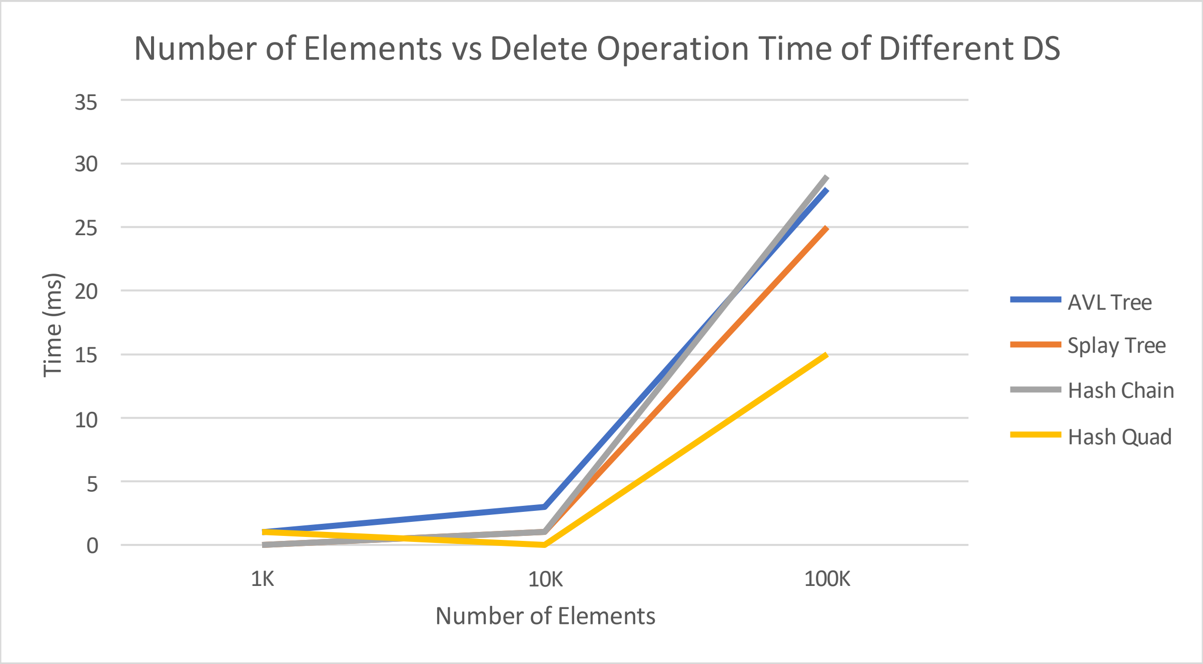
Delete Operations (Time):

Hypothesis:

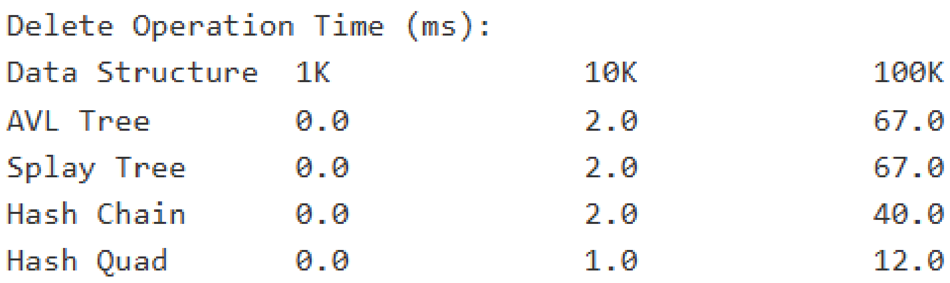
Based on the order of growth comparisons between functions in the data structure classes, we can attempt to predict the outcomes of the Delete operations times. Since the Avl Tree utilizes recursion splitting the tree in half every call, contains is O(log n). Even though splay tree also utilizes recursive calls splitting the tree in half every time, it also has a splay function. Since splay has O(n) complexity, the O(n) complexity dominates the O(log n) of the recursive calls making the whole Splay Tree's contains be O(n).Since other DS contains are worst case O(n), then the Avl Tree has the fastest contains.



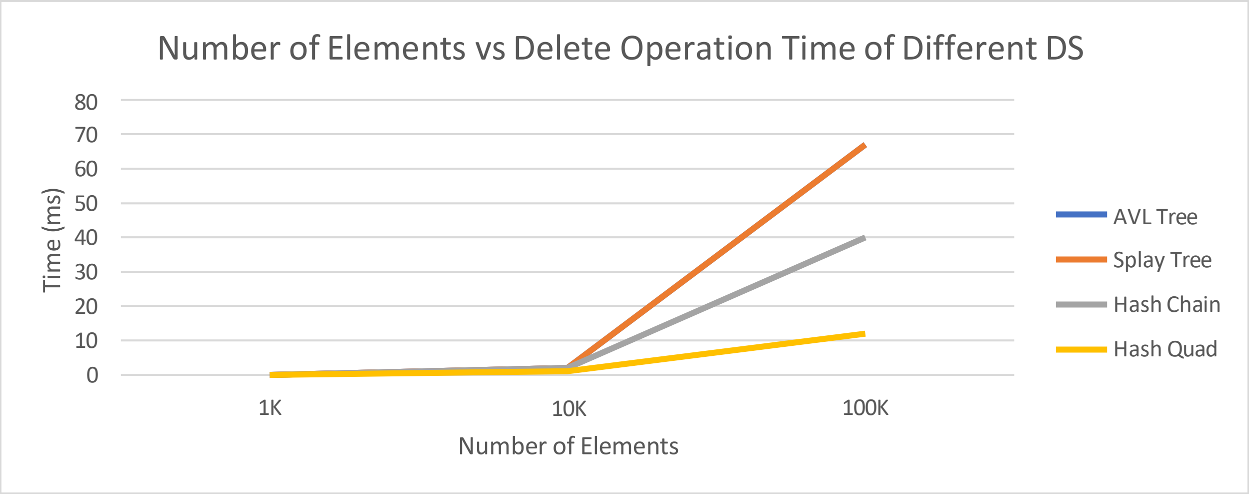
```
Delete Operation Time (ms):
Data Structure  1K          10K          100K
AVL Tree        1.0         3.0          28.0
Splay Tree      0.0         1.0          25.0
Hash Chain      0.0         1.0          29.0
Hash Quad       1.0         0.0          15.0
```

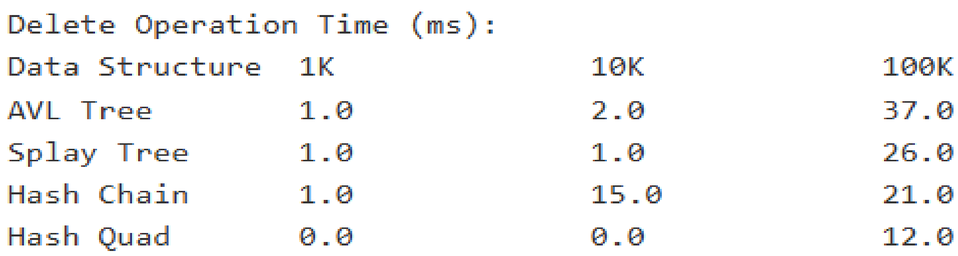


Kareem Saleh

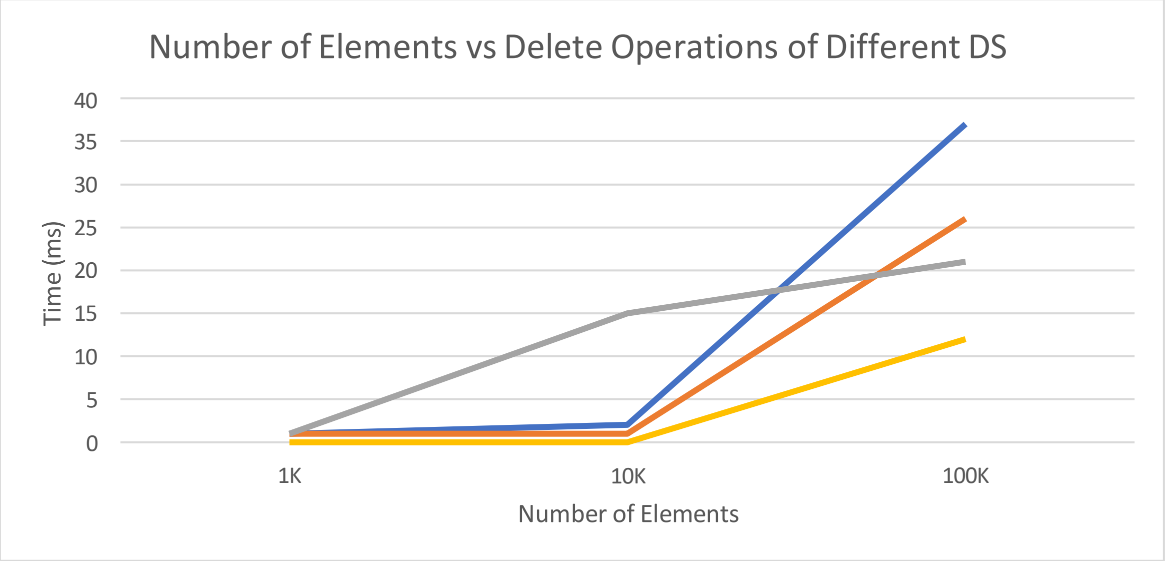


```
Delete Operation Time (ms):
Data Structure  1K              10K             100K
AVL Tree        0.0             2.0             67.0
Splay Tree      0.0             2.0             67.0
Hash Chain      0.0             2.0             40.0
Hash Quad       0.0             1.0             12.0
```





```
Delete Operation Time (ms):
Data Structure  1K              10K             100K
AVL Tree        1.0             2.0             37.0
Splay Tree      1.0             1.0             26.0
Hash Chain      1.0             15.0            21.0
Hash Quad       0.0             0.0             12.0
```



Actual:

Based on the graphs and tables, the general trend indicates that Hash Table using Quadratic Probing has the fastest delete amongst the data structures for the medium and large datasets. Although we have some zeros, this does not mean it was done in no time, it just means that it was less than 1 ms. We expected Avl to have the fastest delete operation. However, we see that the general trend points towards the efficiency of Hash Tables that use quadratic probing. This could have happened because of several reasons. Amongst those is the fact that the Quadratic probing leads to a well-distributed table with fewer collisions, making it optimized for high numbers of deletes, which gets us to an average case of O(1) < O(log n). In addition, this sparse distribution leads to immediate access of the data to delete without long probes.

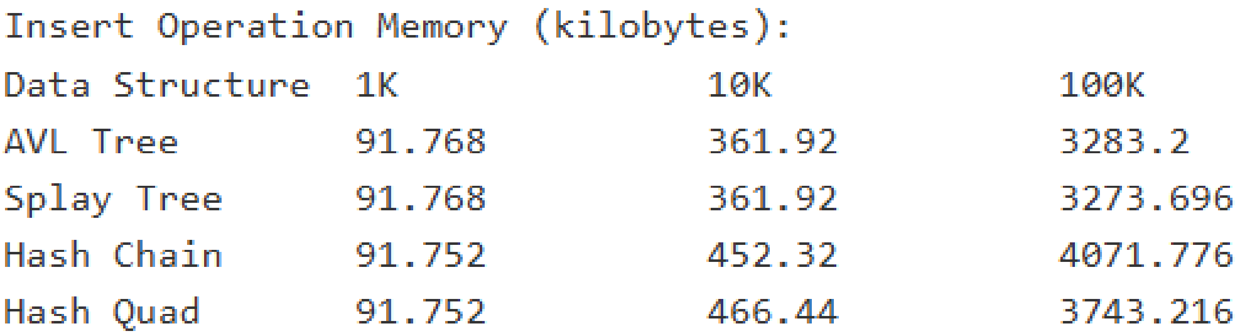
Note:- 0 value here represents minimal or negligible time as opposed to no time at all.

Memory Test Cases:-

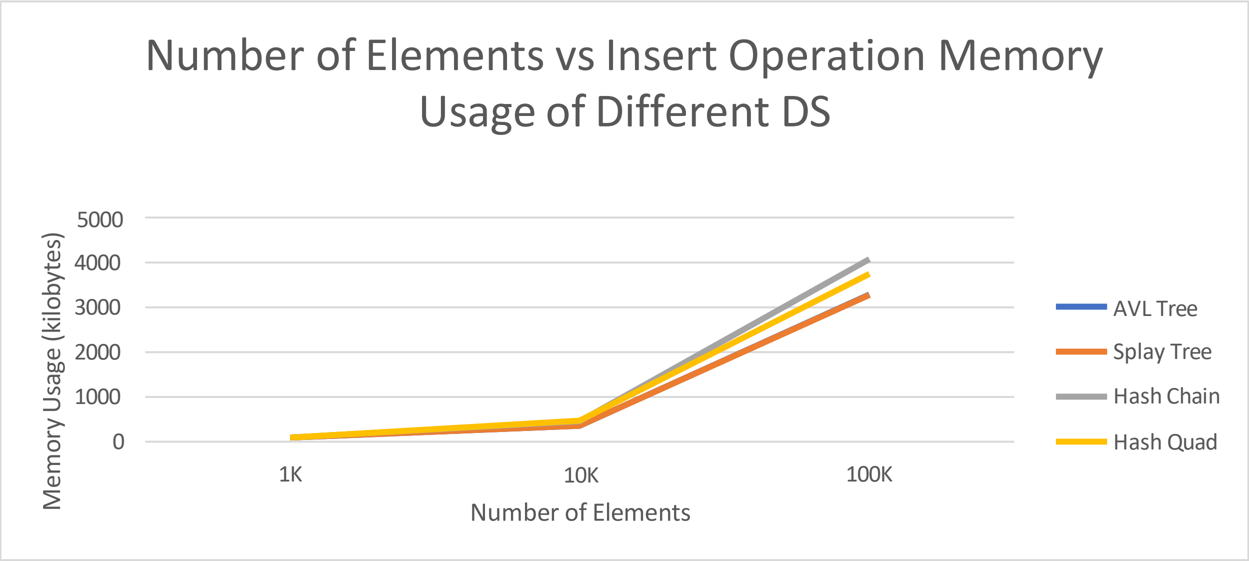
Insert Operations (Memory Usage):

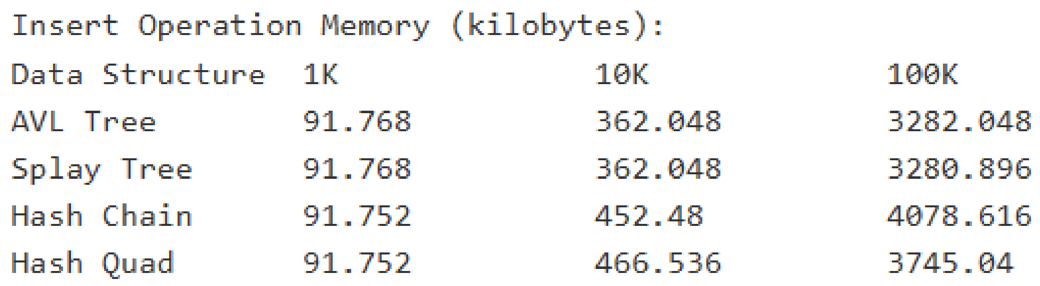
Hypothesis:

If we examine the behavior of different functions, we can try to predict and rank the efficiency of data structure operations in memory usage. By thinking about avl and splay trees, their inserts require creating a new node every single time. However, this is less memory than the Hash Table using Quadratic probing since it involves resizing of the array. Since we have a lot of elements, we will have to do multiple resizes, taking up the most memory out of all the DS. Since Hash Table using Chaining only utilizes linked list to create chains of elements as a solution to collisions, it does not employ node creation or resizing, making it the most memory efficient data structure of the 4, since a linked list and an iterator will not take as much memory as node creation or resizing.

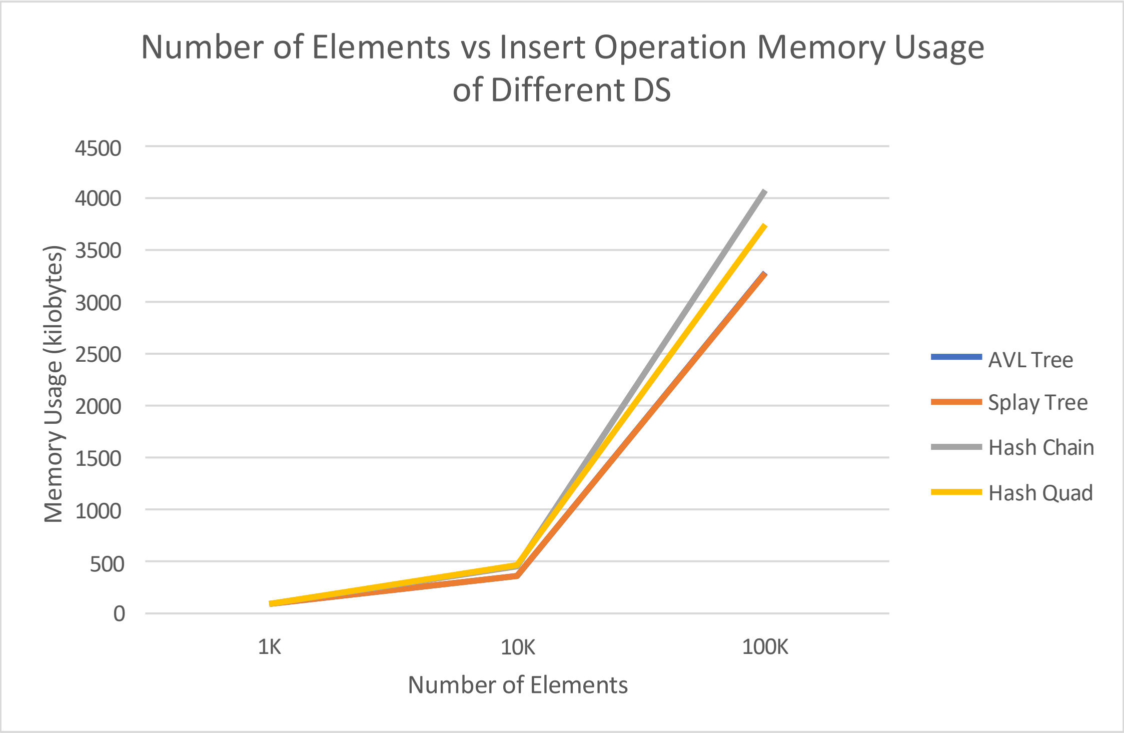


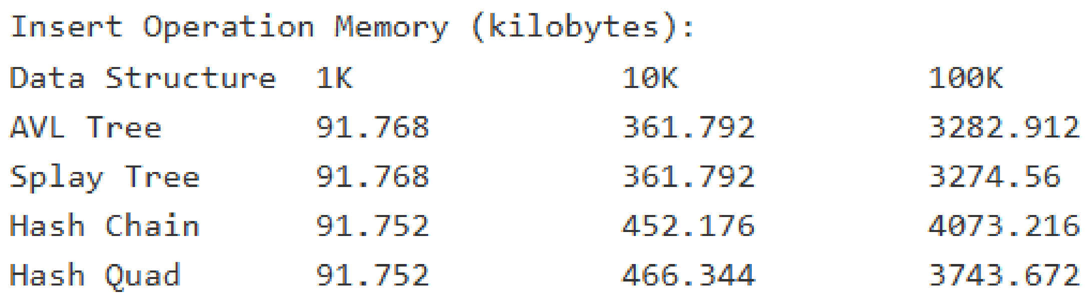
```
Insert Operation Memory (kilobytes):
Data Structure  1K              10K             100K
AVL Tree        91.768          361.92          3283.2
Splay Tree      91.768          361.92          3273.696
Hash Chain      91.752          452.32          4071.776
Hash Quad       91.752          466.44          3743.216
```



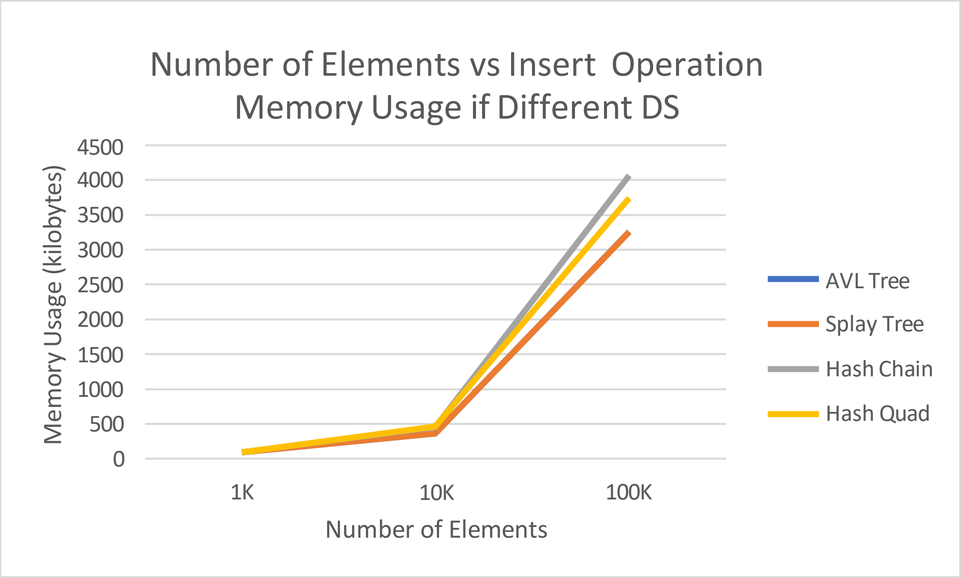


```
Insert Operation Memory (kilobytes):
Data Structure  1K              10K             100K
AVL Tree        91.768          362.048         3282.048
Splay Tree      91.768          362.048         3280.896
Hash Chain      91.752          452.48          4078.616
Hash Quad       91.752          466.536         3745.04
```





```
Insert Operation Memory (kilobytes):
Data Structure  1K              10K             100K
AVL Tree        91.768          361.792         3282.912
Splay Tree      91.768          361.792         3274.56
Hash Chain      91.752          452.176         4073.216
Hash Quad       91.752          466.344         3743.672
```



Actual:

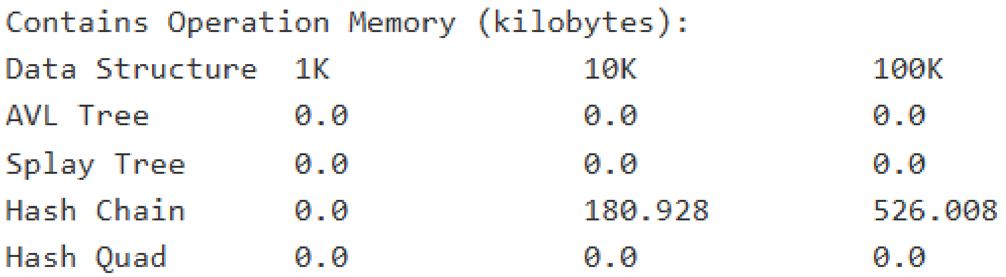
From the graphs and tables, we see that the general trend confirms that our hypothesis was accurate. Hashing Tables using chaining is the most memory efficient data structure of the 4, since a linked list and an iterator will not take as much memory as node creation or resizing. Additionally, avl and splay trees utilize similar memory since they both employ node creation. Finally, hashing using quadratic probing involves resizing of the array. Since we have a lot of elements, we will have to do multiple resizes, taking up the most memory out of all the DS.

Note:- 0 value here represents minimal or negligible memory usage as opposed to no memory at all.

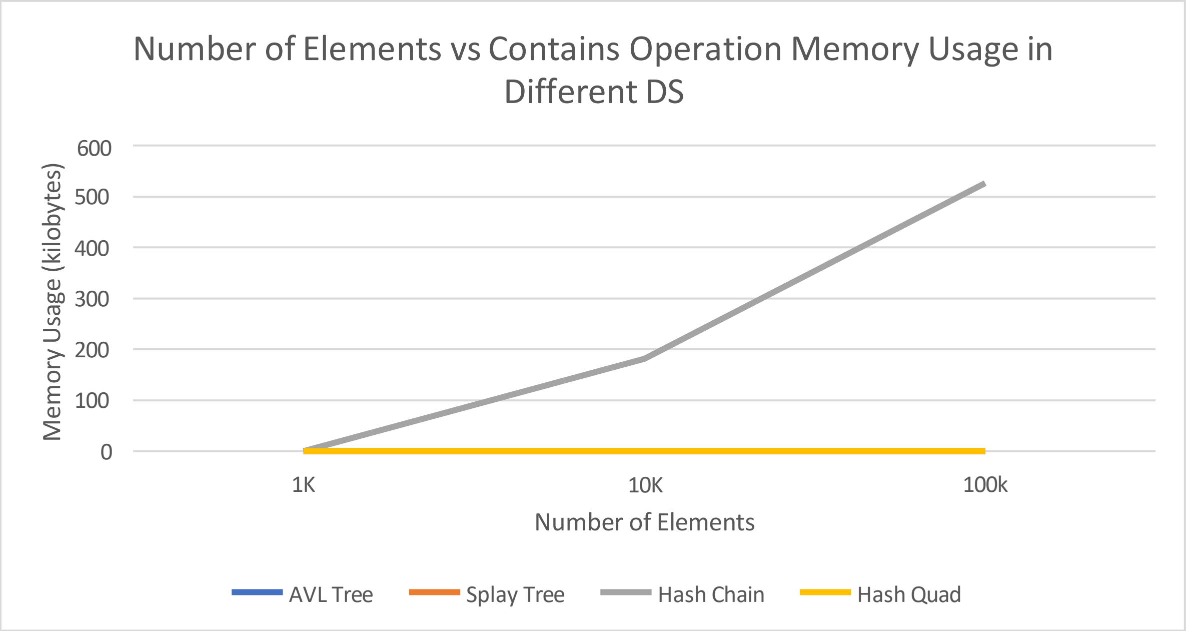
Contains Operations (Memory Usage):

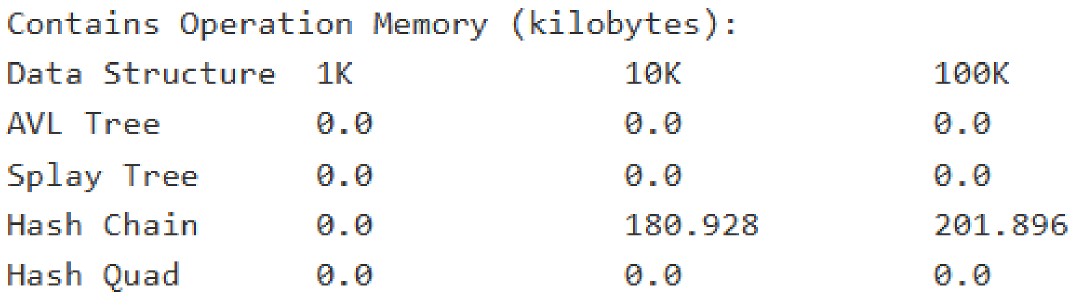
Hypothesis:

If we examine the behavior of different functions, we can try to predict and rank the efficiency of data structure operations in memory usage. By thinking about avl and splay trees, their searches only involve going down the tree to find the element, using negligible extra space. As for Hashing Tables involving quadratic probing, we only need the hashing function and probes to search for the element, which require minimal or negligible extra memory usage. Finally, Hash tables employing chaining use iterators to crawl all through the list to delete an element, which takes up some memory.

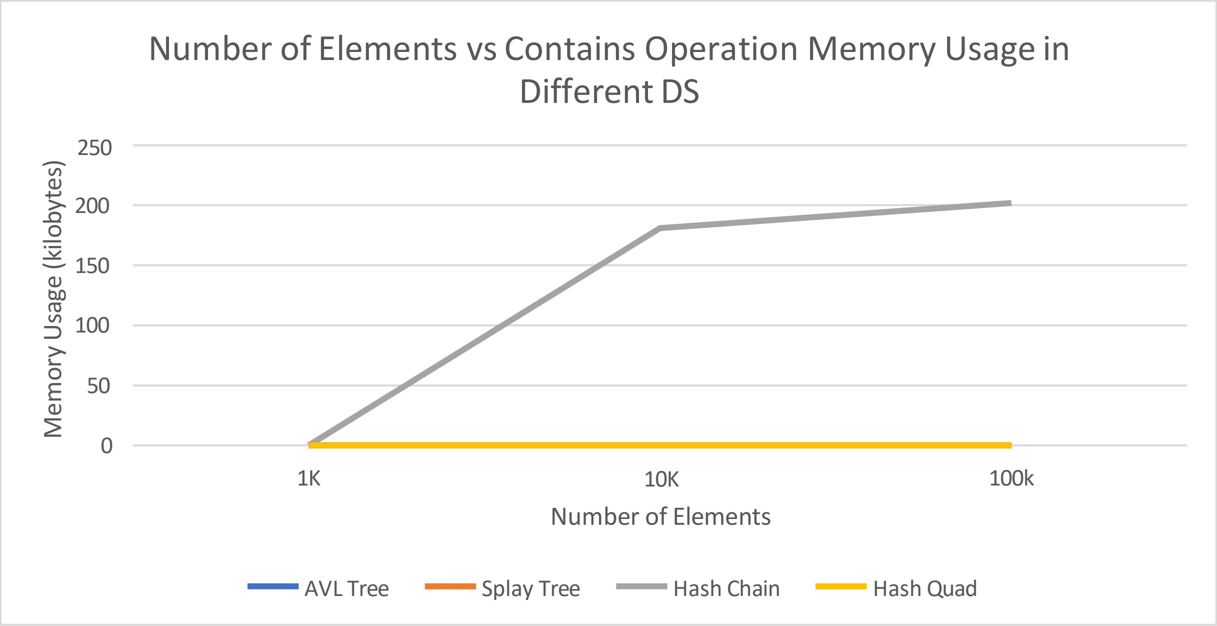


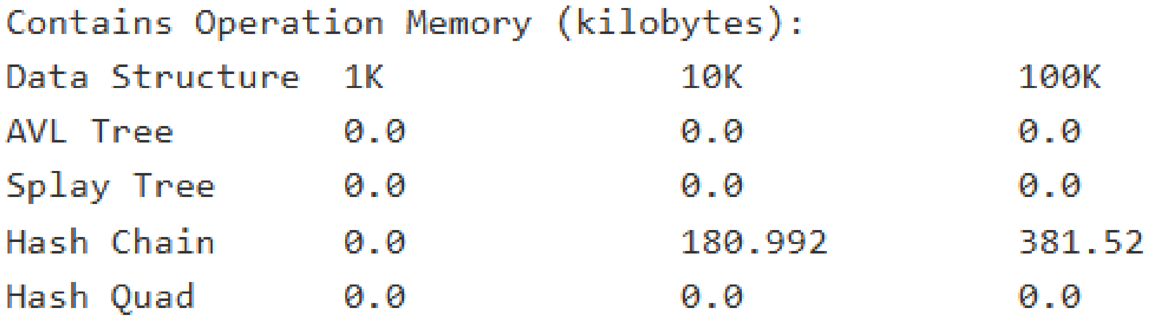
```
Contains Operation Memory (kilobytes):
Data Structure  1K              10K             100K
AVL Tree        0.0             0.0             0.0
Splay Tree      0.0             0.0             0.0
Hash Chain      0.0             180.928         526.008
Hash Quad       0.0             0.0             0.0
```



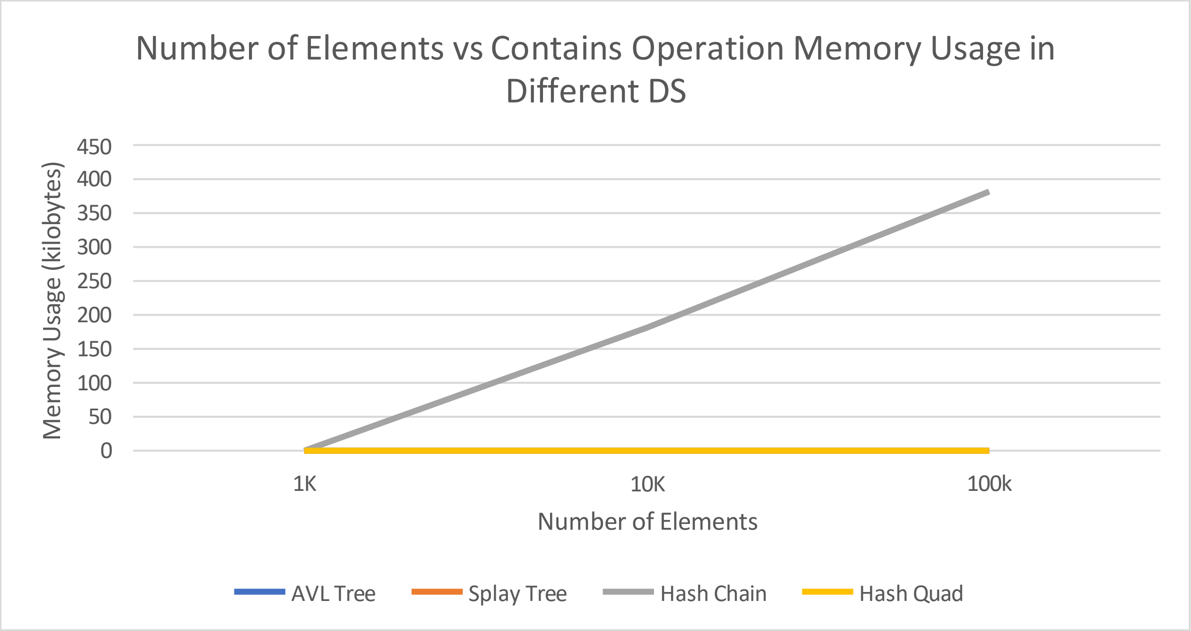


```
Contains Operation Memory (kilobytes):
Data Structure  1K              10K             100K
AVL Tree        0.0             0.0             0.0
Splay Tree      0.0             0.0             0.0
Hash Chain      0.0             180.928         201.896
Hash Quad       0.0             0.0             0.0
```





```
Contains Operation Memory (kilobytes):
Data Structure  1K              10K             100K
AVL Tree        0.0             0.0             0.0
Splay Tree      0.0             0.0             0.0
Hash Chain      0.0             180.992         381.52
Hash Quad       0.0             0.0             0.0
```



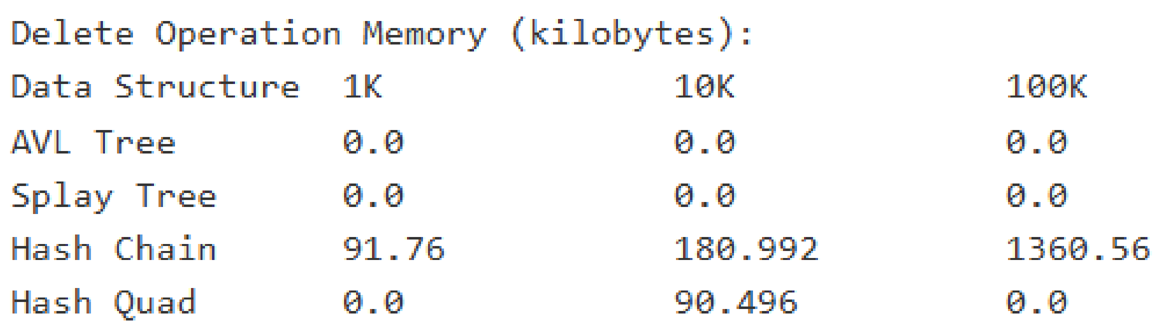
Actual:

From the graphs and tables, we see that the general trend confirms that our hypothesis was mostly accurate. Hash Tables using chaining is the least memory efficient data structure of the 4, since an iterator takes some memory. On the contrary, avl and splay trees as well as Hash Tables using quadratic probing utilize negligible memory since they do not require new creations or iterators. 0 value here represents minimal or negligible memory usage as opposed to no memory at all. However, I noticed some unexpected value in the 10K elements deletion for Hash tables containing quadratic probing which could be a result of wasted memory after resizes done by the insert.

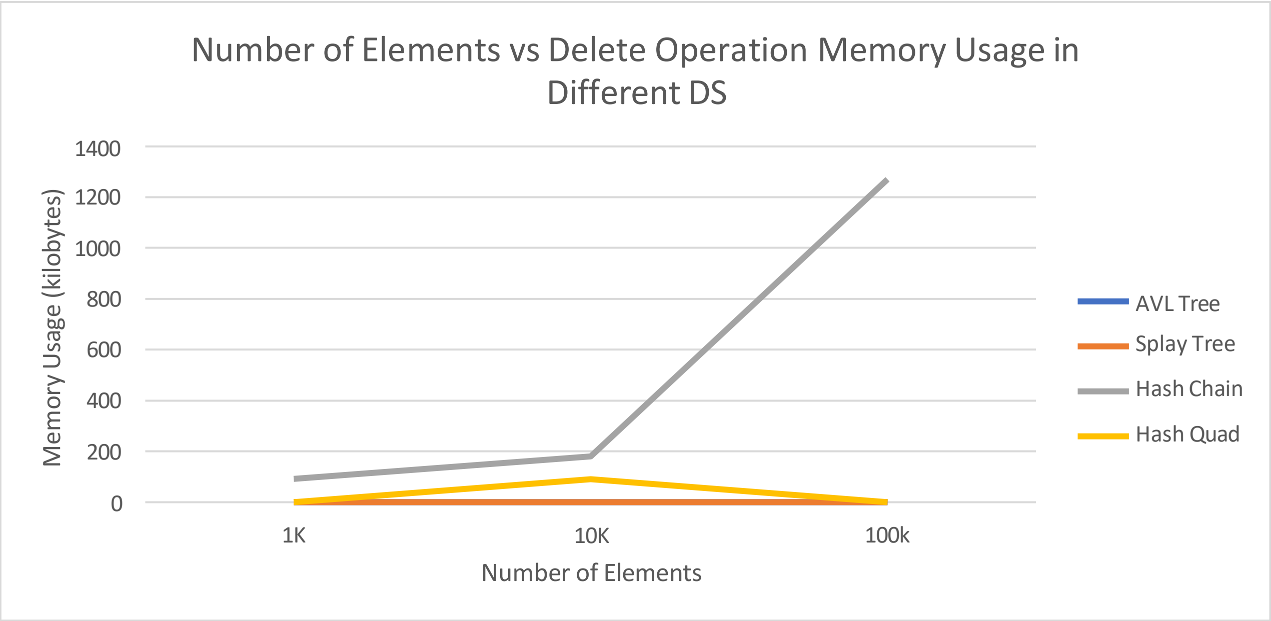
Delete Operations (Memory Usage):

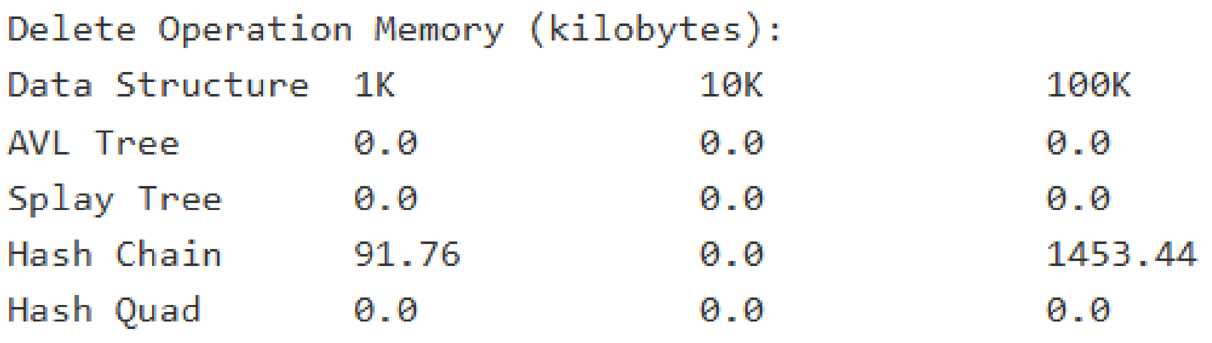
Hypothesis:

If we examine the behavior of different functions, we can try to predict and rank the efficiency of data structure operations in memory usage. By thinking about avl and splay trees, their deletes only involve going down the tree to find then delete the element, using negligible extra space. As for Hashing Tables involving quadratic probing, we only need the hashing function and probes to search for the element, which require minimal or negligible extra memory usage. Finally, Hash tables employing chaining use iterators to crawl all through the list to delete an element, which takes up some memory.

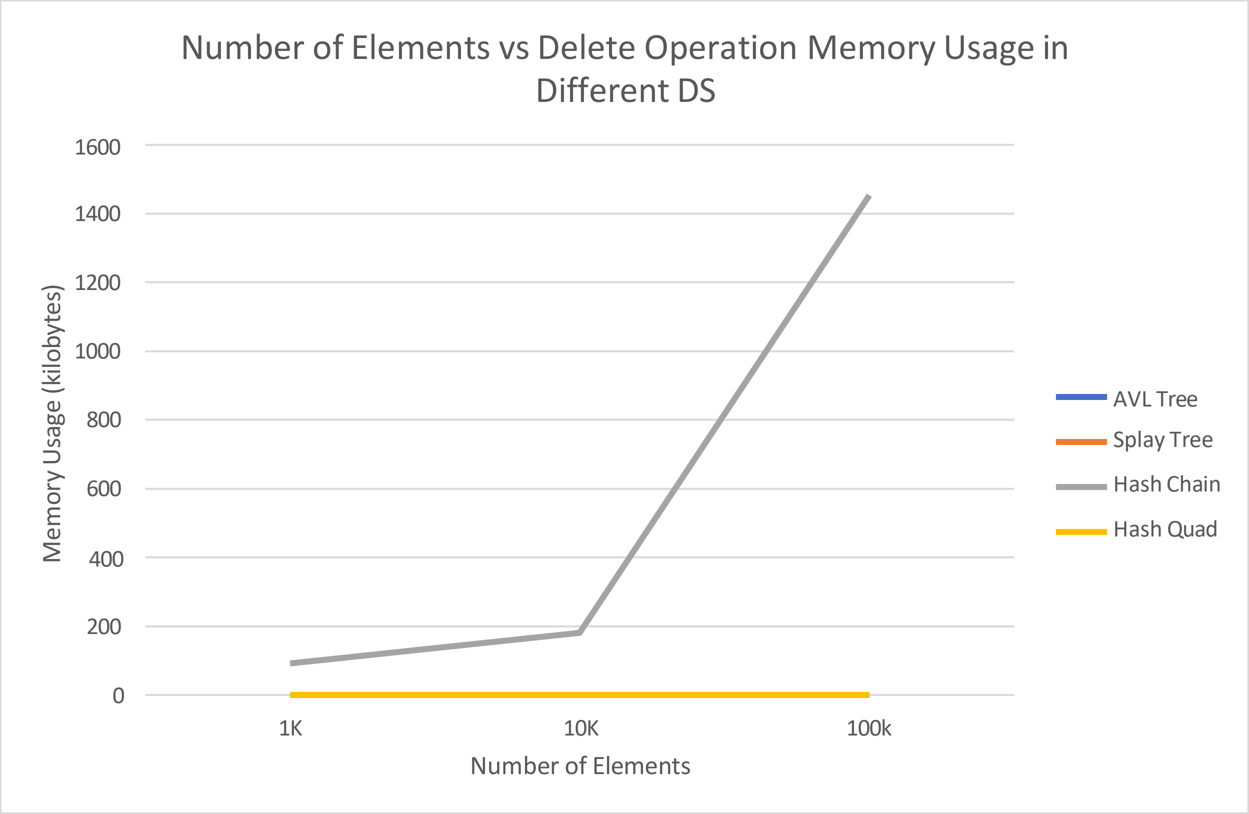


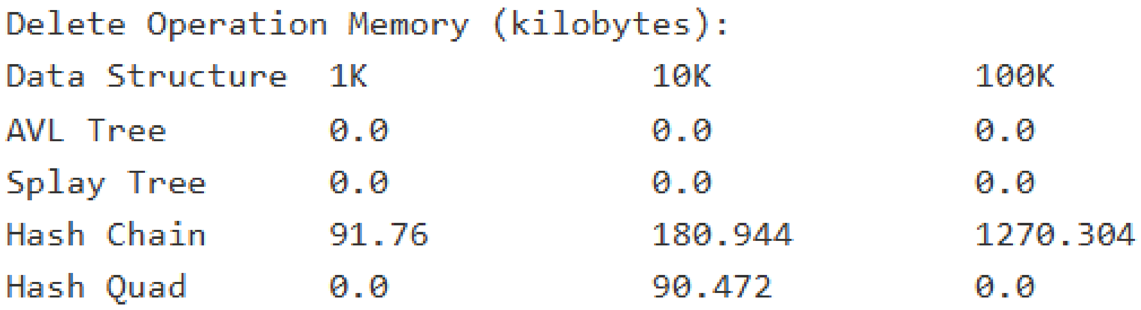
```
Delete Operation Memory (kilobytes):
Data Structure  1K              10K             100K
AVL Tree        0.0             0.0             0.0
Splay Tree      0.0             0.0             0.0
Hash Chain      91.76           180.992         1360.56
Hash Quad       0.0             90.496          0.0
```



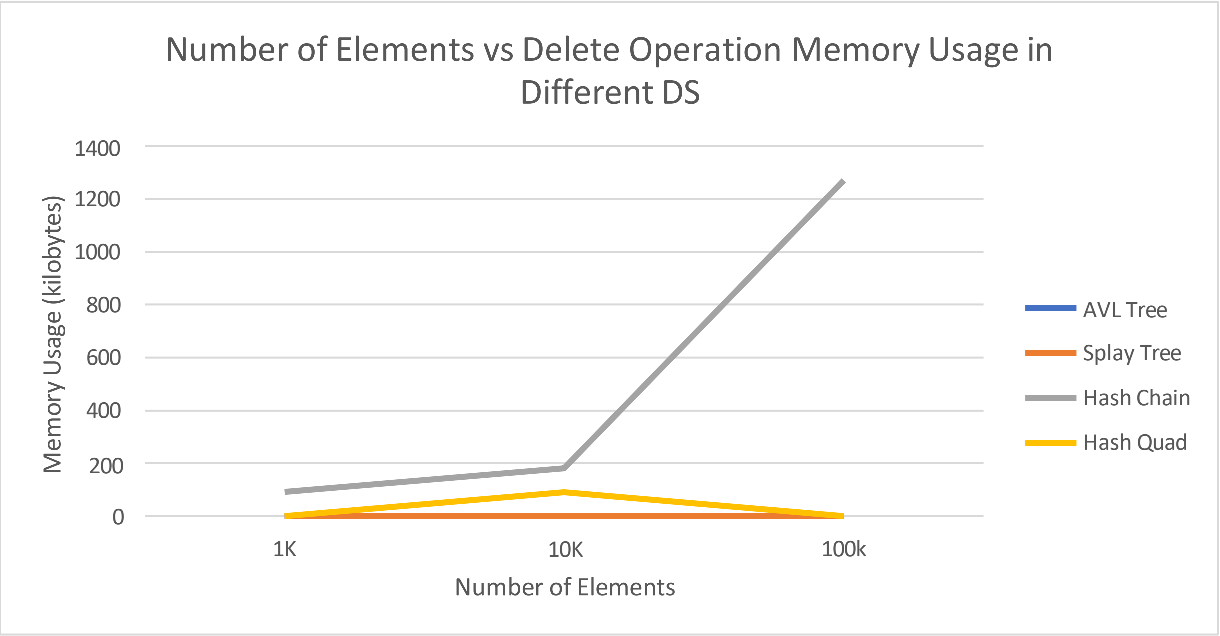


```
Delete Operation Memory (kilobytes):
Data Structure  1K              10K             100K
AVL Tree        0.0             0.0             0.0
Splay Tree      0.0             0.0             0.0
Hash Chain      91.76           0.0             1453.44
Hash Quad       0.0             0.0             0.0
```





```
Delete Operation Memory (kilobytes):
Data Structure  1K              10K             100K
AVL Tree        0.0             0.0             0.0
Splay Tree      0.0             0.0             0.0
Hash Chain      91.76           180.944         1270.304
Hash Quad       0.0             90.472          0.0
```



Actual:

From the graphs and tables, we see that the general trend confirms that our hypothesis was mostly accurate. Hash Tables using chaining is the least memory efficient data structure of the 4, since an iterator takes some memory. On the contrary, avl and splay trees as well as Hash Tables using quadratic probing utilize negligible memory since they do not require new creations or iterators. 0 value here represents minimal or negligible memory usage as opposed to no memory at all. However, I noticed some unexpected value in the 10K elements deletion for Hash tables containing quadratic probing which could be a result of wasted memory after resizes done by the insert.

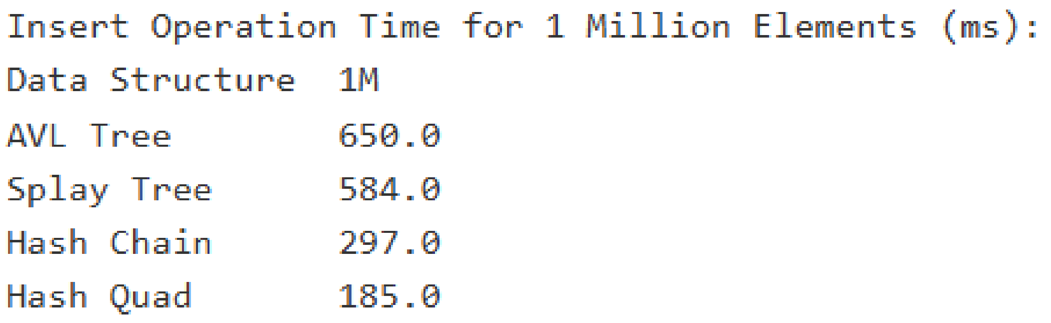
Exploration:

After examining time and memory usage of different data structures through code, we came up with some hypotheses, conducted trials and produced graphs and tables with the results. Although some results complied with our hypotheses, while others introduced us to new ideas about the data structure performances for small, medium and large data sets. But since Big O refers to the performance of data structure operations at really big values, I thought about conducting further trials of the insert operations across the different data structures when dealing with 1 Million random elements, to examine time performance.

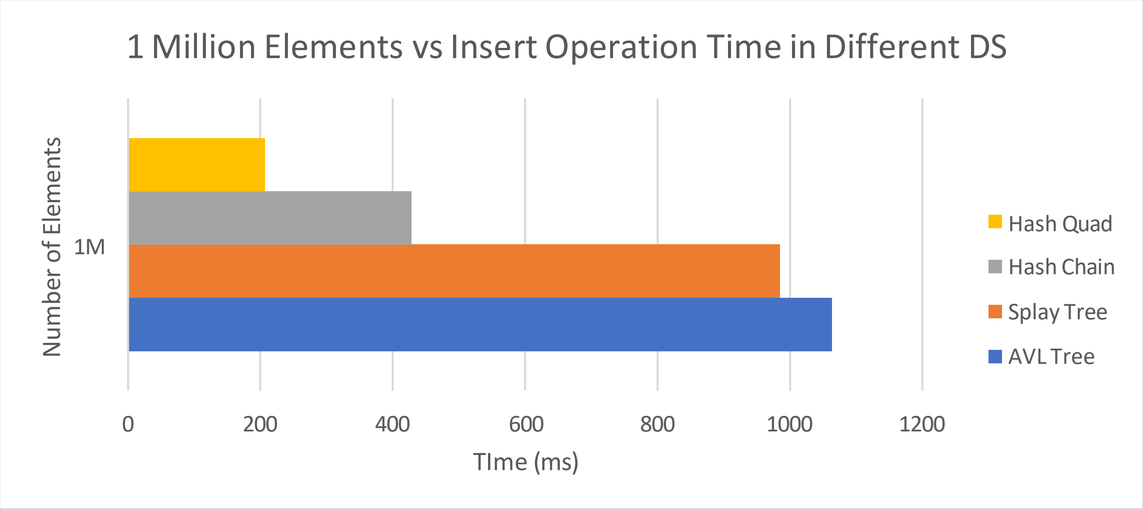
Insert Operation for 1 Million Elements (Time):

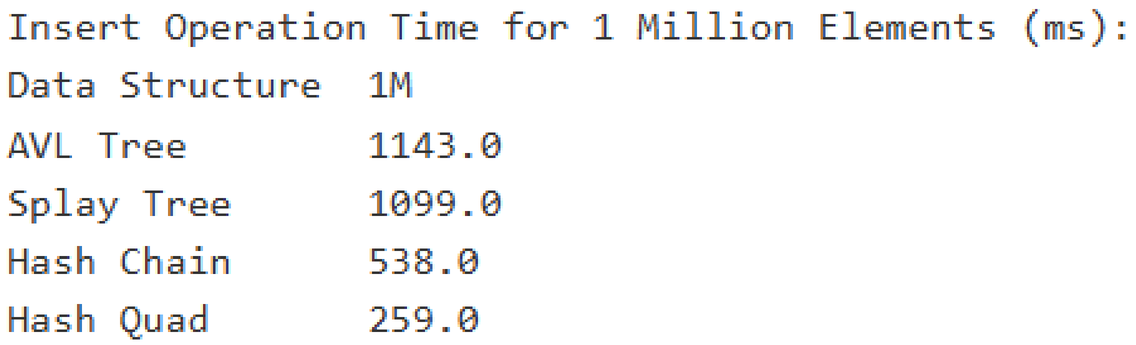
Hypothesis:

Based on the order of growth comparisons between functions in the data structure classes, we can attempt to predict the outcomes of the Insert operations times. Since the Hash Table (Chaining) utilizes the linked list insert which is O(1) and a hash function which is O(1), then it should be O(1). Since other DS inserts are O(log n) or O(n), then the Hash Table using Chaining has the fastest insert.

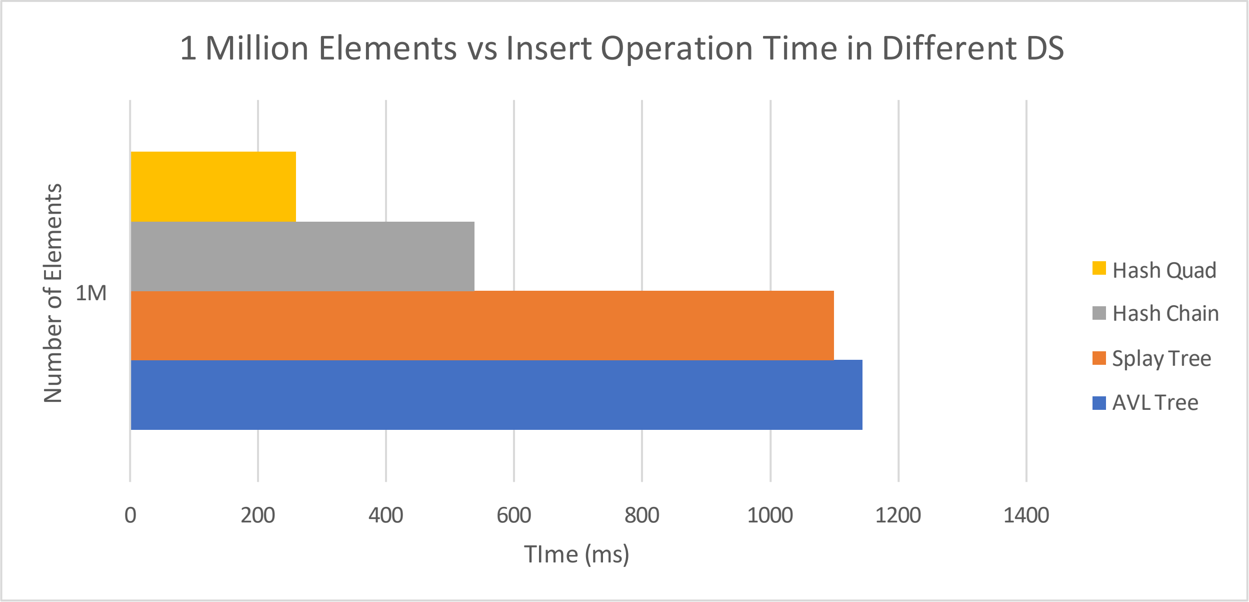


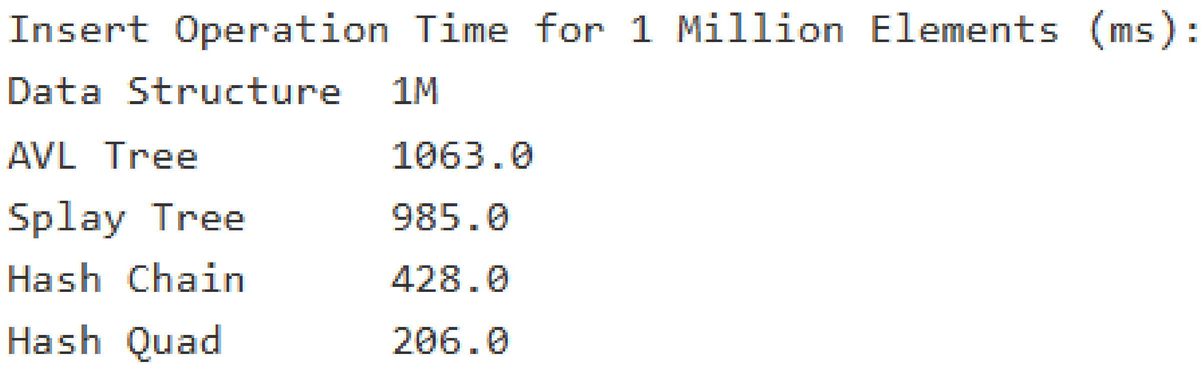
```
Insert Operation Time for 1 Million Elements (ms):
Data Structure  1M
AVL Tree        650.0
Splay Tree      584.0
Hash Chain      297.0
Hash Quad       185.0
```



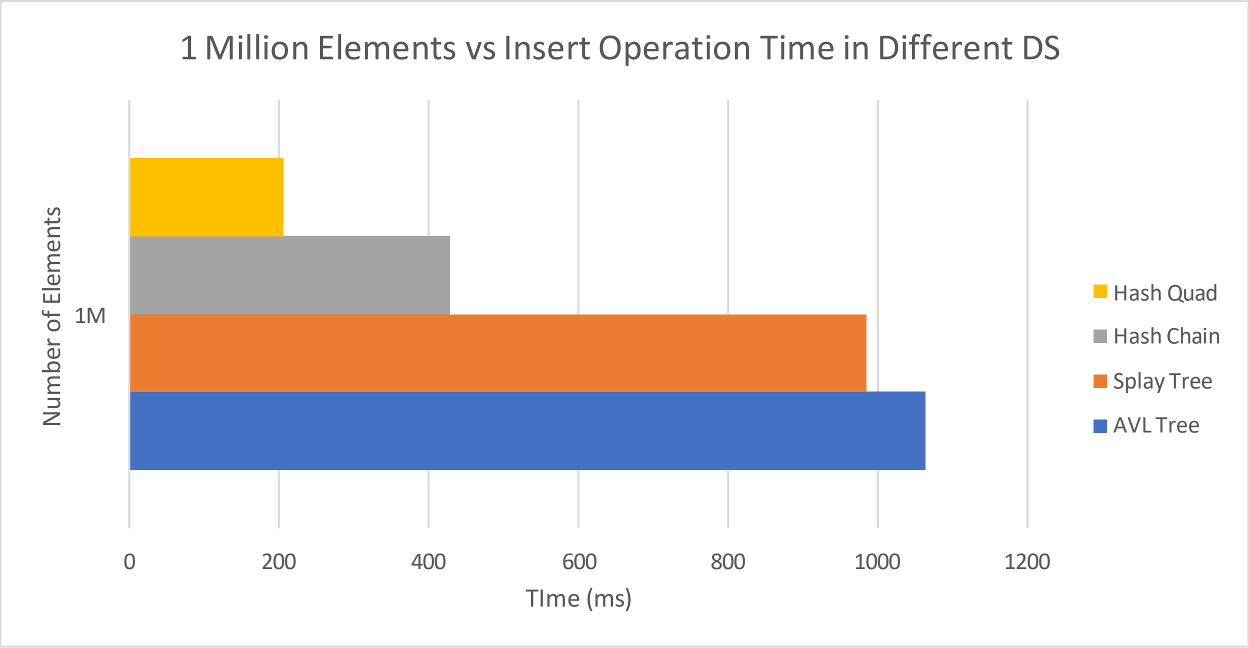


```
Insert Operation Time for 1 Million Elements (ms):
Data Structure  1M
AVL Tree        1143.0
Splay Tree      1099.0
Hash Chain      538.0
Hash Quad       259.0
```





```
Insert Operation Time for 1 Million Elements (ms):
Data Structure  1M
AVL Tree        1063.0
Splay Tree      985.0
Hash Chain      428.0
Hash Quad       206.0
```



Actual:

Based on the graphs and tables, the general trend gives us a notable change in ranking DS efficiencies. Previously, we found out that the general trend indicates that Hash Table using Chaining has the fastest insert amongst the data structures for the small, medium and large datasets.

To recap, the speeds of inserts for splay and avl trees were similar, supporting the fact that they are both O(log n). The Hash Table (Quadratic Probing) insert is slower and it is justified through the O(n) complexity (caused by the resizethat it had (because O(log n) < O(n)). As for the 100K elements, Hash Table employing Chaining is still the fastest but, the Hash Table using Quadratic Probing has insert faster than the avl and splay trees' inserts even though they are O(log n). This could have happened because the Quadratic probing lead to a well-distributed table with fewer collisions due to spars population of the table.

Even though I expected the same rankings in 1 Million elements, Hash Tables using Quadratic probing had faster inserts than the Hash Table using Chaining. Despite seeming unusual at first, revisiting the concepts of Chaining and quadratic probing made the difference. For 1 Million elements, the chains have become longer than before, but this didn't reduce the advantage that the Hash Tables had by using chaining because it is O(1) and we insert at the end. What made the difference was an advantage given to Hash Tables with quadratic probing. This is due to the fact that quadratic probes sparsely distribute the data more as the table capacity increases. So, when the number of elements gets bigger, elements are widely distributed and we have less probes and direct assignment to indices using the Hash Function. Thus, this explains the shift in rankings at 1 Million elements.

Finally, AVL and Splay trees are still slower than hash tables. However, AVL is slower since it calls the balance function for every insert, which takes more time as opposed to the splay.

Kareem Saleh