# CSC212P8 - Report and Reflection

*Nukhbah Majid*

*4/13/2019*

## Report:

In this assignment, we were to carry out a scientific investigation to determine which data structure is the most efficient in "Spell-Checking" a book against an existing dictionary. These were the following data structures that we examined:

```
1. TreeSet
2. HashSet
3. SortedStringListSet
4. CharTrie
5. LLHash
```

These data structures were compared on the following criteria:

```
  (i) How long does it take for each data structure to be filled?
 (ii) What is the insertion speed for an element in all these data structures?
(iii) For HashSet and TreeSet, what is the time difference of adding the data by:
         a. Adding the data straight as input.
         b. Adding the data element by element with a "for loop".
 (iv) Given some specific data that is either in the dictionary or not, how long does
      it take for each data structure to look-up the words from the chosen book in the
      dictionary and determine whether or not the word is present or not.
```

For my analysis, after examining the consrtuction and insertion speeds of the data structures with the dictionary of words, I "spell-checked" the book Dracula by Bram Stoker.

### SpeedCheck

My compiled project contains a `SpeedCheck.java` class that examines the data structures speeds as instructed. The following are the results for construction and insertion of elements into each of the data structures:

Time needed to:
fill a HashSet: 61$\mu$s
fill a TreeSet: 1356$\mu$s
fill a SortedStringListSet: 4608$\mu$s
fill a CharTrie: 2504$\mu$s
fill a LLHash: 2156$\mu$s

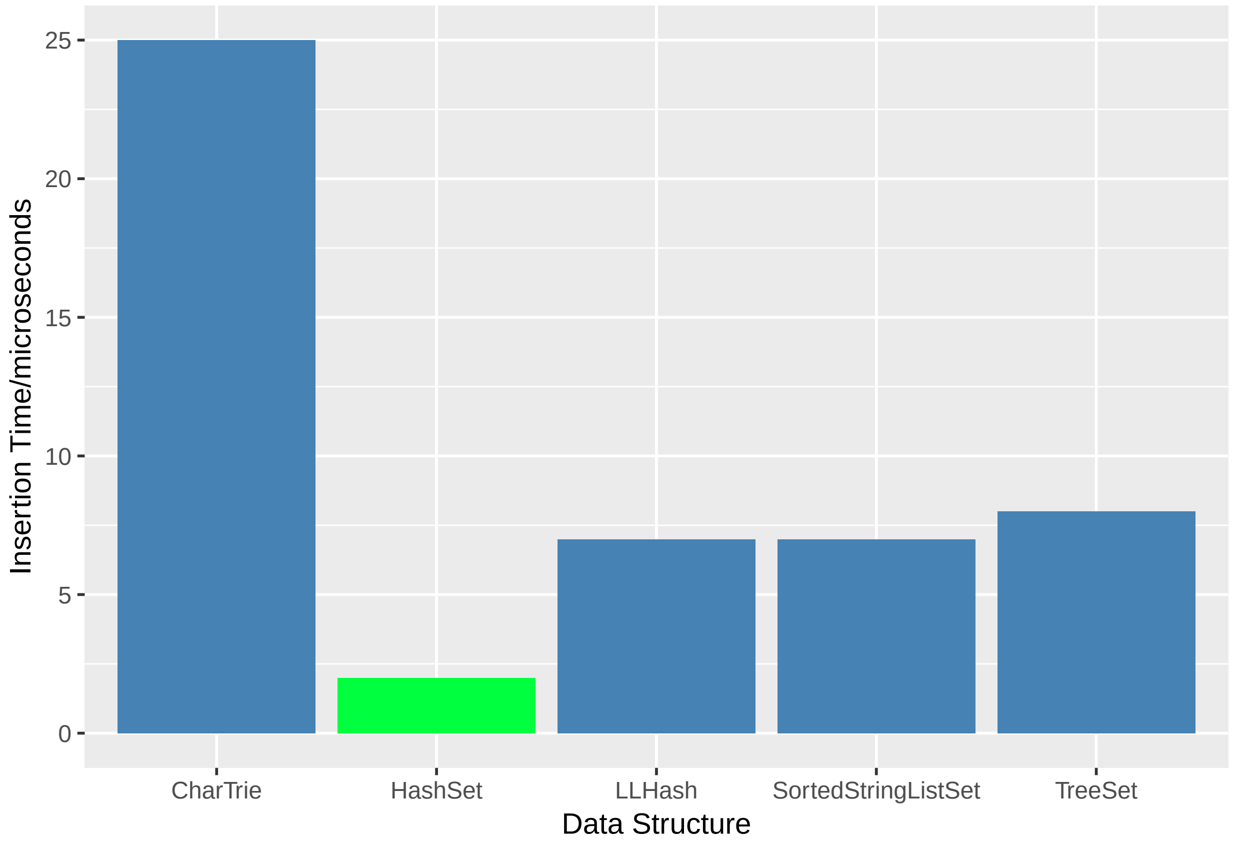
Time needed to:
construct a HashSet with its input data: 13$\mu$s
construct a HashSet by calling add in a for loop: 61$\mu$s
construct a TreeSet with its input data: 30$\mu$s

construct a TreeSet by calling add in a for loop: 1356$\mu$s

Insertion time per element for:
HashSet: 2$\mu$s
TreeSet: 8$\mu$s
CharTrie: 25$\mu$s
LLHash: 7$\mu$s
SortedStringListSet: 7$\mu$s

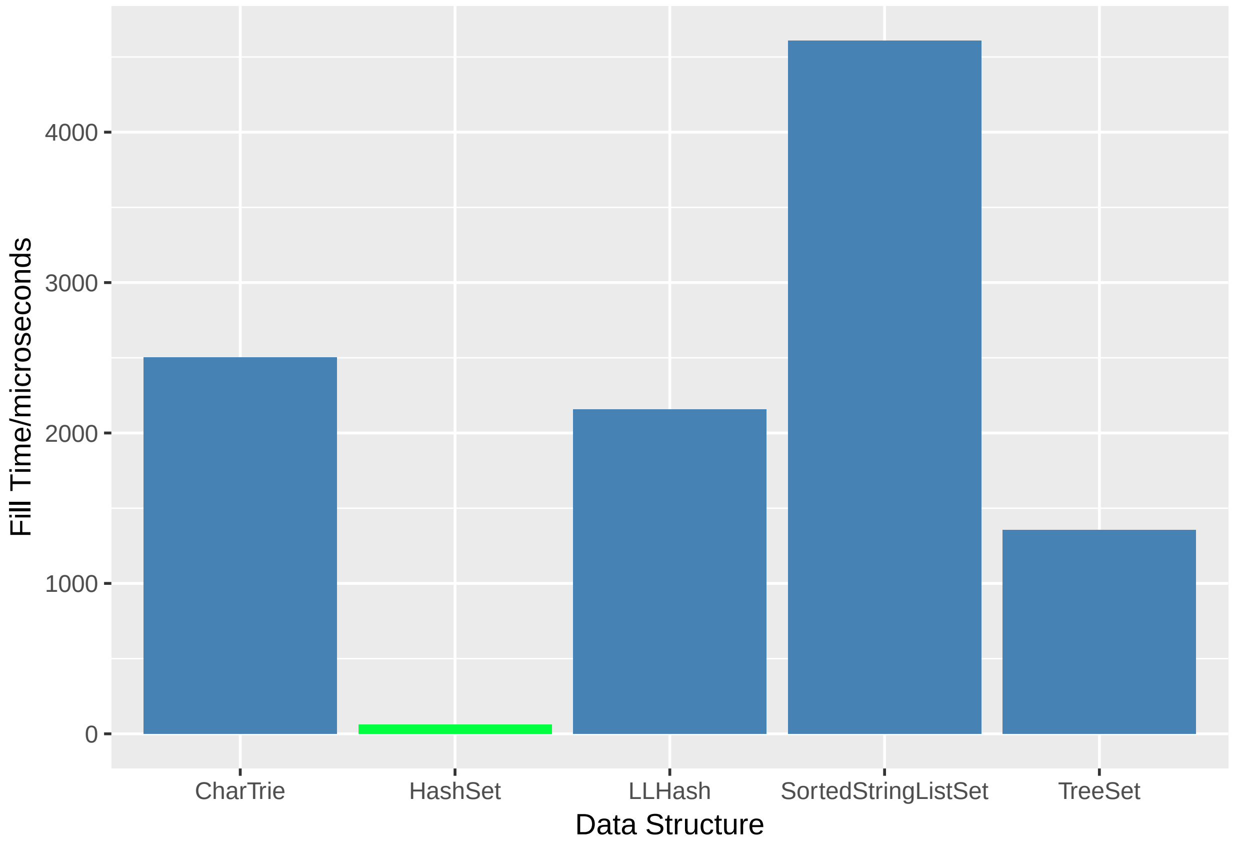
From the results above, it can be concluded that `HashSet` was the best performing data structure in this test. Not only did it form the fastest - whether with the use of a for loop or without - but also was the fastest in inserting a new element. The data structures that performed poorly were `SortedStringListSet` and `CharTrie`.

The barplot below provides a better perspective of the insertion speeds of all data structures:



As for the separate analysis of constructing `HashSet` and `TreeSet` with a **for loop** or by **inputting the data in the definition**, `HashSet` tends to perform better than a `TreeSet`. Generally, construction of either of the two data structures with a for loop took longer. In case of `TreeSet`, which is a self-balancing tree, elements are organized in a **binary tree**, hence adding an element - or accessing it - takes O(logn) time. Since the elements are arranged in a self-balancing binary search tree - also referred to as red-black tree - adding elements one by one requires “repainting” all the nodes after an addition, thereby taking longer to compute with a for loop. Whereas in the case data is inputted right in the instantiation of the `TreeSet`, the painting of the nodes only occurs once. In case of `HashSet`, the buckets in the set are by default 16, however because the load factor determines when to resize the `HashSet`, as soon as the number of inputs in the `HashSet` exceeds the product of **load factor** and the current capacity, the `HashSet` is resized. Adding elements and accessing them takes on average O(1) time. This is because the underlying data structure of a `HashSet` is a **hash table**, hence the elements are stored with the help of a **hash function**. A hash function randomly generates a **hashCode** that, ideally, decreases the chances of collisions; the look-up time for the hash code in memory is O(1), and hence the amortized complexity of `HashSet` for adding elements and accessing them is O(1).

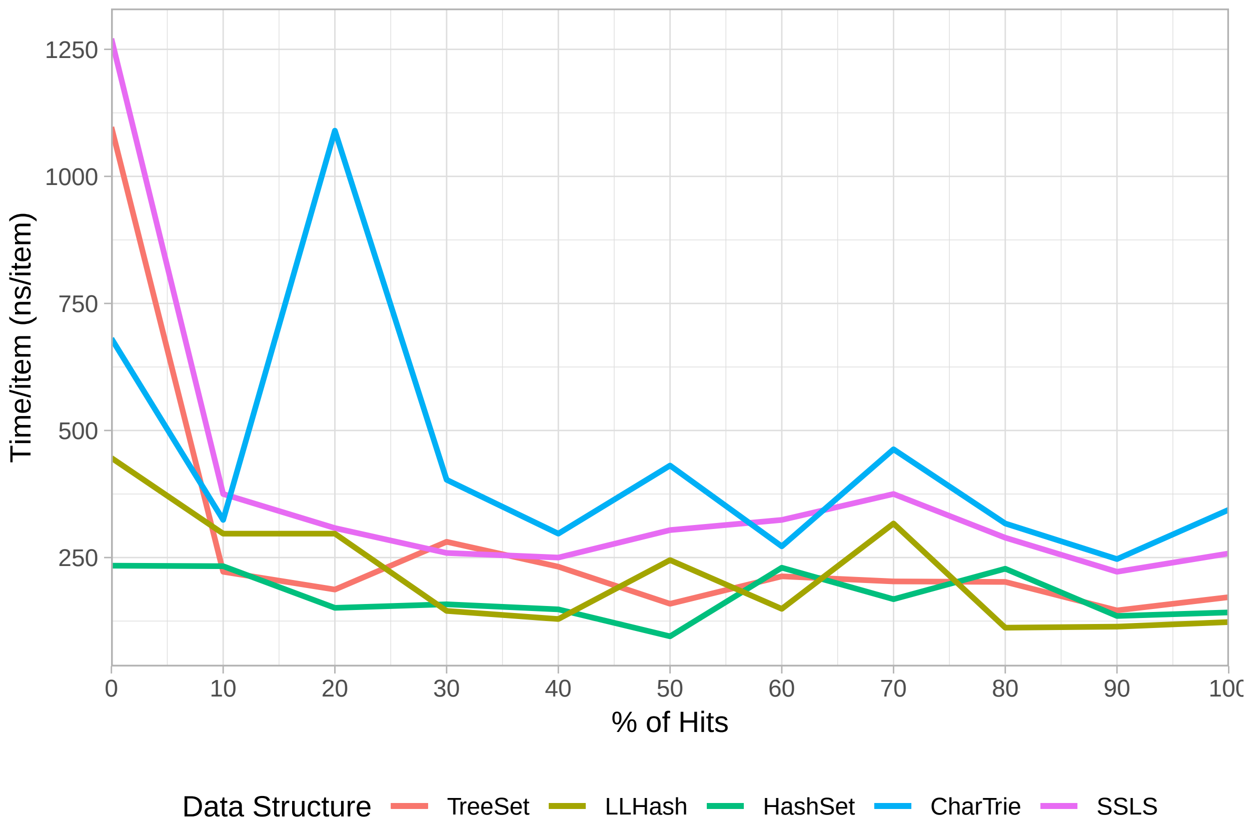
The conclusive fill times for all the Data Structures are also plotted below as a bar plot:



We can observe from this bar plot that there is a huge difference in the fill times of a few Data Structures with others. For example, `CharTrie` and `SortedStringListSet` take particularly long to fill up as compared to `HashSet` and `TreeSet`. It can also be observed from the bar plot that a `HashSet` performs the best in terms of fill time. The second best data structure is a `TreeSet`. The difference between the performances of these two data structures alone is pretty significant. This is because the time complexity for adding and accessing elements in a `HashSet` is O(1), whereas the time complexity of adding and accessing elements in a `TreeSet` is O(logn). This doesn't necessarily mean that O(1) complexity is always better than O(logn) complexity, but it's only for large values of `n` that constant time complexity proves favorable. As our tests require spell-checking against a huge dictionary, constant time complexity is almost always favorable.

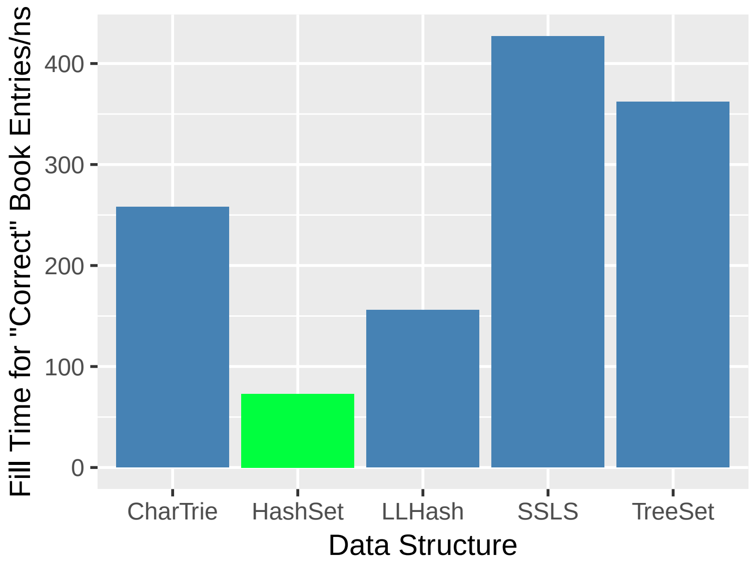
## Hits and Misses Analysis of Query Speeds for each Data Structure

Furthermore, we were to compare and examine the look-up speeds of words stored in those five data structures such that **some fraction** of the words were found in the UNIX dictionary - the number of **hits** - whereas the rest were not - the number of **misses**. Varying the fraction of legit words from 0 to 1 in increments of 0.1, these were the conclusive results for all the data structures:



### Spell-Checking a Project Gutenberg Book

In the next phase of project, we were supposed to load a Project Gutenberg book and check all the words in it if they had matches in the UNIX dictionary, whether they are "correct" are not. The book I chose is a classic called `Dracula` by Bram Stoker. The query speeds for each of the data structures to fill up - after checking the presence of each entry in the dictionary - follows relatively the same pattern. The bar plot below depicts these speeds:



The ratio of misspelled words was found to be ~0.08 - conversely the ratio of found, and thereby "correct" words, was ~0.92. The number of words misspelled were 13628 out of total 166916 words. A few examples of misspelled words were:

```
    Example of some mispelled words:
           [0] the
           [1] gutenberg
           [2] ebook
           [3] dracula
           [4] bram
           [5] ebook
           [6] restrictions
           [7] terms
           [8] gutenberg
           [9] ebook
           [10] online
```

Upon inspecting the entire list of misspelled words, I noticed that most of them are either just plurals or past tenses of pre-existing words in the UNIX dictionary.

## Why are most binary search implementations wrong?

Most binary search implementations are incorrect because of the way the `mid` pointer is configured. This is a problem I also ran into when trying to implement iterative merge sort in P7. The flaw is that most implementations configure the `mid` pointer in the following way:

```
int mid = (low + high) / 2;
```

This leads to many off-by-one errors - running into `IndexOutOfBounds` error because it is not suited for large values of `low` and `high` as mentioned in this article. Therefore, my implementation, as per this article, fixes the binary search method as shown below:

```
private int binarySearch(String query, int start, int end) {
        int mid;
        while (start < end) {
            mid = start + (end-start) / 2;
            if (query.compareTo(data.get(mid)) == 0) {
                return mid;
            } else if (query.compareTo(data.get(mid)) < 0) {
                end = mid;
            } else {
                start = mid + 1;
            }
        }
        return -1;
}
```

## LLHash Size

After completing the `countCollisions` and `countUsedBuckets` I tried to adjust the `numBuckets` argument of the `LLHash`. Increasing the size of the LLHash causes to take more time to look up for elements. This means the number of used buckets **increases** and the number of collisions **decreases**.

## Conclusion

While all data structures have some fundamental attributes that make them unique, with respect to the task of spell-checking, I believe that a `HashSet` is the most appropriate data structure. Main reasons for this conclusion are that a `HashSet` carries out the **adding, deleting, and looking-up** (contains method) in amortized O(1) time, which is optimum for large inputs of `n` (in this case the entries in the dictionary and book respectively). Besides favorable space-time complexity of certain operations, choosing `HashSet` still is a viable choice - this is because we're only checking for correctness of words in the dictionary which is a binary decision - the words are either in the dictionary or not in the dictionary. If we were to implement a "prefix search" or tried to determine the closely related words of a particular entry, then certainly a `TreeSet` or a `CharTrie` would have been appropriate. For the sake of optimum space-time complexity and looking up for the presence of certain words in a dictionary, a `HashSet` suffices.