Author: Hussein Tejan

Dr Matthew Huntbach

ECS510 Algorithms & Data Structures

Wednesday, 3 December 2014

Hash Table

Prior to deciding the data structure I planned to use for my mini-project, I had researched thoroughly into the existing algorithms structured in the likes of Binary Trees, Arrays, LinkedList and Hash Tables that I could implement and make use of in order to build an efficient data structure that could store the words generated from the WordGen class, however I noticed that I was spoilt for choice. Furthermore, I had considered which data structure would be the most efficient and to my surprise it turned out to be HashTables for a number of reasons.

1) The main operations:Find, Insert and Remove time complexity on average are calculated to be O(1) as they take on average constant time, whereas a Linked List corresponds to linear time, further to my discovery; Binary Trees have a time complexity for the above operations on average calculated to be O(log n).

2) Unlike how an array functions, it is not limited to what type of keys it can store.

The following methods play a vital role within my Hash Table i.e: **Add**, **Remove** and **Count**.

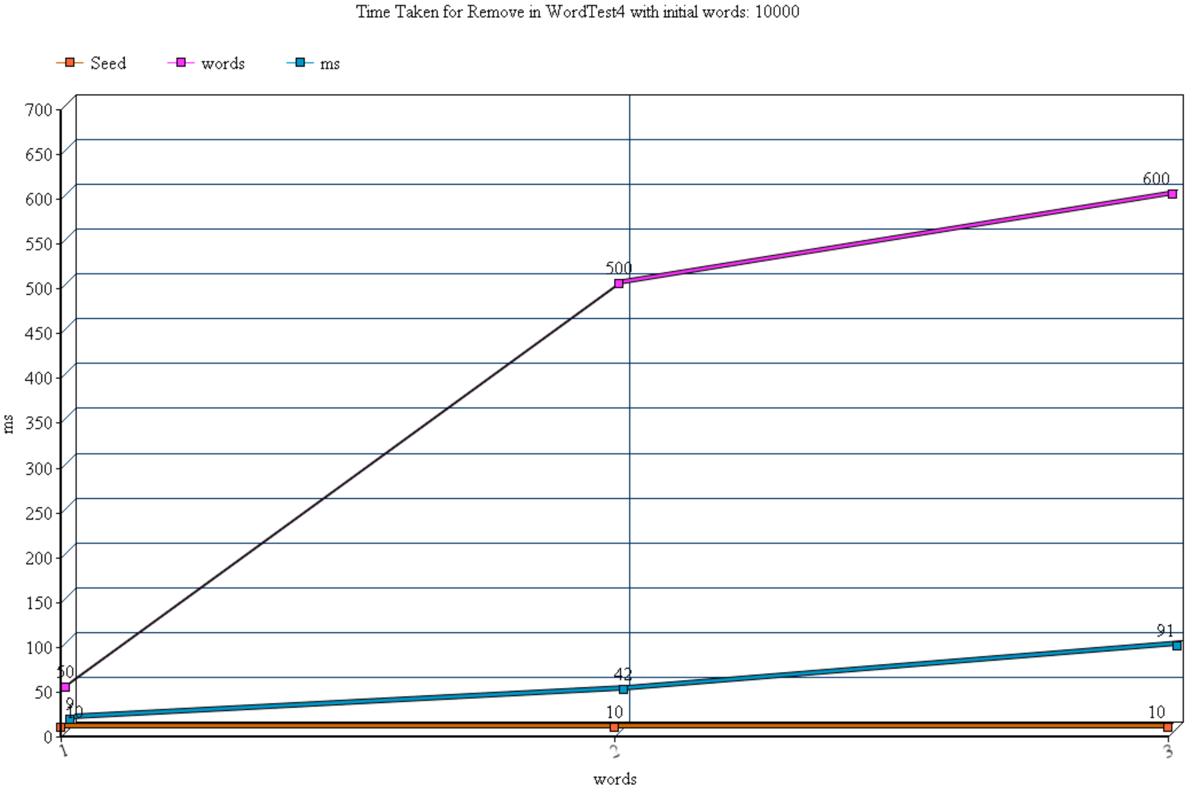
Add: whereby the Hash Table adds words from the WordGen class based on whether words are contained in its Data-Structure or not in addition to also, checks for duplicates within the Hash-Table and increments its counter if the word already exist within a certain key(position) from the first key until there is no word found within the key. Furthermore, if a word already exist in a key(position) the Hash-Table will develop chaining whereby the key stores one value which is linked to the next value stored within its position.

Remove: whereby the Hash Table removes words stored in the Data-Structure generated from the WordGen class based on whether words are already existing in its Data-Structure or not in addition to also, checks for duplicates within the Hash-Table and decrements its counter by **1** if the word already exist within a certain key(position) from the first key until there is no word found within the key. Furthermore, if a word already exist in a key(position) the Hash-Table once, the word would no longer exist in that key and instead the pointer will refer to its next key, thus the word that had originally come before would be loss in memory.

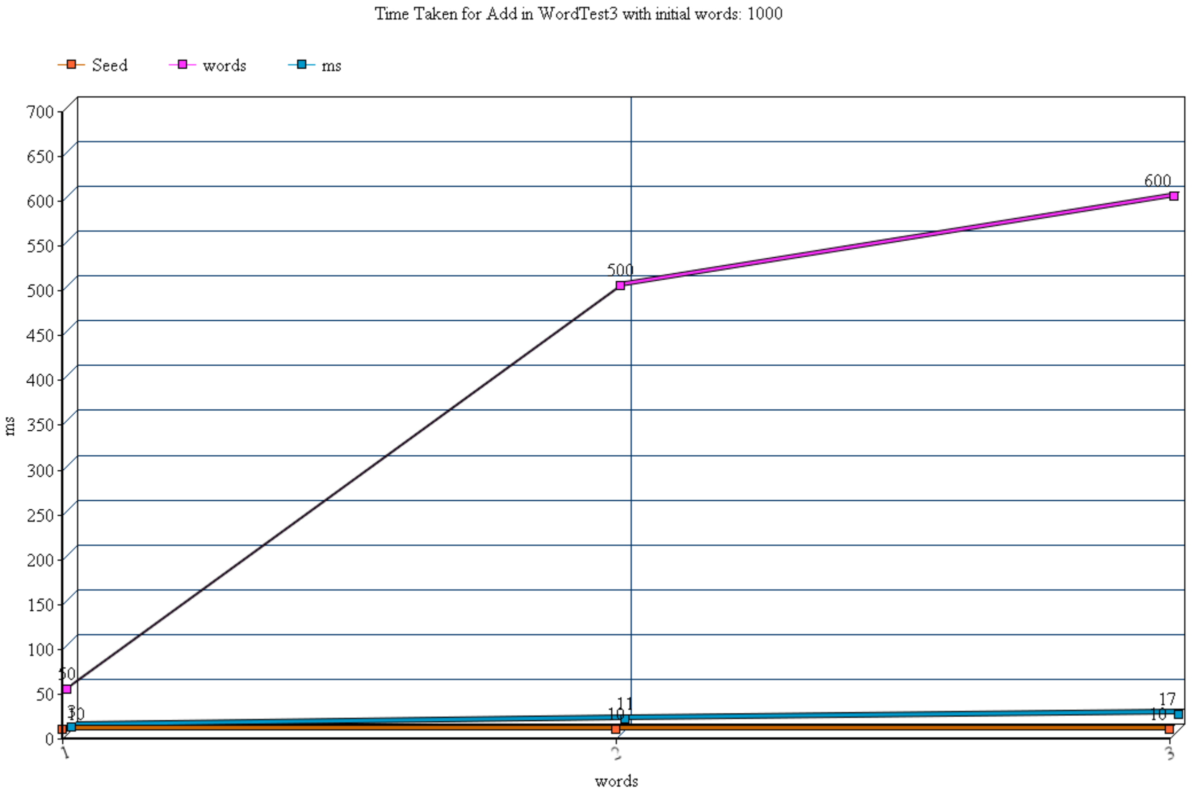
Count: whereby the Hash Table checks whether the first element in the LinkList contains a word and returns how many words are stored in its Data-Structure from the first key to its very last so long as it does not reach a key that is null, otherwise 0 is returned to symbolise no count of words existing within the Hash-Table.

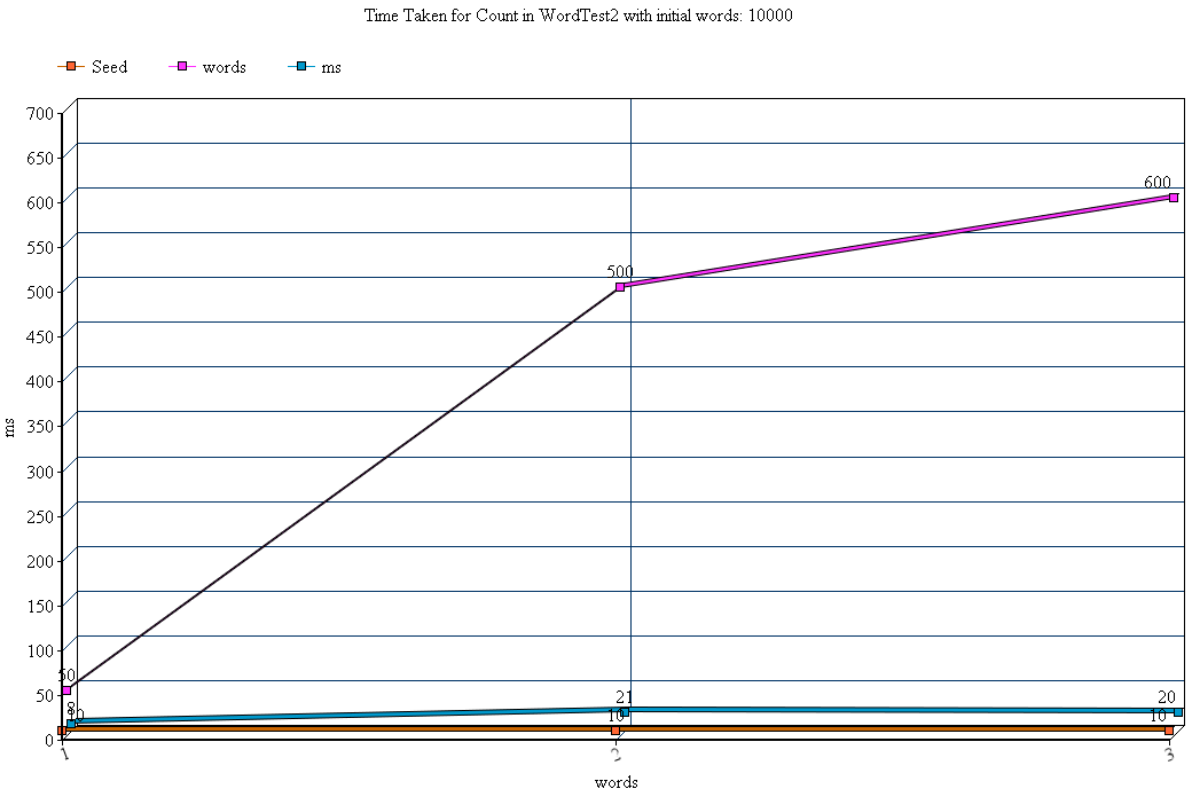
Below are the results from me testing my code on the following methods remove, add, count and comparing them using a graph to represent each significant data .

As you can see based on the graph below, I can come to a conclusion that the more words there are within the Hash-table, the less time the data structure will be quick enough to **remove** the words within its contents.
However, the less words there are within the Hash-table, the more quicker it would be to remove the words within its contents.



As you can see based on the graph below, I can come to a conclusion that the more words there are within the Hash-table, the more time the data structure will take to **add** the words to its contents.
However, the less words there are within the Hash-table, the less time it would take to add the words within its contents.





As you can see based on the graph above, I can come to a conclusion that the more words there are within the Hash-table, the more time the data structure will take to **count** the words within its contents.

However, the less words there are within the Hash-table, the less time it would be to count the words within its contents.

Code:

```
public class WordStoreImp implements WordStore {


    int arraySize = 0;

    Chromosome[] dna;


    WordStoreImp(int linklistSize) {

        arraySize = linklistSize;

        dna = new Chromosome[arraySize];


    }


    protected class Chromosome {


        protected String first = null;

        protected Chromosome next = null;

        protected int counter = 0;


        Chromosome(String h, Chromosome t) {

            first = h;

            next = t;

        }
```

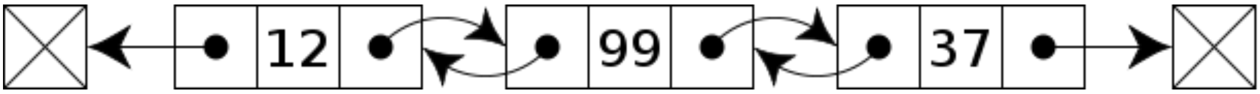
```
}//END Chromosome


public int hashFunction1(String word) {


    int key = word.length() % arraySize;

    if (key < 0) {

        key = -1 * key;

    }


    return key;


}


public void add(String word) {


    if (dna[hashFunction1(word)] == null) {


        dna[hashFunction1(word)] = new Chromosome(word, null);

        dna[hashFunction1(word)].counter = 1;


    } else {


        if (dna[hashFunction1(word)].first.equalsIgnoreCase(word)) {

            dna[hashFunction1(word)].counter = dna[hashFunction1(word)].counter + 1;

            return;
        }

        Chromosome newDna = dna[hashFunction1(word)];
        while (newDna.next != null) {
            if (newDna.next.first.equals(word)) {
                newDna.next.counter = newDna.next.counter + 1;
                return; //leave cell alone
            }//END if statement
            newDna = newDna.next;
        }
        newDna.next = new Chromosome(word, null);
        newDna.next.counter = 1;
    }
}

public void checkword(String word) {
    if (dna[hashFunction1(word)].equals(word)) {
        dna[hashFunction1(word)].counter = dna[hashFunction1(word)].counter + 1;
    }
}

public int count(String word) {

    Chromosome ptr = dna[hashFunction1(word)];
    if (ptr.first.equals(word)) {
```

```
        return ptr.counter;
    }//END if
    else {
        for (; ptr.next!= null; ptr = ptr.next) {
            if (ptr.next.first.equals(word)) {
                return ptr.next.counter;
            }//END if
        }//END for loop
        return 0;
    }//END else
}//END count

public static void printCount(String word, WordStoreImp cc) {
    System.out.println(cc.count(word));
}

public void remove(String word) {

    Chromosome ptr2 = dna[hashFunction1(word)];
    if (ptr2 != null) {
        if (ptr2.first.equalsIgnoreCase(word)) {
            if (ptr2.counter != 1) {
                ptr2.counter = ptr2.counter - 1;
            }//END if statement
            else {
```

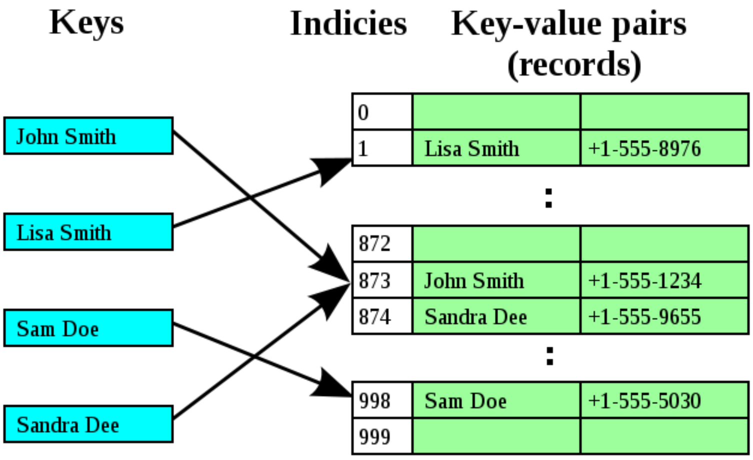
```
            dna[hashFunction1(word)] = dna[hashFunction1(word)].next;
        }//END else
    }


    else {


    while (ptr2.next!= null) {
        if (ptr2.next.first.equalsIgnoreCase(word)) {
            System.out.println("We get in ");
            if (ptr2.next.counter > 1) {
                ptr2.next.counter = ptr2.next.counter - 1;
                return;
            }//END if statement
            else {
                System.out.println("We should remove ");
                ptr2.next = ptr2.next.next;
                return;
            }//END else statement
        }//END if statement
        ptr2 = ptr2.next;
    }//END while loop
    System.out.println("THE " + word + " DOESN'T EXIST!");
}//END else statement
```

```
        }//END outer if statement



    }//END remove


    public int getKey(String word) {
        return hashFunction1(word);
    }


    public String getValue(String word) {
        Chromosome ptr2 = dna[hashFunction1(word)];
        if (dna[hashFunction1(word)].equals(word)) {
            return word;
        }//END if statement
        else {
            while (ptr2 != null) {

                if (ptr2.first.equalsIgnoreCase(word)) {
                    return word;
                }//END if statement
                else {
                    ptr2 = ptr2.next;
                }
            }//END while loop
        }//END else statement
        return " ";
    }//END getValue

    public static void printSomething(WordStoreImp cell) {
        for (Chromosome cc : cell.dna) {
            if (cc == null) {

            } else {
                System.out.println(" The key " + cell.getKey(cc.first) + "  value :" + cc.first + "
and the occurences :" + cc.counter);

                Chromosome crap = cc.next;

                if (crap != null) {
                    System.out.println("The chain of key" + cell.hashFunction1(crap.first));

                    while (crap != null) {
                        System.out.println("  value :" + crap.first + "  and the occurences :" +
crap.counter);
                        crap = crap.next;
                    }//END while loop

                }
            }

        }//END for loop

    }

}//END WordStoreImp
```

Next time, I would be fascinated to explore how a program like this could function efficiently with a doubly LinkedList rather than a single to take advantage of having access to the front and the back of the LinkedList.



In addition, I would also be interested in the future to tackle collisions with Open/Closed Dressing as opposed to Chaining. Whereby, I can place new words I attempted to store in one key in the next empty key.


Keys
Indicies
Key-value pairs (records)
John Smith
Lisa Smith
Sam Doe
Sandra Dee
0
1 Lisa Smith +1-555-8976
872
873 John Smith +1-555-1234
874 Sandra Dee +1-555-9655
998 Sam Doe +1-555-5030
999