Sean Lee Jin Xiang

Report for Set ADT.

Array ADT:

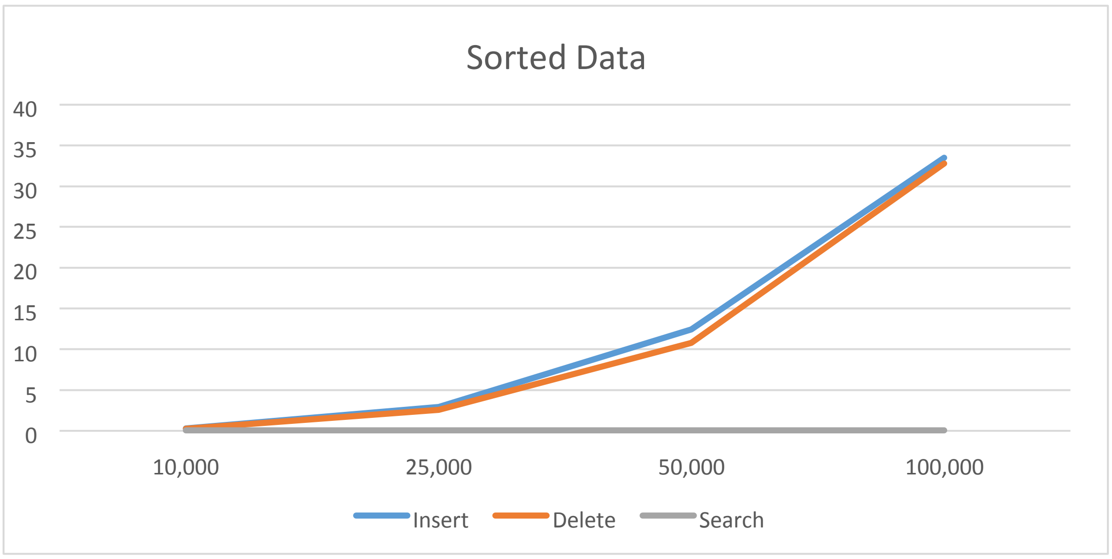
For the array ADT I have implemented a dynamic array structure which dynamically expands (increases) in size as the input data gets larger. If the maximum capacity is reached it will double (*2) in size of the previous array size.

The sorting algorithm used for the Insert Function for the Array ADT is a Binary Insertion sort, which finds the position in which the data has to be inserted using a Binary Search. This gives a $\lceil \log_2(n) \rceil$ comparisons in the worst case, ($O(n \log n)$) , but the entire algorithm still has an average running time of $O(n^2)$ because of the swaps needed to be performed on the array after each insertion. The best case is if the value to be inserted is at the end of the array and worse case would be if it was at the beginning because if it is at the start, ALL of the values have to be shifted to the right.

The same goes for the Delete, a Binary Search was performed to find the position of the data to be deleted then shifts the previous elements of the array forward. The running time of this depends on the location of the deleted data as well, if it is at the end of the array then no elements have to be shifted, and will return a logarithmic running time, which is the time required to search for the delete value, if it is at the start however then all the previous elements have to be shifted forward (to the left), which worst case will be a quadratic running time, $O(n^2)$.

The Search function uses the Binary Search to find the value. Which will always return a logarithmic running time.

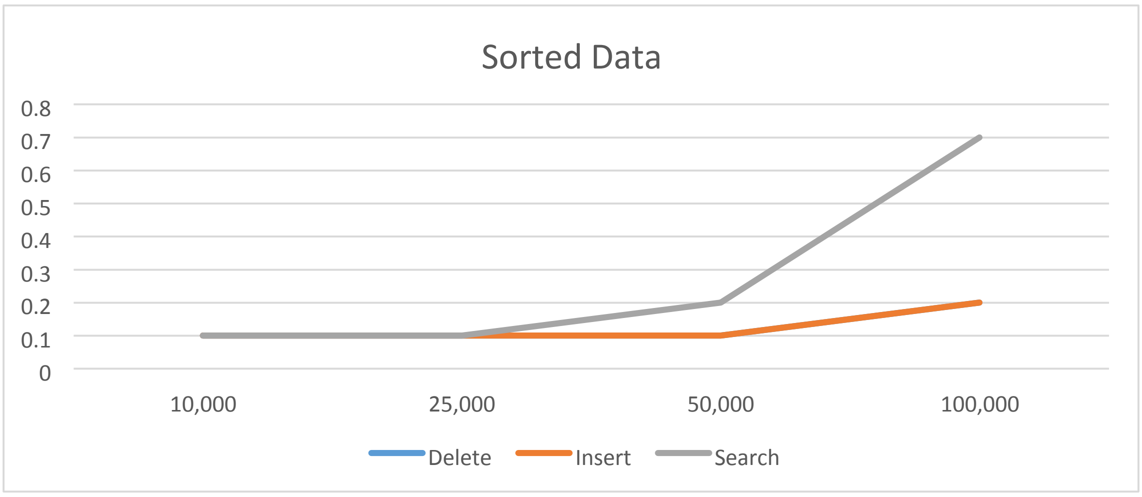
Below are some graphs that shows the performance of the Array ADT:



Binary Tree ADT:

For the binary tree ADT, it sorts the vales as it is entered into the tree, weather it is to the left (less then) or right (more then), which is done in the insert function. It has an average run time of *O* (log *n*). The idea of the binary tree is that data is searched by halving the problem each time. The functions for insert, delete and search finds the correct nodes recursively through the static functions I have created, in-order to have the correct structures to be able to traverse from node to node.

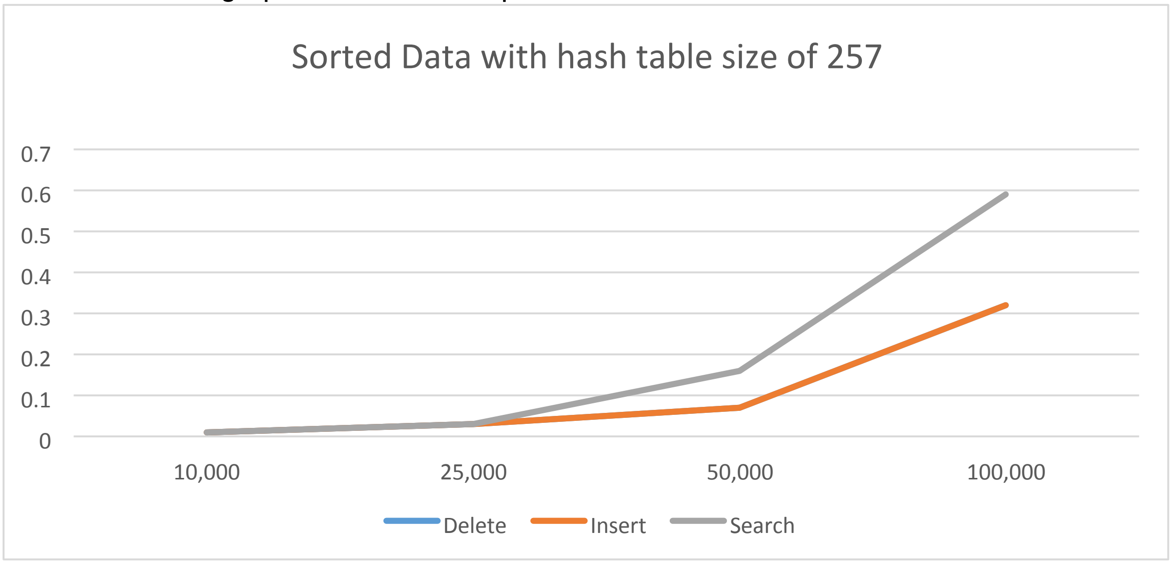
Below are some graphs that show the performance of the Binary Tree ADT:



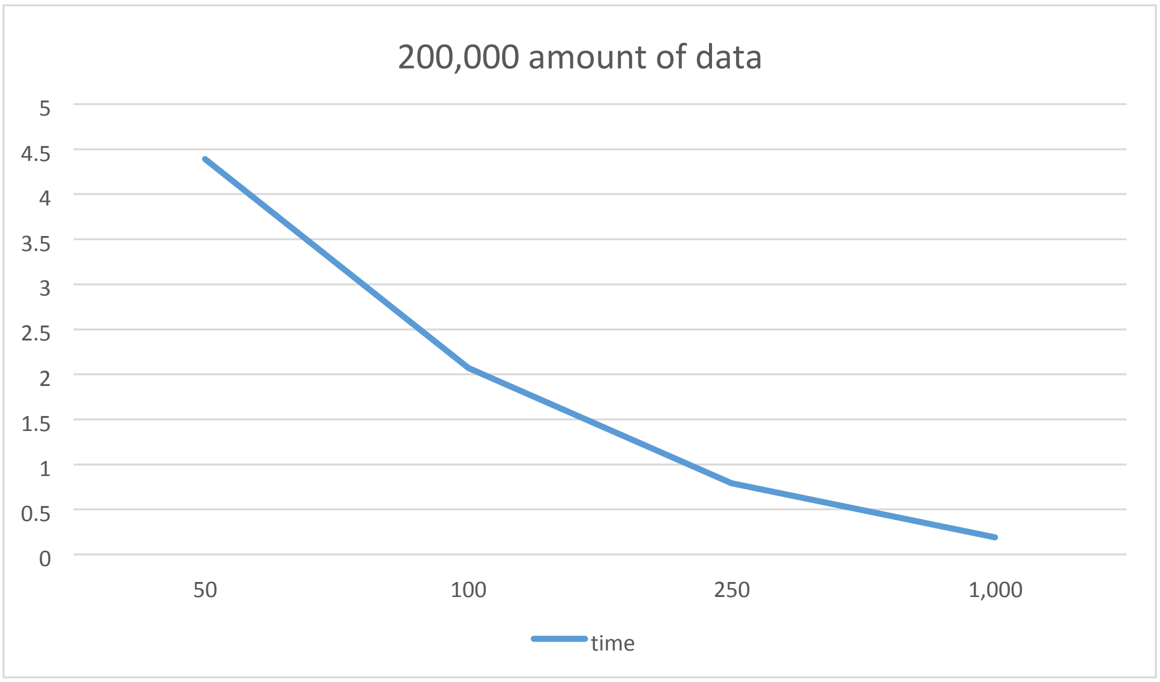
Hash Table ADT:

For the hash table ADT, it stores its values according to the hash key which is retrieved by finding the module of the value and the SIZE factor defined in the set_hash.c page. The value then determines the location of where the value will be stored in the array in the form of link lists. Hash tables are O(1) on average however it does have a worst case time complexity of O(n) which is the size of the entire array. This happens if too many elements were hashed into the same key, so it is very important to choose a good hash function. This can be done by choosing a prime number.

Below are some graphs that show the performance of the Hash Table ADT:



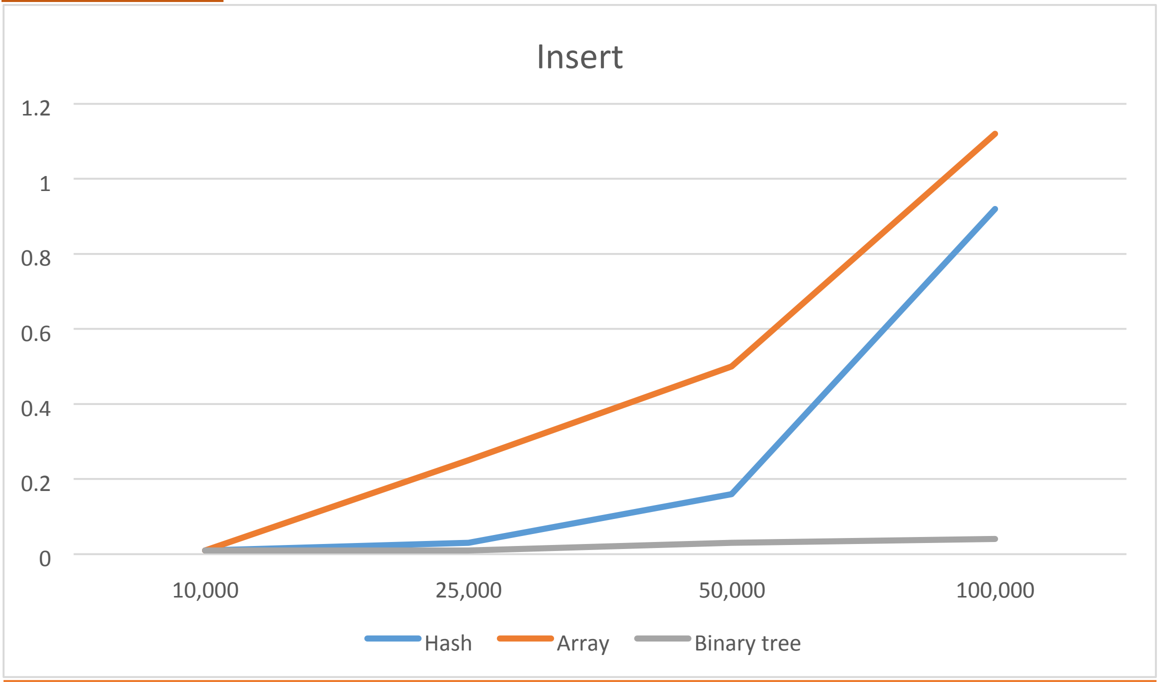
Below is a graph that shows the performance on a constant amount of data but varying hash table size:



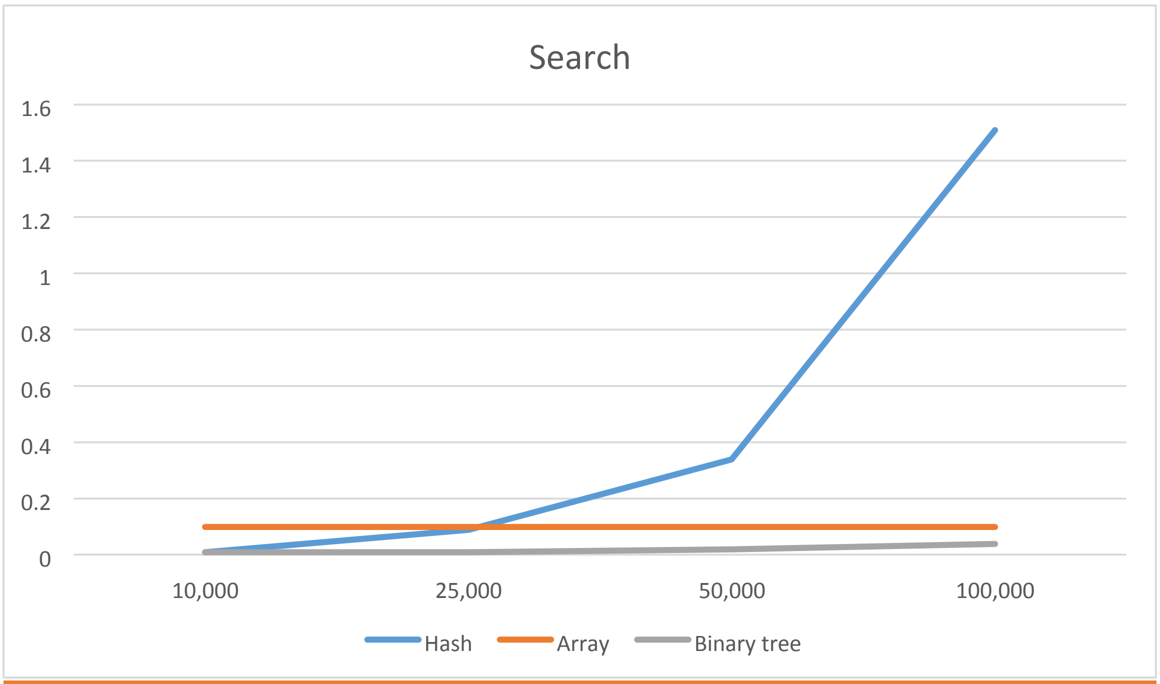
As you can see, the less elements hashed into the same key, the quicker the program runs.

Below are graphs that compare between sets:

Insert Function:



Search Function:


Search
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
10,000
25,000
50,000
100,000
Hash
Array
Binary tree