Week 11: Collections

Assignment 1:

This assignment required us to compare the operations efficiency, in addition to this we had to report the timings of the methods pertaining to eat data structure. We had to run the tests for around a million times, the purpose was to check the effective functioning of each function under that amount of data. Additionally, these observations will give us a proper comparisons regarding which data structure can be used based on the run time i.e. the speed of operations at which it performs them.

----------Collection :ArrayList----------

Operation : add , Collection :ArrayList, Time :1.2E-4 millisecond
Operation : collection equality , Collection :ArrayList, Time :0.0 millisecond
Operation : contains , Collection :ArrayList, Time :0.10502 millisecond
Operation : sizeOf , Collection :ArrayList, Time :3.0E-5 millisecond
Operation : remove , Collection :ArrayList, Time :0.00845 millisecond
Operation : set/change at location, Collection :ArrayList, Time :7.0E-5 millisecond
Operation : add at location , Collection :ArrayList, Time :0.019 millisecond
Operation : sort with custom comparator, Collection :ArrayList, Time :0.02700 seconds
Operation : sort, Collection :ArrayList, Time :0.03400 seconds
Operation : remove with index , Collection :ArrayList, Time :0.02836 millisecond

----------Collection :LinkedList----------

Operation : add , Collection :LinkedList, Time :5.0283597E-5 millisecond
Operation : collection equality , Collection :LinkedList, Time :0.0 millisecond
Operation : contains , Collection :LinkedList, Time :0.18864 millisecond
Operation : sizeOf , Collection :LinkedList, Time :0.0 millisecond
Operation : remove , Collection :LinkedList, Time :2.3E-4 millisecond
Operation : set/change at location, Collection :LinkedList, Time :0.04686 millisecond
Operation : add at location , Collection :LinkedList, Time :0.10707 millisecond
Operation : sort with custom comparator, Collection :LinkedList, Time :0.02400 seconds
Operation : sort, Collection :LinkedList, Time :0.10900 seconds
Operation : remove with index , Collection :LinkedList, Time :2.2E-4 millisecond

----------Collection :TreeSet----------

Operation : add , Collection :TreeSet, Time :2.800022E-4 millisecond
Operation : collection equality , Collection :TreeSet, Time :30.0 millisecond
Operation : contains , Collection :TreeSet, Time :8.4E-4 millisecond
Operation : sizeOf , Collection :TreeSet, Time :1.0E-5 millisecond
Operation : remove , Collection :TreeSet, Time :5.8E-4 millisecond

----------Collection :HashSet----------

Operation : add , Collection :HashSet, Time :1.600058E-4 millisecond
Operation : collection equality , Collection :HashSet, Time :28.0 millisecond
Operation : contains , Collection :HashSet, Time :3.4E-4 millisecond
Operation : sizeOf , Collection :HashSet, Time :1.0E-5 millisecond
Operation : remove , Collection :HashSet, Time :1.1E-4 millisecond

----------Collection :HashMap----------
Operation : add , Collection :HashMap, Time :2.0E-4 millisecond
Operation : collection equality , Collection :HashMap, Time :33.0 millisecond
Operation : replace , Collection :HashMap, Time :1.9E-4 millisecond
Operation : containsKey , Collection :HashMap, Time :8.0E-5 millisecond
Operation : containsValue , Collection :HashMap, Time :0.31823 millisecond
Operation : sizeOf , Collection :HashMap, Time :2.0E-5 millisecond
Operation : remove , Collection :HashMap, Time :5.0E-5 millisecond

----------Collection :TreeMap----------
Operation : add , Collection :TreeMap, Time :1.5E-4 millisecond
Operation : collection equality , Collection :TreeMap, Time :12.0 millisecond
Operation : replace , Collection :TreeMap, Time :2.1E-4 millisecond
Operation : containsKey , Collection :TreeMap, Time :1.6E-4 millisecond
Operation : containsValue , Collection :TreeMap, Time :0.45842 millisecond
Operation : sizeOf , Collection :TreeMap, Time :0.0 millisecond
Operation : remove , Collection :TreeMap, Time :5.3E-4 millisecond

From the results above we can observe that the add function in all the sets of collection the least time taken is by LinkedList, which concludes that for insertion of data in a list this data structure can be used to achieve best performance. The same is the case when we have to remove any element from the data structure. In conclusion, even though some data structure might give best time performance but there are certain cases where a specific data structure will always be preferred because of its relevance pertaining to the problem to be solved.

Assignment 2:

This assignment asked to think about real life examples where the above implemented data structures can be the best structure used for efficient working of that application.

Case 1:

When a user login for the first time in any website, say - *Facebook, Gmail, Quora* etc. They have to create a user account for them, they have to provide information about their name, the date of birth (in some cases), and the most important ones – username and password. The username has to unique to every individual account, the applications doesn't accept a username if it is already used by some other user. So how do these applications store that information? We speculate that the username and passwords are stored in a HashMap, where the username acts as a key and the password the value. Since HashMap doesn't allow any duplication of the key data, it gives the perfect support to the app to check whether the key is already used by anyone else.

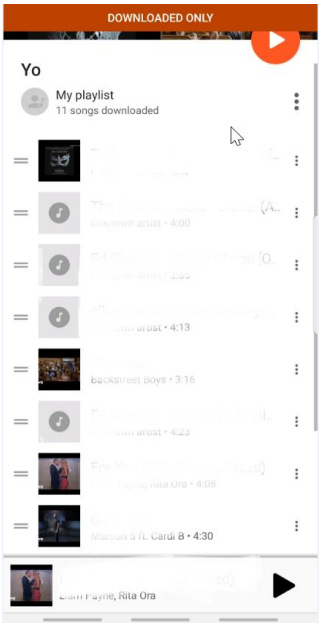
Why this can be advantageous? HashMap while retrieving the data does not guarantee the order it was stored in which makes accessing of a key, in large applications like mentioned above, easier and takes less time than any other data structure.

It also inserts, searches and deletes the data in O(1) time complexity, which is gives the best performance. Our statistics which we obtained from our program in Assignment 1, also support

the notion that searching or fetching the key takes less time, 8.0 X 10^{-5} millisecond, as reported. Even if the user

Case 2:

The second example, albeit taken reference from various websites, is apt of an example for were the linkedList should be used. The example of creating a music playlist. Whenever a user creates his/her customized playlist they create a list sort of music of their choice, the basic working of the playlist is that song after song plays, as per what order the user has given. In this we can say that the playlist acts as a Linked List, i.e. whenever the user wants to insert a new song or reorder the list they can easily make it work, as LinkedList supports a node to be added at any location without affecting the actual list functionality. Insertion for LinkedList takes O(1) time complexity, which gives the best performance when used.



Even in a web browser if a user browses for a website and jumps to the next site from the current one and again to the next one, if they want to return to the initial site they had browsed they can just move backward and navigate to the desired site, this clearly shows that the sites are linked to each other in the form of a list, because they are ordered in the way they are browsed.

TreeSet are those data Structures which are used when the data is ordered in any way required for the problem defined. For example, while filling up an online application, we might have to give phone number, generally we are asked to provide our country codes with the number. Frequently there is list from which we can choose the correct country to fill in the country codes. These lists are ordered, often lexicographically do that searching for one's country becomes easier. The results provided also support the fact searching for that particular key becomes much easier if the sets presented in an ordered fashion.