ECS510 Algorithms and Data Structures in an Object-Oriented Framework
Mini Project 2017

Implementing a String Collection Class

Submitted by:
Alexander Kanouni Hassani--Bornerand
Student ID: 160588750
Submitted on: 11/12/2016
No. of pages: 15

Table of Contents

1. Code Section 3

2. Design decisions and their explanations 8

2.1 Algorithms 8

2.2 Data Structure 10

3. Experimental Results 12

3.1 Membership Test 12

3.2 Add Test 13

3.3 Remove Test 14

Works Cited 16

```
/**
 *
 * @author alexandrebornerand
 */

public class WordStoreImp implements WordStore{
    private WordStoreDT[] wordsStored;
    int count;
    //constructor
    public WordStoreImp(int size){
        wordsStored = new WordStoreDT[size];
        count=1;
    }

    /*** EXPAND array ***/

    private void expand() {
        WordStoreDT[] wordsStored1 = new WordStoreDT[wordsStored.
           length*2];
        for (int i=0; i<wordsStored.length; i++)
            wordsStored1[i] = wordsStored[i];
        wordsStored=wordsStored1;
    }//END expand

    /*** ADD METHOD ***/

    @Override
    public void add(String word) {
        int index = hash(word);
        WordStoreDT<String> ws = wordsStored[index];
        if (ws==null) wordsStored[index] = new WordStoreDT<>(word,null
           );

        else {
            if (!isRepeated(wordsStored[index], word)) {
                wordsStored[index] = new WordStoreDT<>(word,
                  wordsStored[index]);
            }
        }
        count++;
        if (count >= 0.75*wordsStored.length)
            //ensures at least 25% of array remains empty in order to
              accomodate more additions and avoid collisions.
            expand();
    }//END add method

    /**** COUNT METHOD ******/

    @Override
    public int count(String word) {
        //returns the number of times a string appears in the
           collection.
        int index = hash(word);
        WordStoreDT<String> ws = wordsStored[index];
        if (ws == null){
```

```java
            return 0;
        }
        else {
            WordStoreDT<String> ptr = ws;
            for (; ptr!=null; ptr = ptr.next) {
                if (word.equals(ptr.first))
                    return ptr.count;
            }
        }
        return 0;
    }//END count

    /**** REMOVE METHOD ******/

    @Override
    public void remove(String word) {
        //removes one occurence of a String from the collection, or
           leaves the collection unchanged if the String does not
           occur in it.
        int index = hash(word);
        WordStoreDT<String> ws = wordsStored[index];
        if (ws!=null) {
            WordStoreDT<String> ptr = ws;
            for (;ptr.next!=null&&!word.equals(ptr.next.first);ptr=ptr
               .next) {
                if (word.equals(ptr.first)) {
                    if (ptr.count>1) {
                        ptr.count--;
                        return;
                    }
                    else if (ptr.count == 1) {
                        if (ptr.next!=null){
                            count--;
                            wordsStored[index] = wordsStored[index].
                             next;}
                        return;

                    }
                }
            }
        }
    }//END remove

    /**** INITIAL HASH FUNCTION ****/

    private int hash_initial(String word) {
        //initial hash function, too slow

        //converts each letter of a string into its ASCII equivalent,
        //sums up the numbers and returns the remainder when divided
           by the size of the array.
        int sum=0;
        char[] ascii = word.toCharArray();
        for (int i=0; i<ascii.length; i++) {
            sum = 43*sum*ascii[i];
        }
        return Math.abs(sum)%wordsStored.length;
    }//END hash_initial

    /**** FINAL HASH FUNCTION ****/

    private int hash(String word) {
        //final hash functin, different hashing algorithm
        int hashKey =0;
        for (int i=0; i<word.length(); i++)
            hashKey=31*hashKey+word.charAt(i);
        return Math.abs(hashKey)%wordsStored.length;
    }//END hash

    /*** ISREPEATED METHOD *****/

    private boolean isRepeated(WordStoreDT ws, String word) {
        WordStoreDT<String> ptr = ws;
        while (ptr.next!=null) {
            if (ptr.first.equals(word)) {
                ptr.count++;
                return true;
            }
            ptr=ptr.next;
        }
        return false;
    }//END isRepeated

    /*** TRIAL METHOD: displays the words in array ***/
    //In order for this method to work, it must be added in
       WorstStore.java

    @Override
    public void display(){
        for(int i=0;i<wordsStored.length; i++) {
            if (wordsStored[i]==null) {/*System.out.println("oops");
                Scanner scanner = new Scanner(System.in);
                System.out.println("Enter a word to add");
                String word = scanner.nextLine();
                System.out.println(hash(word));*/
            continue;}
            WordStoreDT<String> current = wordsStored[i];
            while(current!=null) {
                System.out.println(current.first);
                current=current.next;
            }
            System.out.println();
        }
    }//END display

    /************* WORDSTOREDT CLASS ****************/

    private static class WordStoreDT <String> {
        String first;
        WordStoreDT<String> next;
        int count;

        WordStoreDT(String f, WordStoreDT<String> n) {
            first = f;
            next = n;
            count = 1;
        }
    }//END class WordStoreDT
}//END class WordStoreImp
```

```java
import java.util.Scanner;
/*
 * TEST CLASS FOR ADD,COUNT,REMOVE METHODS
 */
class WordTestCustom
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        WordGen.initialise(input);
        System.out.print("Enter the number of words you wish to
           generate: ");
        int n = input.nextInt();
        WordStore words = new WordStoreImp(n);
        for(int i=0; i<n; i++)
            words.add(WordGen.make());
        words.display(); //displays the words currently stored
        //add an arbitrary number of whatever word chosen
        String toAdd = "wend";
        words.add(toAdd);
        words.add(toAdd);
        words.add(toAdd);
        System.out.println(toAdd + " occurs " + words.count(toAdd) + "
           times once added."); //check how many times word chosen
           occurs in stored list once added
        String toRemove = "wend";
        System.out.println(toRemove + " occurs " + words.count
           (toRemove) + " times before removal."); //check occurences
           before removal
        //remove an arbitrary number of whatever word chosen
        words.remove(toRemove);
        words.remove(toRemove);
        words.remove(toRemove);
        words.remove(toRemove);
        System.out.println(toRemove + " has been removed.");
        System.out.println(toRemove + " occurs " + words.count
           (toRemove) + " times after removal.");//check occurences
           after removal
        words.display();
    }
}//END WordTestCustom
```

2. Design decisions and their explanations

2.1 Algorithms

The algorithms used for each method were important in increasing the implementation's efficiency. The most significant changes observed were caused by tweaks to methods `add()` and `hash()`.

Initially, `WordStoreDT` did not have a count variable. The `add()` method simply added another occurrence of the word to the linked list and the `count()` method iterated through the linked list, incrementing its own counter variable for each occurrence of the word. Testing showed that adding a counter variable count to the `WordStoreDT` class made it possible to simplify the process of adding an occurrence of a word already in the collection. To do this, the `add()` method simply calls on `isRepeated()` to check if its argument matches a word already stored. If there is a match, the instance's count variable is incremented, rather than storing an additional word in the list. The `count()` method then becomes much simpler; it must simply return the instance's count variable. These changes may have made the `add()` method slightly less efficient when dealing with smaller numbers, but greatly increased `add()`, `count()` and `remove()`'s respective efficiencies when dealing with large amounts of data.

It was also important to ensure that the algorithm used by `add()`, adds new words to the start of the linked list, rather iterating through the list until reaching the end, each time a word is added. This can be seen below.

```
wordsStored[index] = new WordStoreDT<String> (word,wordsStored[index]);
```

Figure 1 – code to add a new word to the start of the list.

In order to implement the chosen data structure, a hash function was needed to translate the word generated to an integer, which is then used to index the array. Below is an example of a simple hashing algorithm to obtain a key.

Word to be added: Mia

```
String word = “Mia”;
```

First, each letter’s ASCII code must be obtained and accessible. This can be done by declaring a char array and initialising it using String’s `toCharArray()` function.

```
char[] ascii = word.toCharArray();
```

A counter variable must be created and the char array needs to be iterated through, with each letter’s ASCII code being summed together.

```
int sum = 0;
for (int i=0; i<ascii.length; i++) {
    sum = sum+ascii[i];
}
```

The `hash()` *method must then return the remainder when* sum *is divided by the size of the array of linked lists.*

```
return sum%array.length;
```

Figure 2 – Converts each letter of a string into its ASCII equivalent, sums up the numbers and returns the remainder when divided by the size of the array.[1]

The basic hashing algorithm seen above will provide a different key for different words, which can then be used to index the array. However, there will be a large number of collisions due to its simplicity, making it inefficient. In order to increase the efficiency of my `hash()` function, it was necessary to increase the range of numbers being returned by the algorithm (see final hash function on (coding section page 3) page 5). The final hash method has time complexity of O(N).

The `remove()` method works similarly to the `add()` method. It first checks that the linked list is not null. If `ws!=null`, the method proceeds to attempt to remove an occurrence of word. To do this, it iterates through the linked list until it finds a match. Once the match is found, its count variable is decreased by one if it is greater than 1. However, if it is 1, its count variable is decreased to 0 and it is removed altogether. If no match is found in the linked list, nothing is done. The `remove()` method has time complexity of O(N).

In order to minimise the number of possible collisions, a helper method was created to expand the array. As an array’s size cannot be dynamically changed, it is necessary to create a new array (referenced by `wordsStored1`) with size double that of the current array (referenced by `wordsStored`), and manually copy `wordsStored`’s elements to `wordsStored1`. The variable wordsStored is then made to point to the same array as `wordsStored1`, thus `wordsStored` now holds the same *n* elements it previously held, but has size 2n. Due to the nature of the data structure used. The `expand()` method has time complexity of O(N).

[1] (Drumm, 2017)

2.2 Data Structure

After considering the data structures taught in the module, I decided to research data structures more suitable for handling large amounts of data (such as >1000000 entries). I chose to use a hash table data structure.

A hash table implements an associative array ADT, consisting of an array whose elements are linked lists. A linked list is, in itself, a data structure. It is made up of cells, each of which contains a piece of data and a pointer to the next cell in the list).

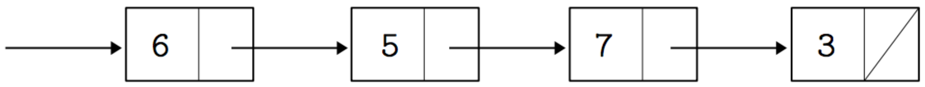

Figure 3 - An illustration of a linked list. The piece of data is stored in the leftmost piece of the cell part of the cell, and the pointer in the rightmost part of the cell.[2]

The hash table data structure uses a hash function to map keys to values (i.e. the generated word), where keys are used to index the array and access the linked list the element holds (see page 4). This is called separate chaining.

In order to implement a hash table, a Cell-type helper class called WordStoreDT was created. This helper class was made static as it does not change for any instance of `WordStoreImp`, thus using less memory. `WordStoreDT` has three attributes, each with different types; first (type String) to indicate the piece of data (the generated word in this case) in one of the linked list's cells, next (type `WordStoreDT<String>`) to indicate the pointer to the next cell, and count (type int) to keep track of how many times the same word has been added. All are initialised using WordStoreDT's constructor. It is important to note that count must be initialised to 1, as a word occurs once the first time it is added.

An import limitation linked to hash tables is the possibility of collisions. Collisions occur when the hash function produces the same hash key for two different values. Following the example from *Figure 2*, and assuming an array size of 11 (i=0 to 10) "Mia" would form the hash key 4, rounded up from 3.6 (77 + 105 + 97=279)%11, and "Sue" would also form the same hash key, (83+117+101=301)%11 = 4. This is where the linked list held by the indexed array element proves to be useful, as it can be used to store multiple elements whose values produce the same hash key, as seen below (*Figure 4*).

[2] (Tzevelekos, 2017)

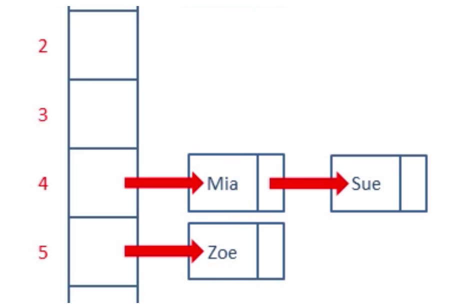

Figure 4 – Shows where Mia is added, and where Sue is then added.[3]

This reduces the efficiency of the program; it is therefore important to write a hashing algorithm that avoids collisions as much as possible. One way to do this is to ensure approximately 25% of the array remains unoccupied at all times. This is achieved by calling `expand()` to "change" the size of the array once 75% of the array has been filled, as seen in add(), as well as in Figure 5 (pictured below).

```
if (count >= 0.75*wordsStored.length) expand();
```

Figure 5 – the line of code from `add()` used to expand the array if necessary.

[3] Note: my own implementation differs from the example given, such that words with the same hash key were added to the beginning of the linked list rather than the end.

3. Experimental Results

For each of the tests (membership, add and remove), two sets of data have been collected. The first set involves keeping the collection size (number of words initially generated) constant, while varying K, the number of words being tested. The second set does the opposite; the collection size varies while the number of words being tested remains constant.

The tables show the raw data collected, which is then used to calculate the time taken to perform a single operation.

K = 50000, 600000, 700000
N = 10000000

3.1 Membership Test

These tests were conducted using the provided java file WordTest2.java

Table 1 – time to test membership of K words in collection of size N where N=10000000

	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Average**
K /words	**Time /ms**	**Time /ms**	**Time /ms**	**Time /ms**	**Time /ms**
500000	70.0	68.0	64.0	72.0	68.5
600000	78.0	79.0	78.0	73.0	77.0
700000	94.0	94.0	85.0	85.0	89.5

When K = 500000
Average time taken to test membership of a single word to collection N = $1.37\text{x}10^{-4}$ ms
When K = 600000
Average time taken to test membership of a single word to collection N = $1.283333333\text{x}10^{-4}$ ms
When K = 700000
Average time taken to test membership of a single word to collection N = $1.282142857\text{x}10^{-4}$ ms
Average of the above times for collection of size N = $1.310634921\text{x}10^{-4}$ ms

Table 2 – time to test membership of 200000 words in collection of size N where N=1000000, 5000000, 10000000.

	Time /ms				
Collection N /words	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Average**
1000000	23.00	24.00	22.00	27.00	24.50
5000000	31.00	32.00	29.00	28.00	28.75
10000000	28.00	32.00	29.00	31.00	30.00

When N=1,000,000
Average time taken to test membership of a single word to collection N = $1.225\text{x}10^{-4}$ ms

When N=5,000,000
Average time taken to test membership of a single word to collection N = 1.43753x10^{-4} ms
When N=10,000,000
Average time taken to test membership of a single word to collection N = 1.5 x10^{-4} ms
Average of the above times for collection of size N = 1.3875x10^{-4} ms

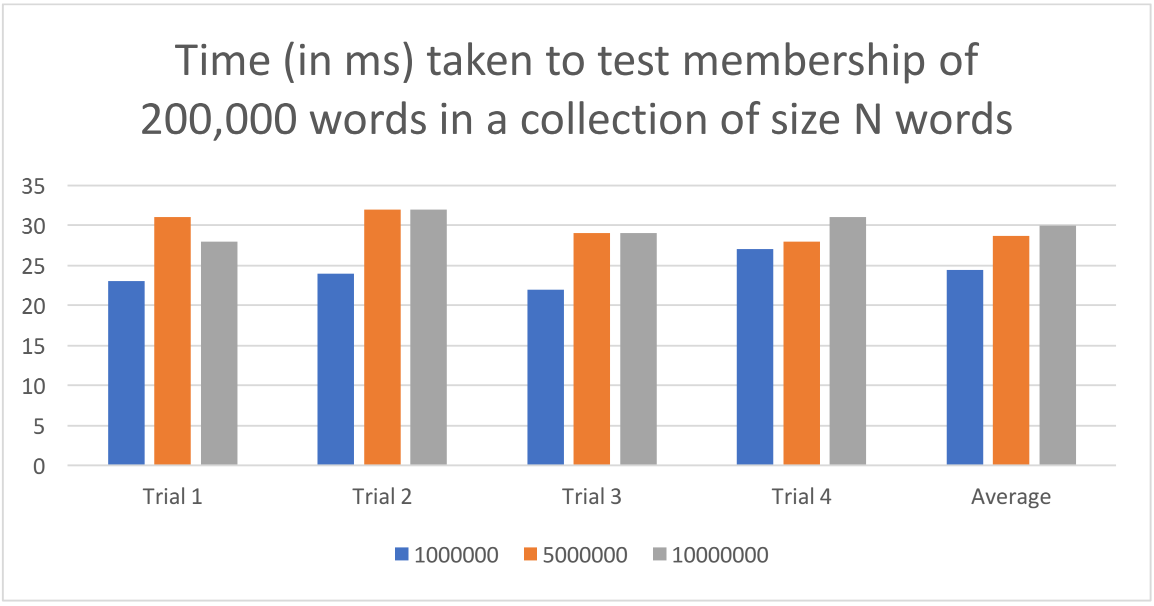

Chart 1

3.2 Add Test

These tests were conducted using the provided java file WordTest3.java, with seed = 1.

Table 3 – time to add K words to collection of size N where N=10000000

	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Average**
K /words	**Time /ms**	**Time /ms**	**Time /ms**	**Time /ms**	**Time /ms**
500000	72.0	86.0	69.0	73.0	75.0
600000	90.0	92.0	84.0	84.0	87.5
700000	106.0	97.0	94.0	103.0	100.0

When K = 500000,
Average time taken to add a single word to collection N = 1.5x10^{-4} ms
When K = 600000
Average time taken to add a single word to collection N = 1.458333333x10^{-4} ms
When K = 700000
Average time taken to add a single word to collection N = 1.428571429x10^{-4} ms
Average of the above times for collection of size N = 1.462301587x10^{-4} ms

Table 4 – time to add 200000 words to collection of size N where N=1000000, 5000000, 10000000.

	Time /ms				
Collection N /words	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Average**
1000000	24.00	24.00	25.00	25.00	24.50
5000000	28.00	29.00	28.00	30.00	28.75
10000000	36.00	33.00	33.00	35.00	34.25

When N=1,000,000
Average time taken to add a single word to collection N = $1.225\text{x}10^{-4}$ ms
When N=5,000,000
Average time taken to add a single word to collection N = $1.4375\text{x}10^{-4}$ ms
When N=10,000,000
Average time taken to add a single word to collection N = $1.7125\text{x}10^{-4}$ ms
Average of the above times for collection of size N = $1.458333333\text{x}10^{-4}$ ms

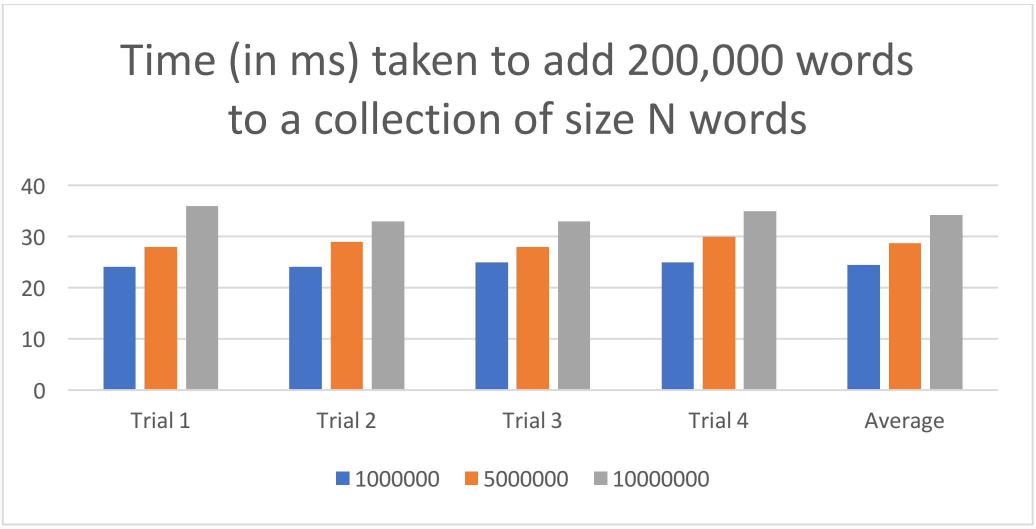

Chart 2

3.3 Remove Test

These tests were conducted using the provided java file WordTest4.java, with seed = 1.

Table 5 - time to remove K words from collection of size N where N=10,000,000

	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Average**
K /words	**Time /ms**	**Time /ms**	**Time /ms**	**Time /ms**	**Time /ms**
500000	79.00	73.00	81.00	73.00	76.50
600000	94.00	95.00	102.00	91.00	95.50
700000	111.00	105.00	115.00	108.00	109.75

When K = 500000,
Average time taken to remove a single word from collection N = 1.53×10^{-4} ms
When K = 600000
Average time taken to remove a single word from collection N = $1.591666667 \times 10^{-4}$ ms
When K = 700000
Average time taken to remove a single word from collection N = $1.567857143 \times 10^{-4}$ ms
Average of the above times for collection of size N = $1.563174603 \times 10^{-4}$ ms

Keeping K constant: 200,000

Table 6 – time to remove 200000 words from collection of size N where N=1000000, 5000000, 10000000.

	Time /ms				
Collection N /words	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Average**
1000000	26.00	24.00	27.00	24.00	25.50
5000000	32.00	33.00	29.00	32.00	31.50
10000000	34.00	36.00	33.00	34.00	34.25

When N = 1,000,000,
Average time taken to remove a single word from collection N = 1.275×10^{-4} ms
When N=5,000,000
Average time taken to remove a single word from collection N = 1.575×10^{-4} ms
When N=10,000,000
Average time taken to remove a single word from collection N = 1.7125×10^{-4} ms
Average of the above times for collection of size N = $1.520833333 \times 10^{-4}$ ms

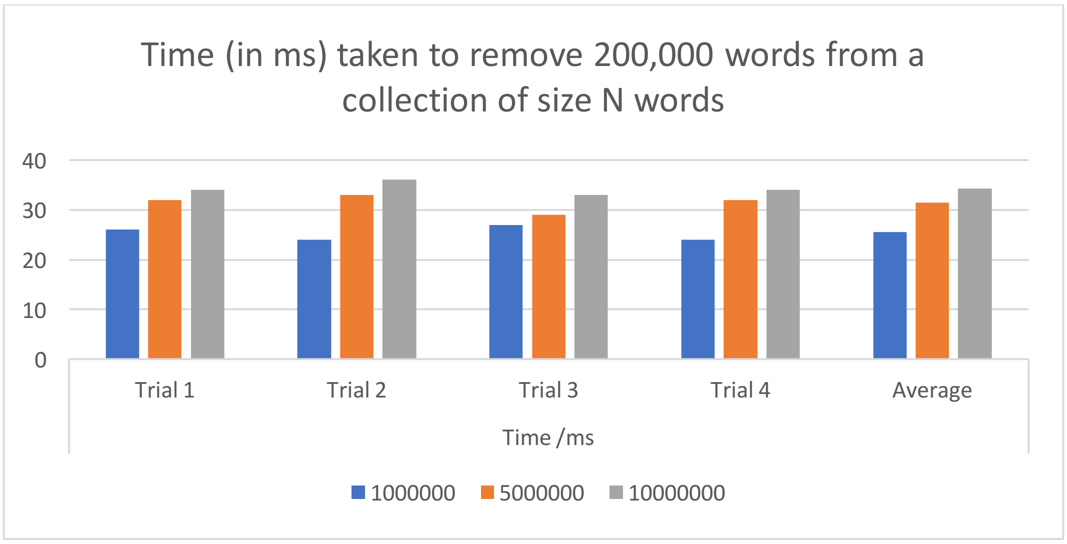

Chart 3

Works Cited

Drumm, K., 2017. *Hash Tables and Hash Functions.* [Online]
Available at: https://www.youtube.com/watch?v=KyUTuwz_b7Q
[Accessed 08 12 2017].
Tzevelekos, N., 2017. *Lecture 8 - Linked Lists,* s.l.: s.n.