Assignment 1: Multisets - S3656436, S3656260

There are 3 different scenarios that were generated,

1. Addition,
2. Removal
3. Searches

Each with 3 different set amount of data elements,

1. 100 elements
2. 1,000 elements
3. 10,000 elements

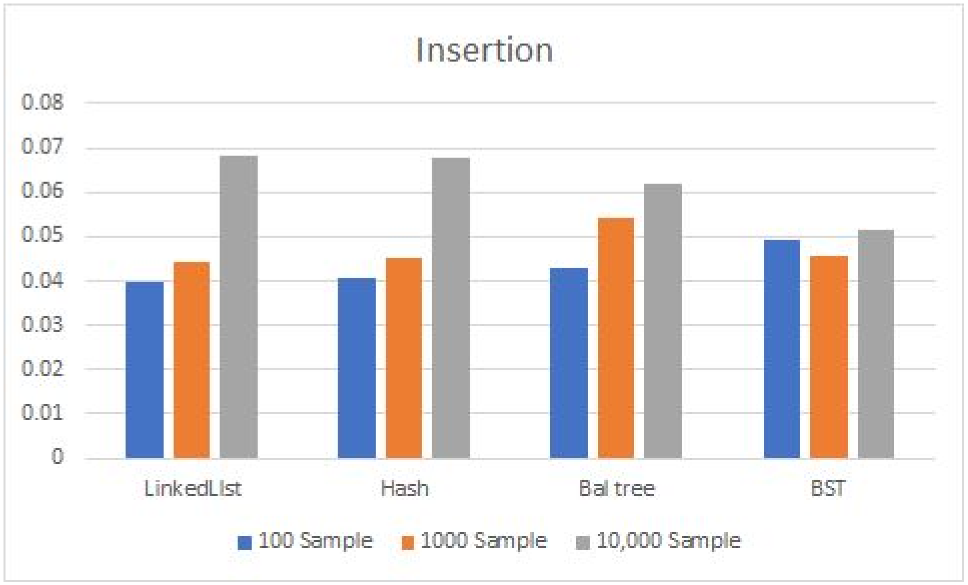
For our datasets, we decided upon 3 different ranges for our 3 different scenarios. For our first scenario, addition, we generated 3 lots of datasets with ranges up to 100 elements, 1,000 elements, and finally 10,000 elements. This was also followed for our other two scenarios, Removal and Searches, and was repeated for each Multiset, creating a total of 27 different test files in total.

We decided to have our ranges be factors of 10 as we believed larger jumps between each tests would greater highlight the time differences and leave us with a more accurate result, we also left our removals and additions at the same level for each scenarios, with searches remaining at a constant of 20 so we could see the differences the sizes of the multisets make in the times as the program is forced to continue down each multisets unique system searching for values.

Each scenario was independently created using a home-made python script (testScript.py*) that allowed the user to set their own values for each type of scenario (ie. how many additions) while also allowing for removes and removeAlls to also be created to further test multisets. This came with the downside of having random integers being called, but we believed that these would be small amounts of unused code and would not affect the overall samples if it was maintained throughout.

For timing, we used the regular python testing script but with a few modifications (line 60, 113, 117), placing a timer using the nano.time function, we time the script beginning at its first attempt and print that to console alongside our results and the scripts regular output, giving us a very accurate timing for each of our tests and allowing us to quickly test and retest depending on what is needed.

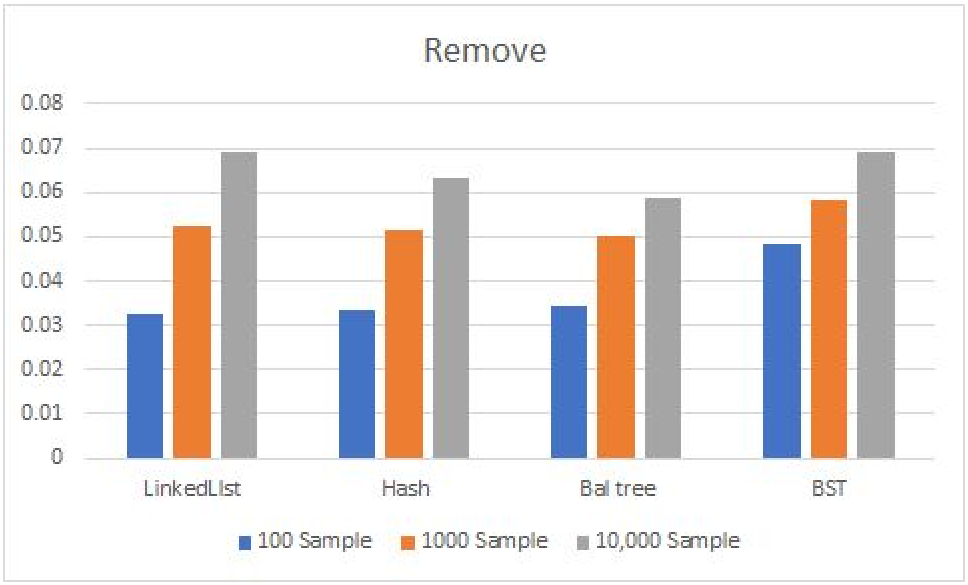
For our data structures, we began to notice some obvious advantages and disadvantages as we went through each test.



For our testing on insertion, we found that smaller sample sizes often were quicker when placed into a linked list or Hash, Balanced tree's only suffered a small increase, while contrasting this BST surprisingly saw a higher time than its own larger sample.

For medium sample sizes, Linked Lists and Hashes kept their small timing, while balanced tree suffered a large increased and BST decreased in time.

For larger samples, the previous two winner suffered heavy time loses, and our previously slower BST finally becoming the fastest as the sample size becomes large and progressive searching times increase. This was also present in the balanced tree, showing trees to have an advantage in the larger sets.



For our testing on removal, we found that again in smaller sizes, linked lists and hash maps were the obvious choice for speed on smaller sets. Contrasting this, BST's timings was the largest to reflect how long searching through trees with many leaves can take.

For our testing on searching, our results did not properly reflect an accurate result, and we were not able to use these tests.

In conclusion, our testing on each multiset showed results relating to each of the multsets advantages! As the data sets were smaller, the hash table algorithm performed best with the smallest time for each scenario. Hash table would be recommended for the smaller sets of data as the analysis shows. As the data grows, BST is the best solution, and hence is recommended for bigger data sets. Although this data does have a conclusion, there can be many errors in testing which could lead to false evaluation.