Mini Project 2017 – Muhammad Sajid Alam 150073455

Code:

```
interface WordStore
{
    public void add(String word);
    public int count(String word);
    public void remove(String word);
}

class WordStoreImp implements WordStore
{
    List[] theArray;
    int arraySize;

    public WordStoreImp(int size){
        arraySize = size;
        theArray = new List[size];
        for (int i = 0; i < arraySize; i++) {
            theArray[i] = new List();
        }
    }

    public void add(String word){
         if(arraySize == 0){
              return;
         }
        int hashKeyValue = hashFunction(word);
        Word WordObj = new Word(word, hashKeyValue);
        theArray[hashKeyValue].insert(WordObj);
    }

    public int count(String word){
         if(arraySize == 0){
              return 0;
         }
        int hashKeyValue = hashFunction(word);
        int counter = theArray[hashKeyValue].find(hashKeyValue, word);
        return counter;
    }

    public void remove(String word){
         if(arraySize == 0){
              return;
         }
        int hashKeyValue = hashFunction(word);
        theArray[hashKeyValue].removeWord(word);
    }

    public int hashFunction(String word){
        int hashKeyValue = 0;
        for (int i = 0; i < word.length(); i++) {
            int charCode = word.charAt(i) - 96;
            hashKeyValue = (hashKeyValue * 27 + charCode) % arraySize;
        }
        return hashKeyValue;
    }
}
```

```
class List
{
    private Word latestWord = null;

    public void insert(Word newWord){
        Word previousWord = latestWord;
        latestWord = newWord;
        latestWord.setNext(previousWord);
    }

    public int find(int hashKey, String wordToFind){
        Word currentWord = latestWord;
        int count = 0;
        while (currentWord != null){
            if (currentWord.getWord().equals(wordToFind))
                count++;
            currentWord = currentWord.getNext();
        }
        return count;
    }

    public void removeWord(String wordToRemove){
        Word currentWord = latestWord;
        if (currentWord == null){
            return;
        }
        else if (currentWord.getWord().equals(wordToRemove)){
            latestWord = currentWord.getNext();
            return;
        }
    }
}

class Word {
    private String theWord;
    private int key;
    private Word next;

    public Word(String newWord, int hashKeyValue) {
        this.theWord = newWord;
        this.key = hashKeyValue;
    }
    public String getWord(){
        return theWord;
    }
    public int getKey(){
        return key;
    }
    public Word getNext (){
        return next;
    }
    public void setNext (Word nextWord){
        next = nextWord;
    }
}
```

Code Explanation:

As well as implementing the interface `WordStore` into `WordStoreImp` I have also created two other classes namely `List` and `Word`. These files are used together to define a collection of words. During the project I decided on using a hash table data structure to create an efficient implementation. Through studying various websites, I figured out that hash tables were a type of data structure that gives each 'keys', in my case Strings, to a value through a hash function (an algorithm that generates a value) [1]. These values could then be used to map each 'key' to their respective position in an array [1]. This allows us to add, remove and see if a key is stored very rapidly by simply using the appropriate hash function to generate its corresponding value. The alternative would be to sort all they keys and use several loops to achieve the same. This would be a much slower and inefficient process which is why I decided on hash tables. Upon further reading I also learnt that hash tables can have issues such as collisions where the same hash value is produced. There are several strategies to solve this such as open addressing, separate chaining and dynamic resizing [2]. I decided on using separate chaining strategy as it involved using Linked Lists, a data structure I felt confident in implementing.

While researching separate chaining I found a hash function that is efficient and unlikely to lead to many collisions. The hash function is written in method `hashFunction()` in the class `WordStoreImp`. The algorithm works by taking each characters ASCII code and taking it away from the ASCII code of 'a', this number is then added onto the current hashKeyValue which has been multiplied by 27(represents all letters from a to z and a space). The modulo of this value is taken from array size to ensure hash value isn't larger than the array [3]. Once it has gone through entire word it returns the hash key value.

The `WordStoreImp` class creates an array of `List` in `theArray` variable with its size initialised in the constructor with the method argument, `size`. Each `List` object has `latestWord` which stores a pointer to null or most recent entry in link list. When a word, "hello" for example, is added for the first time it will go through the `add` method in `WordStoreImp` where a hash key value is generated using the `hashFunction`. The value for "hello" for array size 100 would be 30. I then create a `Word` object called `WordObj` where I store in its private fields "hello" and 30 in `theWord` and `key` variables respectively. Then within position 30 of `theArray` within that `List` class the method insert is called to add `WordObj` the link list. My insert method lets the first `Word` object always point to null while every consecutive `Word` points to the `Word` object that comes after it by using the `setNext` method found in `Word` class. As a result, the `List` class' private variable `latestWord` object always stores the most recent Word added and that Word will always be pointing to previous the previous `Word` so on and so forth until it reaches the first `Word` which will be pointing to null. I have drawn diagram below to help illustrate the result of adding 2 "hello" words.

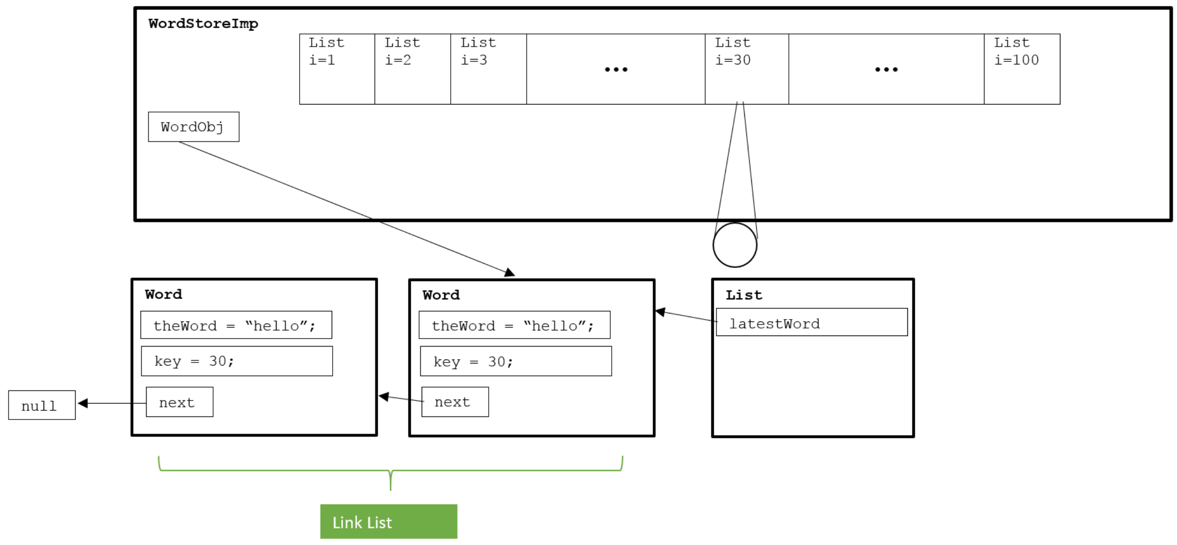


Diagram 1: Result of adding the word "hello" twice.

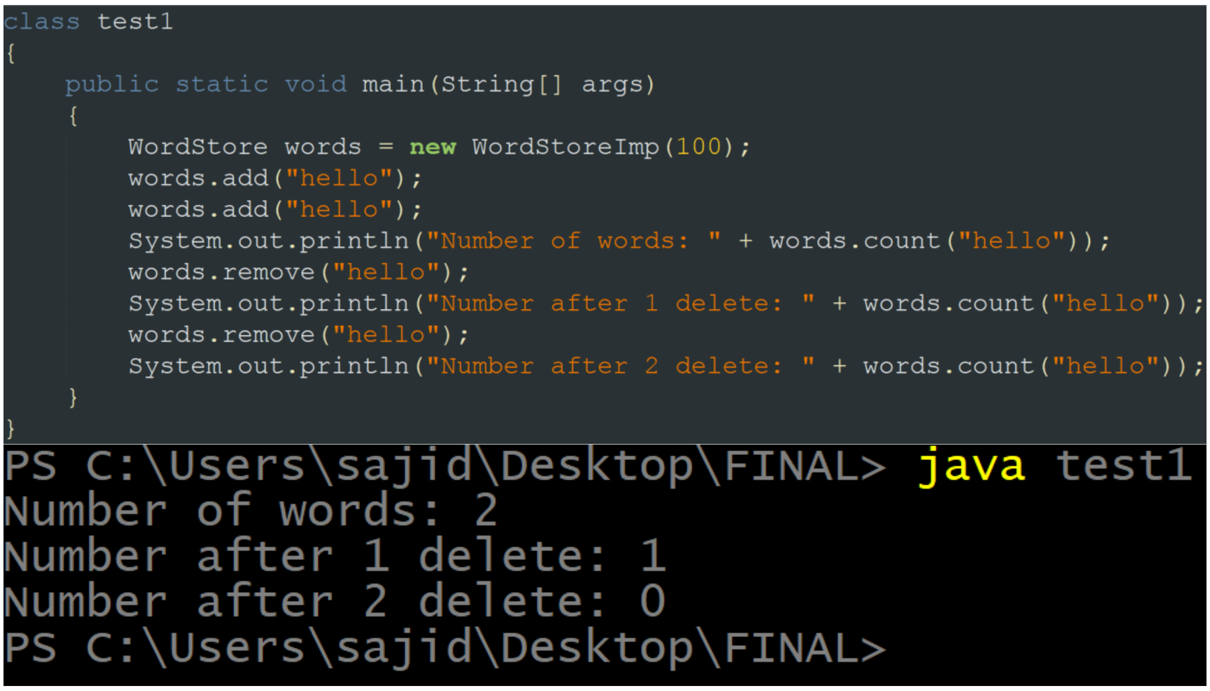
Muhammad Sajid Alam
150073455

Since I implemented hash tables I am able to quickly find and count how many words there are. The `count` method in WordStoreImp calculates hash key value and uses that to call the `find` method in correct `list` object. This method then has to simply run a while loop going through the link list, by using temporary pointers and `getNext` method in `Word`, which returns pointer to next word in link list, until it reaches null. If there are no words in it then it will be null by default and return 0. Within the while loop I also have an if statement to make sure the word in `Word` object is the same (I have created getters for some variables in `Word` class such as `getNext`). In the case of Diagram 1 when word "hello" is wanting to be counted, the counter will only increment twice before `next` points to null. At this point I simply return the counter back through `find` and `count` methods.

Lastly to remove one occurrence of the word I once again in method `remove` in `WordStoreImp` calculate hash key value and find appropriate `List` and this time call `removeWord` method, which will either do nothing if `latestWord` variable is pointing to null or will set the `latestWord` variable to point to `latestWord.getNext()` essentially bypassing the most recent entry of the `word` object and making the previous `Word` object become the latest.

<u>Correctness and Testing of Code:</u>

Simple Test



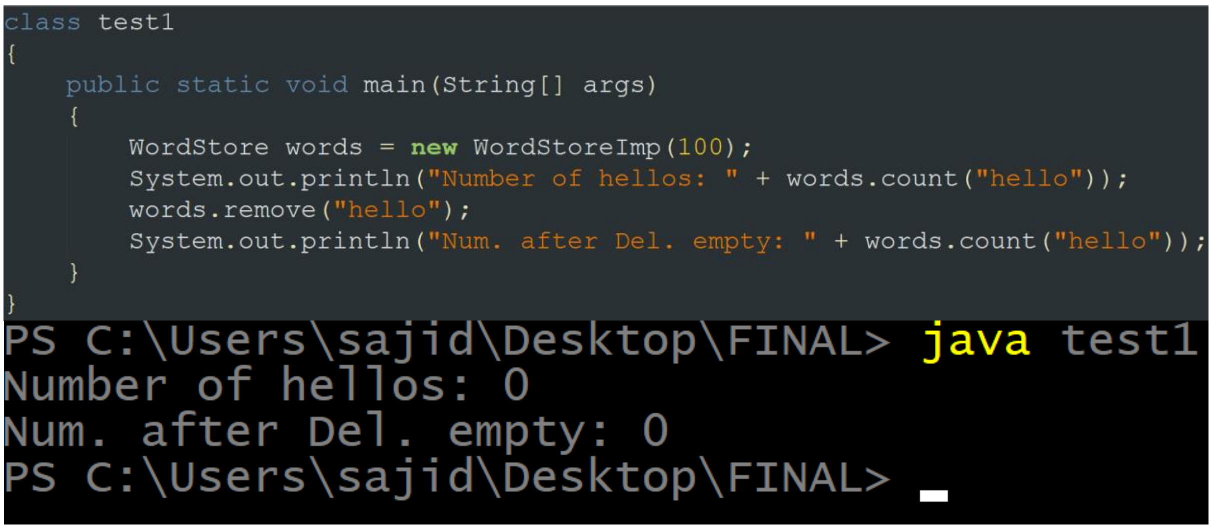
```
class test1
{
    public static void main(String[] args)
    {
        WordStore words = new WordStoreImp(100);
        words.add("hello");
        words.add("hello");
        System.out.println("Number of words: " + words.count("hello"));
        words.remove("hello");
        System.out.println("Number after 1 delete: " + words.count("hello"));
        words.remove("hello");
        System.out.println("Number after 2 delete: " + words.count("hello"));
    }
}
```

```
PS C:\Users\sajid\Desktop\FINAL> java test1
Number of words: 2
Number after 1 delete: 1
Number after 2 delete: 0
PS C:\Users\sajid\Desktop\FINAL>
```

Diagram 2: Correctness testing of adding 2 words and removing

In diagram 2 I carry out a simple test to show that my code is running correctly. I add 2 "hello" words to `words` using the add method in `WordStoreImp` I then print out the number of "hello" words in the list. From the terminal it successfully managed to count number of "hello" word in the list by printing out 2. I then remove while printing count each time. The counts printed after delete are also correct.

Corner Case Tests

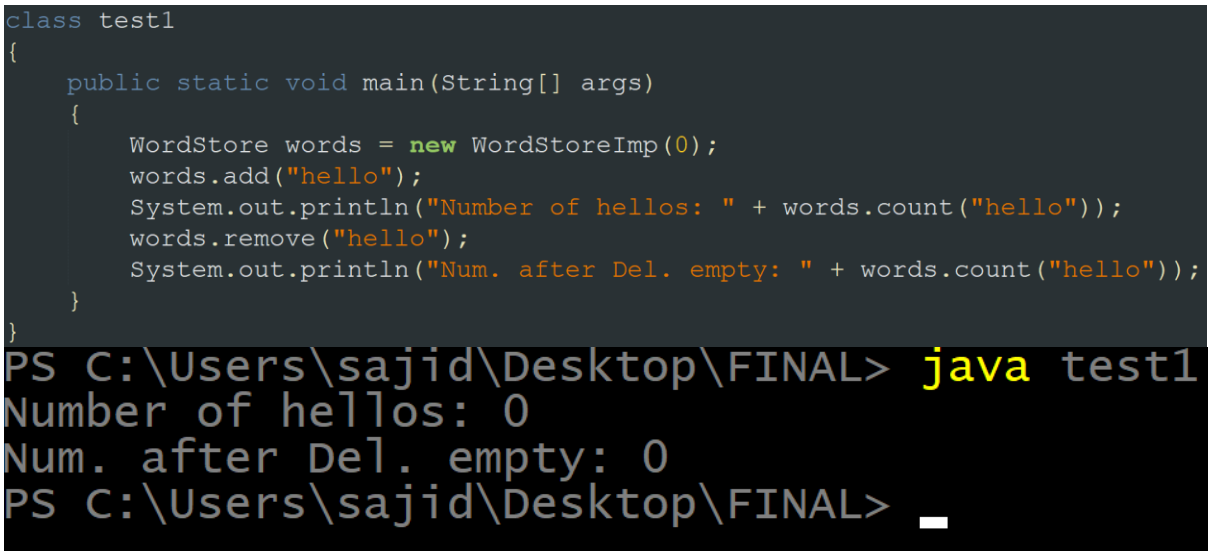


```
class test1
{
    public static void main(String[] args)
    {
        WordStore words = new WordStoreImp(100);
        System.out.println("Number of hellos: " + words.count("hello"));
        words.remove("hello");
        System.out.println("Num. after Del. empty: " + words.count("hello"));
    }
}
```

```
PS C:\Users\sajid\Desktop\FINAL> java test1
Number of hellos: 0
Num. after Del. empty: 0
PS C:\Users\sajid\Desktop\FINAL> _
```

Diagram 3: Correctness testing for counting empty list

Since it is removed, the pointer to `word` object in the `list` for "hello" will be pointing to null therefore by default returns 0.



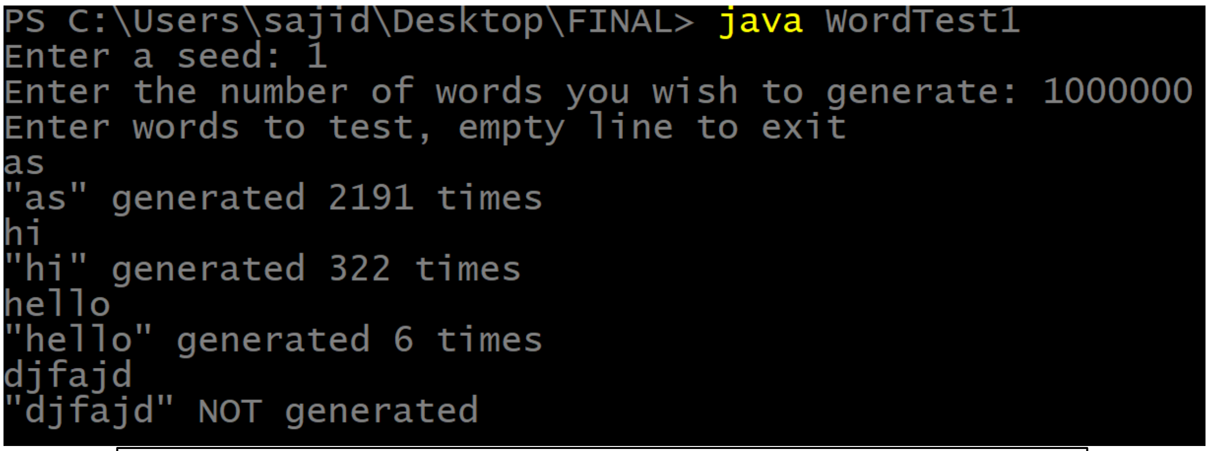
```
class test1
{
    public static void main(String[] args)
    {
        WordStore words = new WordStoreImp(0);
        words.add("hello");
        System.out.println("Number of hellos: " + words.count("hello"));
        words.remove("hello");
        System.out.println("Num. after Del. empty: " + words.count("hello"));
    }
}
```

```
PS C:\Users\sajid\Desktop\FINAL> java test1
Number of hellos: 0
Num. after Del. empty: 0
PS C:\Users\sajid\Desktop\FINAL> _
```

Diagram 4: Correctness testing for adding, removing and counting in empty collection

If there is an empty collection I have an if statement for each add, count and remove method that if it finds `arraySize` to be set to 0 then it will either not add or remove any new words or return 0 for count. Since my hash functions relies on being divided by `arraySize` I can not allow it to go through the 3 methods in `WordStoreImp`.

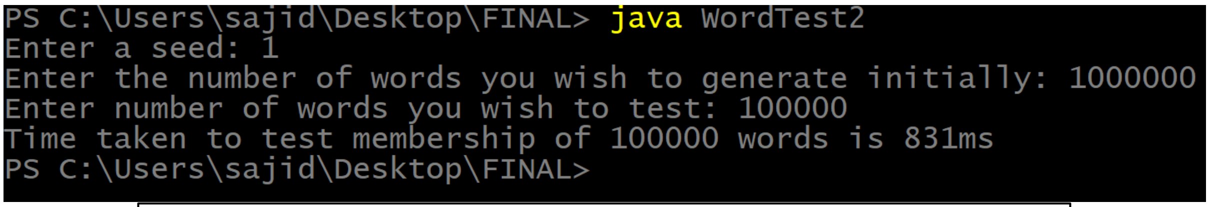
Large Tests



```
PS C:\Users\sajid\Desktop\FINAL> java WordTest1
Enter a seed: 1
Enter the number of words you wish to generate: 1000000
Enter words to test, empty line to exit
as
"as" generated 2191 times
hi
"hi" generated 322 times
hello
"hello" generated 6 times
djfajd
"djfajd" NOT generated
```

Diagram 4: Correctness testing for generating 1 million words and counting correctly

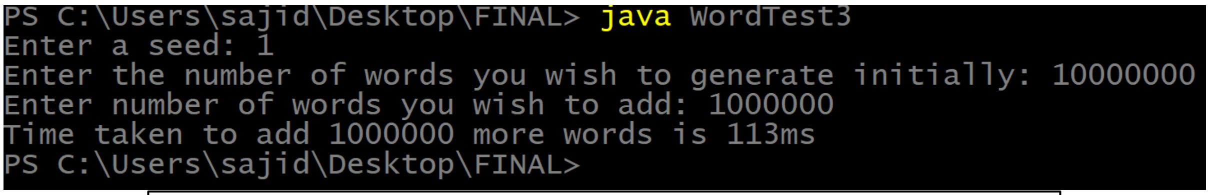
In Diagram 4 I use WordTest1 and generate 1 million words and count as to how many words there are. This shows my code can handle very large count.



```
PS C:\Users\sajid\Desktop\FINAL> java WordTest2
Enter a seed: 1
Enter the number of words you wish to generate initially: 1000000
Enter number of words you wish to test: 100000
Time taken to test membership of 100000 words is 831ms
PS C:\Users\sajid\Desktop\FINAL>
```

Diagram 5: Correctness testing for counting 100k words in collection of 1 million

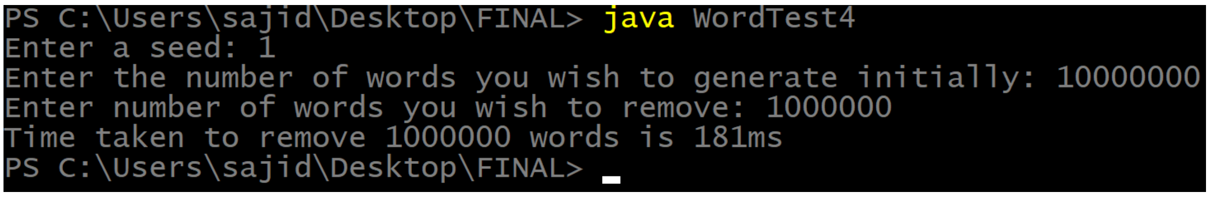
In Diagram 5 I use WordTest2 to again generate 1 million words and search for 100k words. Going beyond this count of words slows down my code. My code is able to handle several hundred thousand-word counts.



```
PS C:\Users\sajid\Desktop\FINAL> java WordTest3
Enter a seed: 1
Enter the number of words you wish to generate initially: 10000000
Enter number of words you wish to add: 1000000
Time taken to add 1000000 more words is 113ms
PS C:\Users\sajid\Desktop\FINAL>
```

Diagram 6: Correctness testing for adding 1 million words to 1 million collections

In Diagram 6 I use WordTest3 to generate 10 million words and test how long it takes to add 1 million to it. My code allows me to add 1 million words to 10 million. Creating a collection of size larger then 10 million as there is not enough memory.



```
PS C:\Users\sajid\Desktop\FINAL> java WordTest4
Enter a seed: 1
Enter the number of words you wish to generate initially: 10000000
Enter number of words you wish to remove: 1000000
Time taken to remove 1000000 words is 181ms
PS C:\Users\sajid\Desktop\FINAL> _
```

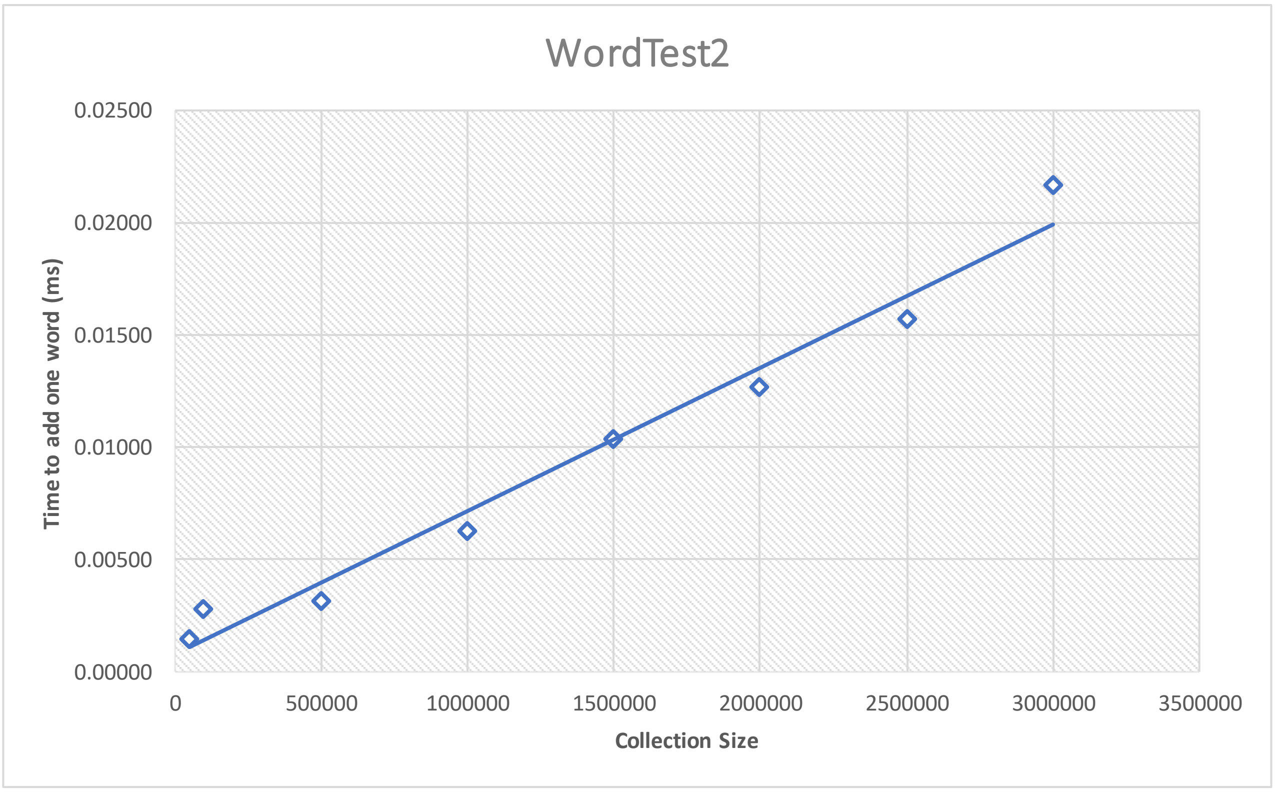
Diagram 7: Correctness testing for removing 1 million words from 1 million collections

Diagram 7 shows similar to Diagram 6 but this time removing 1 million words.

Efficiency testing

WordTest2 Results:
Seed: 1
Words Being Tested: 10000

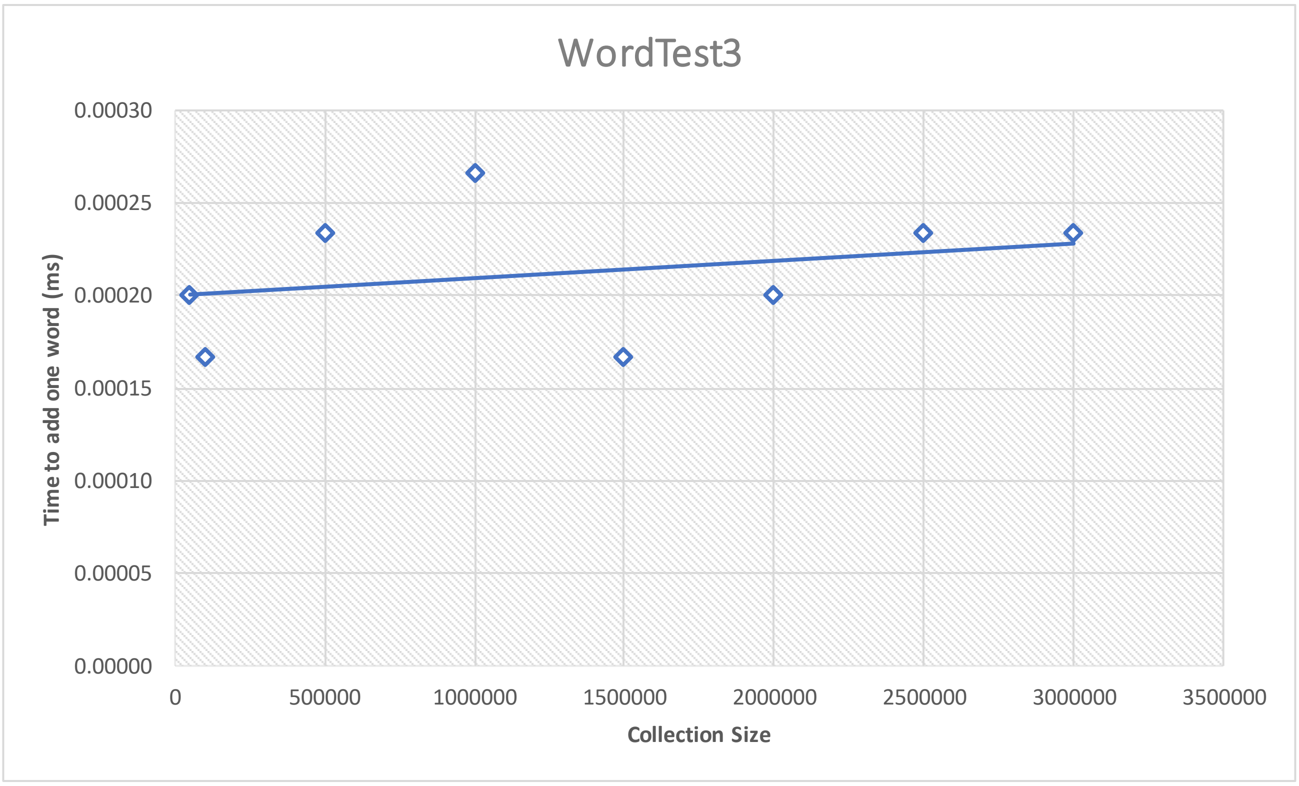
WordTest2					
Collection Size	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Time Average (ms)	Time for one Word (ms)
50000	15	16	13	14.7	0.00147
100000	25	29	30	28.0	0.00280
500000	31	34	31	32.0	0.00320
1000000	85	52	52	63.0	0.00630
1500000	101	108	102	103.7	0.01037
2000000	103	97	180	126.7	0.01267
2500000	157	158	157	157.3	0.01573
3000000	252	142	257	217.0	0.02170



Muhammad Sajid Alam
150073455

WordTest3 Results:
Seed: 1
Words Being Added: 10000

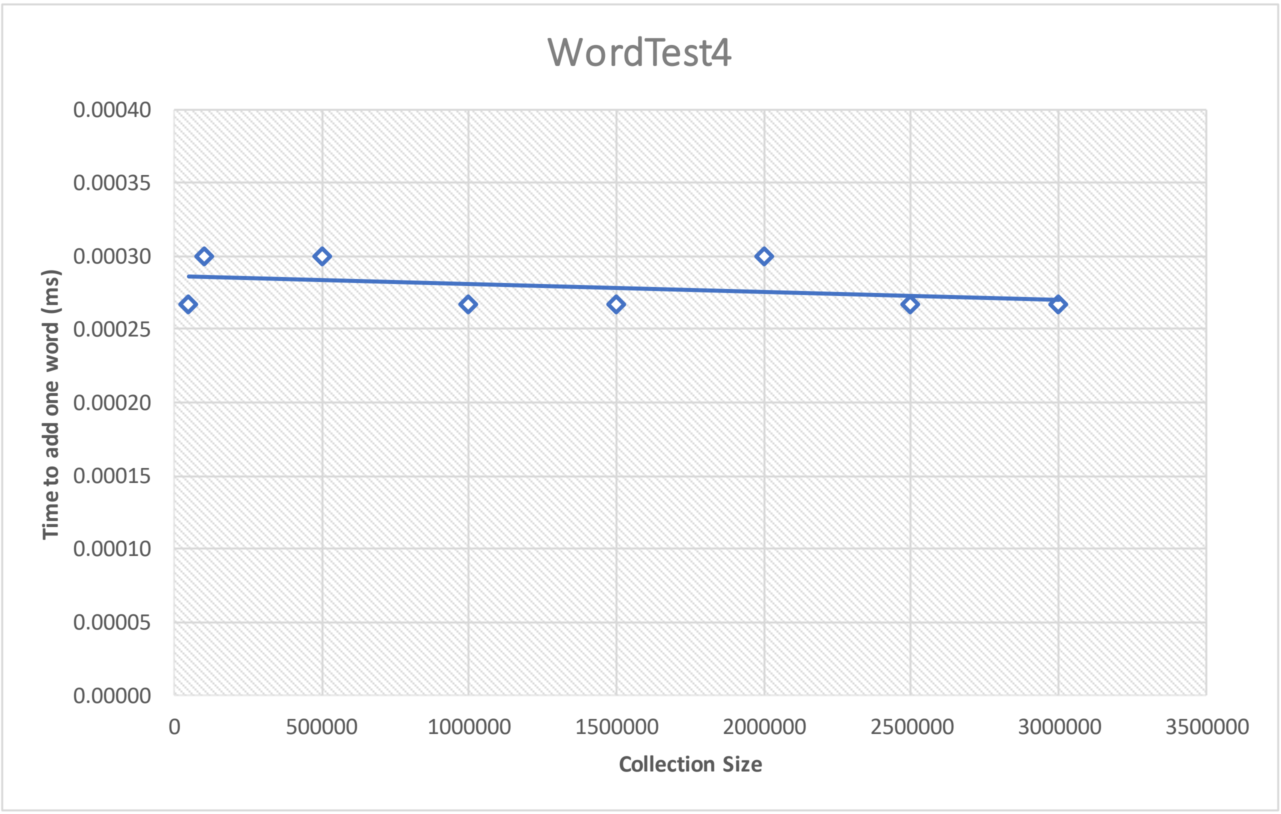
WordTest3					
Collection Size	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Time Average (ms)	Time for one Word (ms)
50000	2	2	2	2.0	0.00020
100000	2	2	1	1.7	0.00017
500000	3	2	2	2.3	0.00023
1000000	3	2	3	2.7	0.00027
1500000	2	1	2	1.7	0.00017
2000000	2	2	2	2.0	0.00020
2500000	3	2	2	2.3	0.00023
3000000	2	2	3	2.3	0.00023



Muhammad Sajid Alam
150073455

WordTest4 Results:
Seed: 1
Words Being removed: 10000

WordTest4					
Collection Size	Time 1 (ms)	Time 2 (ms)	Time 3 (ms)	Time Average (ms)	Time for one Word (ms)
50000	3	2	3	2.7	0.00027
100000	3	3	3	3.0	0.00030
500000	3	3	3	3.0	0.00030
1000000	2	3	3	2.7	0.00027
1500000	3	3	2	2.7	0.00027
2000000	3	3	3	3.0	0.00030
2500000	2	3	3	2.7	0.00027
3000000	3	3	2	2.7	0.00027



Muhammad Sajid Alam
150073455

I tested the efficiency of my code by using WordTest2, WordTest3 and WordTest4 to see how efficient my code is at counting, adding and removing words. For each experiment I kept the number of words I was adding, removing or counting constant at 10k. I believe this was a reasonable number to demonstrate my codes efficiency as I can collect meaningful times. The seed which I used was also kept constant at 1. My independent variable was the collection size that I varied to see how it affected time. I repeated the collecti on size 3 times and recorded averages to generate reliable results. I also calculated how long it would take to carry out the operation for 1 word by diving the time by my collection size. My graphs are plotted with time for one word against collection size. In WordTest2 my graph is showing a positive linear correlation where as collection size increases so does time taken to test membership of 1 word. The theoretical complexity suggests it as O(N) shown by the linearity in the graph. This is due to the while loop found in `List` class under the `find` method. To test for membership the while loop goes through every single thing in link list to count it so the bigger the collection the longer the while loop will run. I can solve this by adding a counter variable to `List` which will increment and decrement every time a word is added or removed from the list. This would then allow me to simply return the counter variable from `find` method.

My WordTest3 and WordTest4 show a similar trend in that they have similar gradients where its almost 0. Both tests, for every sized collection always ranged between 2 and 3 milliseconds. No matter how large the collection size became, adding and removing always took roughly the same time. This is a benefit from using separate chaining hash tables which allows for quick addition and removals by appending on to end of the link list or simply bypassing to remove. Both these graphs suggest the theoretical complexity of it to be O(1) where regardless of the size of the array it can still carry out its function with constant time.

Muhammad Sajid Alam
150073455

Bibliography:

[1] - www.tutorialspoint.com. (2017). *Data Structures and Algorithms Hash Table*. [online] Available at: https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm [Accessed 7 Dec. 2017].

[2] - Algolist.net. (2017). *HASH TABLE (Java, C++) | Algorithms and Data Structures*. [online] Available at: http://www.algolist.net/Data_structures/Hash_table [Accessed 7 Dec. 2017].

[3] - YouTube. (2017). *Java Hash Tables 3*. [online] Available at: https://www.youtube.com/watch?v=SVsT7oG4ap8&t=501s [Accessed 7 Dec. 2017].