

Algorithms and Data Structures in an Object Oriented Framework

Mini-Project Report

By Elif Sebnem Cudi

Aim of the Project

The aim of the project is to store and manipulate large quantities of the `string` data type in an efficient manner.

The data structure will be written in a way that allows the same `string` value to be added more than once. The data structure should allow for both the addition and deletion of strings, and for searching for the number of times the `string` is found in the collection.

Choice of Data Structure

The data structure chosen for this project is hash tables, with the main reason being the fact that in this project the data does not have to be ordered. The project only requires basic set of operations of adding, deleting, and searching, which the hash tables excel at with their $O(1)$ time complexity. Although it is less memory efficient compared to other data structures, hash tables retrieve data fast, and this was identified as the more important factor for this project.

More on Hash Tables and Design Choices

Hash tables map unique keys to their associated values. These are usually pairs such as: ID number being the unique keys, and personal information such as telephone number, email address, etc. being the associated values. The hash table uses a “hash function” to convert the key to an array index, where the associated value to the key will be inserted.

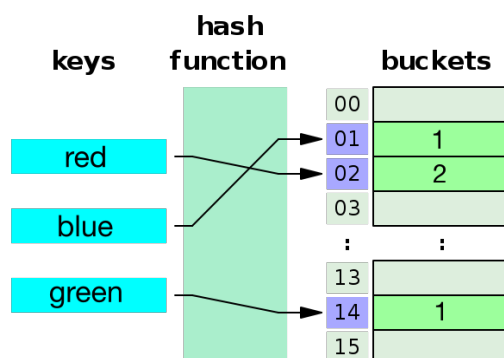


Figure 1 An example of a hash table where the strings “red”, “blue”, and “green” were given the index values “02”, “01” and “14”, and were added to the hash table twice, once and once respectively.

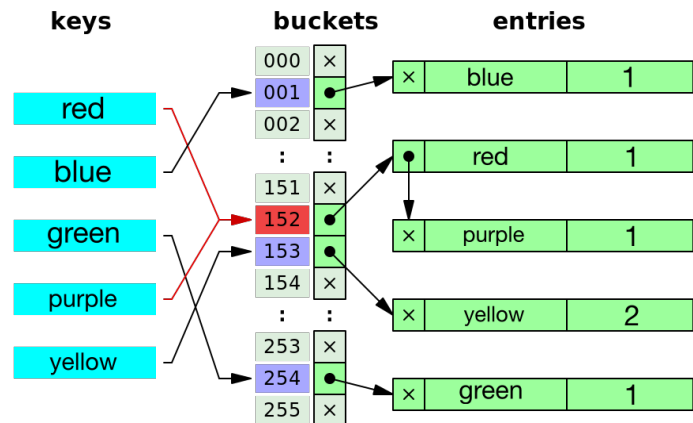
The figure above is a basic example of a hash table, where the keys and values inside the buckets mirror the way they are used in this project: the keys are the `strings` to be stored, and the values are the amount of times the `strings` were added to the hash table. The design choice behind the latter is explained in more detail under the “Handling duplicate words” heading.

However, **Figure 1** assumes that all strings have a unique hash function outcome, which they do not. Two different keys can produce the same hash value, which is called a **collision**.

To deal with collisions, keys are also stored in the buckets to identify the ones that generate the same hash. A well written function for the hash function to produce as many unique hashes as possible is also an option. Once those precautions are taken, there are two strategies to choose from: open addressing or chaining.

The project utilises the chaining strategy, where: each bucket of the hash table contains a linked list, which stores key-value pairs with the same hash. When a collision happens, the key-value pair is stored in the bucket as a linked list along with the previous entries that were already in the bucket.

Figure 2 A visual where the words “red” and “purple” have the same hash value in a chaining hash table. The result is them being put in the same bucket where “red” is the first item of a linked list, and “purple” is then added as the second element of the linked list. It is a basic representation of how the hash table of the project operates.



Open Addressing vs. Chaining

Open addressing is a strategy where the buckets have a single key-value pairs stored inside, and if a collision happens, the collided key-value pair is allocated to the next empty bucket.

Figure 3 compares the two strategies, however, the sole reason chaining was picked for this project over open addressing is that with open addressing, once a key-value pair is deleted, it is replaced with a “DELETED” entry. Since the project works with large quantities of data, over time the hash table would be full of “DELETED” entries, which is undesirable because it is a waste of memory.

	Chaining	Open addressing
Collision resolution	Using external data structure	Using hash table itself
Memory waste	Pointer size overhead per entry (storing list heads in the table)	No overhead ¹
Performance dependence on table's load factor	Directly proportional	Proportional to $(loadFactor) / (1 - loadFactor)$
Allow to store more items, than hash table size	Yes	No. Moreover, it's recommended to keep table's load factor below 0.7
Hash function requirements	Uniform distribution	Uniform distribution, should avoid clustering
Handle removals	Removals are ok	Removals clog the hash table with "DELETED" entries
Implementation	Simple	Correct implementation of open addressing based hash table is quite tricky

Figure 3 Comparison between chaining and open addressing

Prime numbers for the hash function

The hash produced by the hash function has to be as unique as possible to make sure there is no meaningless bucket space wasted, and to make sure the lowest amount of collisions occur. It happens that the product of a prime with any other number has the best chance of being unique due to the fact that a prime was used to compose it in the first place (computinglife, 2008).

Threshold of 90%

The load factor of an array is the ratio between the number of stored items and array's size. Although a hash table with a chaining strategy can work with load factors bigger than 1, it is not ideal as the higher the load factor, the less likely the time complexity of the hash table will be $O(1)$. Therefore, the threshold of 90% was set to compromise both on time complexity and memory efficiency.

Handling duplicate words

The “value” of the key-value pair has been decided to be the count of how many times a `string` is added to the hash table, to save memory and increase efficiency:

- If the hash table was to store each key-value pair as a unique entry in the buckets, it would waste memory space by simply creating more unique entries.
- This method also increases efficiency all around because now all methods only have to find one key-value pair to retrieve and modify the information to do with it's value. For example, the method `countOccurrences()` in the class `HashTable` will now only have to match a single key to get information on how many times the `string` was added to the hash table.

Space efficiency

If the hash table of the project was to only store a small amount of `strings`, there would be no point of having a huge hash table. However, the project requires adding and deleting of `strings`, therefore a resizing method was incorporated. This is a dynamic resizing where the resizing depends on the amount of unique entries within the table.

This means that the hash table will only make space for the desired amount of entries, instead of taking up a lot of space all at once.

Testing

For testing, the class `wordGen` will be used to generate words with lower case alphabetic characters that resemble the English language, and they will be passed and stored in the hash table through the `wordStoreImp` class.

The java files `wordTest2.java`, `wordTest3.java`, and `wordStore4.java` are designed to test the efficiency of the operations `count`, `add` and `remove` of the `wordStoreImp` class respectively. The code in these programs generates a collection of words, then generates a second set of words stored in an array and applies the operation on the collection with each word from the second set. It measures and displays the time taken to apply the operation repeatedly (Huntbach, M., 2017).

The java file `wordTest5.java` however, is designed to test the accuracy and it's ability to store duplicate strings without creating a completely new key-value pair.

WordTest2.java

This test will check for the efficiency of the `count` operation of the `wordStoreImp` class. Three seeds will be checked, each will be run 3 times, and the average of each seed-run will be plotted on a graph. The graphs will be **Figure 4** and **Figure 5**.

Initial amount of words generated: 1×10^3 ; 1×10^4 ; 1×10^5 ; 5×10^5 ; 1×10^6 ; 5×10^6

Amount of words generated to test for occurrence: 1×10^4 ; 1×10^5

Unit of time: Milliseconds (ms)

WordTest3.java

This test will check for the efficiency of the `add` operation of the `wordStoreImp` class. Three seeds will be checked, each will be run 3 times, and the average of each seed-run will be plotted on a graph. The graphs will be **Figure 6** and **Figure 7**.

Initial amount of words generated: 1×10^3 ; 1×10^4 ; 1×10^5 ; 5×10^5 ; 1×10^6 ; 5×10^6

Amount of words generated to add: 1×10^4 ; 1×10^5

Unit of time: Milliseconds (ms)

WordTest4.java

This test will check for the efficiency of the `delete` operation of the `wordStoreImp` class. Three seeds will be checked, each will be run 3 times, and the average of each seed-run will be plotted on a graph. The graphs will be **Figure 8** and **Figure 9**.

Initial amount of words generated: 1×10^3 ; 1×10^4 ; 1×10^5 ; 5×10^5 ; 1×10^6 ; 5×10^6

Amount of words generated to delete: 1×10^4 ; 1×10^5

Unit of time: Milliseconds (ms)

WordTest5.java

For this test, `wordTest5.java` will be ran on Netbeans and a screenshot of the output console will be taken. This screenshot will be **Figure 10**.

Written Code

WordStoreImp.java

```
/** -----
 * Author: Elif Sebnem Cudi
 * Last Updated: 10/12/2017
 *
 * This class implements the WordStore interface and
 * acts as a bridge between the test code and the hash table.
 * ----- */
class WordStoreImp implements WordStore {

    HashTable h = new HashTable(); // Creates a hash table specific to the WordStoreImp object

    /**
     * Passes the string to be stored to the hash table.
     */
    public void add(String word) {
        h.put(word,1);
    }

    /**
     * Passes the string to be counted to the hash table
     * and returns the amount of occurrences found.
     */
    public int count(String word) {
        return h.countOccurrences(word);
    }

    /**
     * Passes the string to be removed to the hash table.
     */
    public void remove(String word) {
        h.remove(word);
    }

    /**
     * Method placed by the author to be used during testing.
     * Prints out information about the hash table.
     * For this class to correctly implement it's interface,
     * this method has also been created in WordStore.
     */
    public void print() {
        h.print();
    }
}
```

HashTable.java

```
/** -----
 * Author: Elif Sebnem Cudi
 * Last Updated: 10/12/2017
 *
 * Main class for the hash table.
 * It is a chaining hash table: the table is an array of linked lists
 * containing key-value pairs with the same hash. The information stored is
 * a string, and the amount of times it was added to the hash table.
 * ----- */
public class HashTable {

    private final int INITIAL_TABLE_SIZE = 128;
    private float threshold = 0.9f; // The load factor
    private int maxSize = 115; // Initial maximum amount of entries before the table is set to expand
    private int size = 0; // Count for the number of entries

    LinkedHashEntry[] table; // Initialising the hash table

    /**
     * Class constructor for the initial hash table with the size 128.
     * Sets every bucket to null.
     */
    HashTable() {
        table = new LinkedHashEntry[INITIAL_TABLE_SIZE];
        for (int i = 0; i < INITIAL_TABLE_SIZE; i++) {
            table[i] = null;
        }
    }

    /**
     * This method passes a string into a hash function. The resulting hash is
     * used to determine the index in the hash table which the initial string
     * will be placed in.
     *
     * If the bucket is null, it will make the string the first item of a
     * linked list within the bucket.
     * If there already exists a linked list within the bucket, it
     * will traverse the linked list.
     * If the string is found to already exist within the linked list,
     * it will increment it's count by 1, else, it will add
     * the string to the end of the list with count 1.
     *
     * With every addition of a new word, size is also incremented. If
     * the size surpasses the threshold, a new hash table
     * is created twice the size of the previous one and the contents
     * of the previous hash table are copied over by the resize() method.
     */
    public void put(String key, int count) {

        int hash = hashCode(key); // Retrieves the hash value

        if (table[hash] == null) {
            table[hash] = new LinkedHashEntry(key, count);
            size++;
        } else {
            LinkedHashEntry entry = table[hash];
            while (entry.getNext() != null && !entry.getKey().equals(key)) {
                entry = entry.getNext();
            }
        }
    }
}
```

```

    }
    if (entry.getKey().equals(key)) {
        entry.increaseCount();
    } else {
        entry.setNext(new LinkedHashMapEntry(key, count));
        size++;
    }
}
if (size >= maxSize) {
    resize();
}
}

/**
 * This method expands the hash table by creating a new
 * hash table double its previous size, and copying the contents of the
 * table over to the new one. However, each word is placed at a new index
 * as a result of the hash function returning an integer dependent
 * on the new table size.
 */
private void resize() {

    size = 0; // Variable set back to 0 to recount the amount of unique entries
    int tableSize = 2 * table.length;
    maxSize = (int) (tableSize * threshold); // Increases the maximum size to fit the 90% threshold
    LinkedHashMapEntry[] oldTable = table;
    table = new LinkedHashMapEntry[tableSize];

    for (int i = 0; i < tableSize; i++) {
        table[i] = null;
    }
    for (int i = 0; i < oldTable.length; i++) {
        if (oldTable[i] != null) {
            LinkedHashMapEntry entry = oldTable[i];
            while (entry != null) {
                put(entry.getKey(), entry.getCount());
                entry = entry.getNext();
            }
        }
    }
}

/**
 * This method removes entries from the hash table.
 *
 * Taking a single string as it's argument, the method finds the string's
 * hash index, and, if the index is not null in the first place, searches
 * for the string within the linked list. It will remove the string
 * by setting the entry to be the next entry.
 *
 * It will also decrement the size variable to suit the amount of unique
 * entries within the hash table.
 */
public void remove(String key) {

    int hash = hashCode(key); // Retrieves the hash value

    if (table[hash] != null) {
        LinkedHashMapEntry prevEntry = null;
        LinkedHashMapEntry entry = table[hash];

```



```

        while (entry.getNext() != null && !entry.getKey().equals(key)) {
            prevEntry = entry;
            entry = entry.getNext();
        }

        if (!entry.getKey().equals(key)) {
            if (prevEntry == null) {
                table[hash] = entry.getNext();
            } else {
                prevEntry.setNext(entry.getNext());
            }
            size--;
        }
    }
}

/**
 * This method is the hash function. It finds what position in the hash
 * table the method put() should put the string based on the number
 * generated by the computation.
 *
 * In this case, the function runs through all the characters of it's
 * String argument.
 *
 * For the first multiplication, it multiplies the initial hash value 7
 * with 31, and adds the ASCII value of the first character of the
 * String argument to the result. It then multiplies this result with 31
 * and adds the ASCII value of each character of it's String argument until
 * it reaches the end of the String.
 *
 * If finally divides this number by the size of the table, returning the
 * remainder of the division as hash value: the index of the bucket
 * which the key-value pair will be placed in.
 *
 * This function will always produce a number between 0 and the
 * table size.
 */
public int hashCode(String key) {

    int hash = 7;

    for (int i = 0; i < key.length(); i++) {
        hash = hash * 31 + (int) key.charAt(i);
    }
    return absoluteValue(hash % table.length);
}

/**
 * This method returns the absolute value of it's integer argument.
 */
int absoluteValue(int n) {
    if (n < 0) {
        return n * (-1);
    } else {
        return n;
    }
}

/**

```

```

* This method returns the number of occurrences of a particular
* string within the hash table.
*
* It will first generate the hash index of the toMatch string and
* try to find it within the linked list of that particular bucket.
* If the key is not found, it will return 0, if the key is found,
* it will return the information saved within the count variable on how
* many times the word was generated.
*/
int countOccurrences(String toMatch) {

    int hash = hashCode(toMatch);

    if (table[hash] == null) {
        return 0;
    } else {
        LinkedHashMap entry = table[hash];

        while (entry != null) {
            if (entry.getKey().equals(toMatch)) {
                return entry.getCount();
            }
            entry = entry.getNext();
        }

        return 0;
    }
}

/**
* This method was written by the author to experiment how the hash table
* stores information.
*
* It goes through every entry within every bucket and can print out:
* every key, every information on the amount of times the string was
* added, the hash index, the total strings stored including duplicates,
* and the table size at the point of method's execution.
*/
void print() {
    int sumOfEntries = 0;
    for (int i = 0; i < table.length; i++) {
        LinkedHashMap entry = table[i];
        System.out.println("Bucket: " + i + ":");

        while (entry != null) {
            System.out.println("> Value: " + entry.getKey() + " | Amount generated: " +
entry.getCount());
            sumOfEntries = sumOfEntries + entry.getCount();
            entry = entry.getNext();
        }
        System.out.println();
    }
    System.out.println("Total words saved: " + sumOfEntries);
    System.out.println("Table size: " + table.length);
}
}

```

LinkedHashEntry.java

```
/** -----  
 * Author: Elif Sebnem Cudi  
 * Last Updated: 10/12/2017  
 *  
 * This is the class for the linked lists that make up the hash table.  
 *  
 * It stores a string, and information on how many times  
 * the string was added to the hash table.  
 * ----- */
```

```
public class LinkedHashEntry {  
  
    private String key;        // Variable for the string stored  
    private int count;        // Variable for the amount stored  
    private LinkedHashEntry next;  
  
    LinkedHashEntry(String key, int count) {  
        this.key = key;  
        this.count = count;  
        this.next = null;  
    }  
  
    public String getKey() {  
        return key;  
    }  
  
    public int getCount() {  
        return count;  
    }  
  
    public void increaseCount() {  
        count++;  
    }  
  
    public void decreaseCount(){  
        count--;  
    }  
  
    public LinkedHashEntry getNext() {  
        return next;  
    }  
  
    public void setNext(LinkedHashEntry next) {  
        this.next = next;  
    }  
}
```

WordTest5.java

```
/**
 * Author: Elif Sebnem Cudi
 * Last Updated: 10/12/2017
 *
 * This program tests the accuracy of the hash table and it's ability to store
 * duplicate strings without creating a completely new key-value pair.
 *
 * It adds strings that represent the days of the week to the hash table, with
 * different number of strings added for each day. It then prints out how many
 * should have been added to the hash table, and then utilises the count()
 * method of the WordStoreImp class to find the actual counts of the strings.
 *
 * It will then print the total number of strings that should have been saved
 * versus the total number of strings that were actually saved.
 */
public class WordTest5 {

    public static void main(String[] args) {

        int totalSaved = 0;
        String[] daysOfTheWeek = new String[]{"Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday", "Sunday"};;
        WordStore words = new WordStoreImp();

        for (int i = 0; i < 3; i++) {
            words.add(daysOfTheWeek[0]);
            totalSaved++;
            words.add(daysOfTheWeek[5]);
            totalSaved++;
        }

        for (int i = 0; i < 5; i++) {
            words.add(daysOfTheWeek[2]);
            totalSaved++;
        }

        for (int i = 0; i < 10; i++) {
            words.add(daysOfTheWeek[3]);
            totalSaved++;
        }

        for (int i = 0; i < 2; i++) {
            words.add(daysOfTheWeek[1]);
            totalSaved++;
        }

        words.add(daysOfTheWeek[4]);
        totalSaved++;

        System.out.println("Monday should be saved 3 times.");
        System.out.println("Tuesday should be saved 2 times.");
        System.out.println("Wednesday should be saved 5 times.");
        System.out.println("Thursday should be saved 10 times.");
        System.out.println("Friday should be saved once.");
        System.out.println("Saturday should be save 3 times.");
        System.out.println("Sunday should be not have been saved.");

        int count;
```

```

for (int i = 0; i < daysOfTheWeek.length; i++) {
    count = words.count(daysOfTheWeek[i]);
    System.out.print("\"" + daysOfTheWeek[i] + "\" ");
    if (count == 0) {
        System.out.println("NOT saved");
    } else if (count == 1) {
        System.out.println("saved once");
    } else {
        System.out.println("saved " + count + " times ");
    }
}

System.out.println("\nThe total amount of words saved should be: " + totalSaved);
words.print();
}
}

```

Test Results

WordTest2.java

Figure 4: Testing against 10,000 words

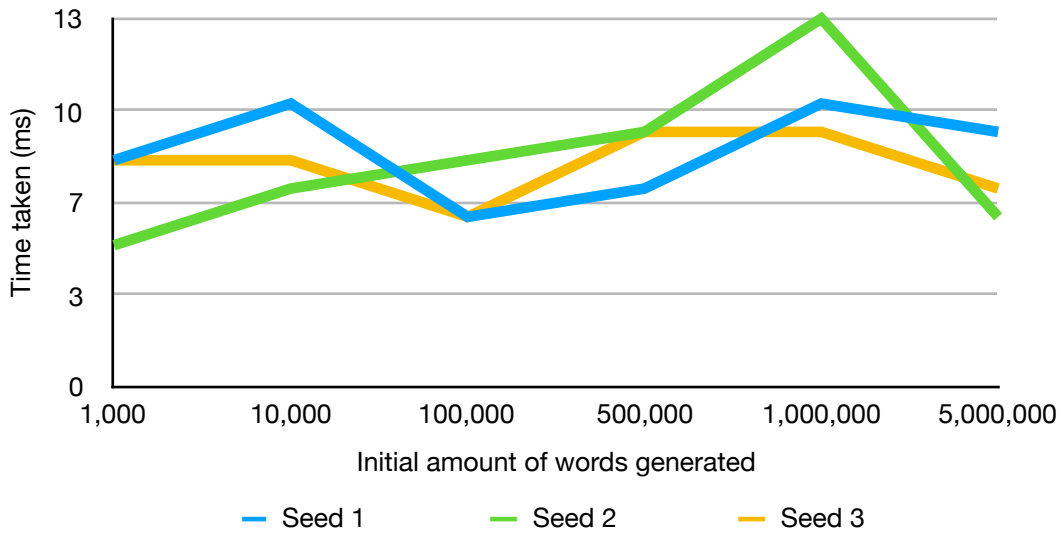


Figure 5: Testing against 100,000 words

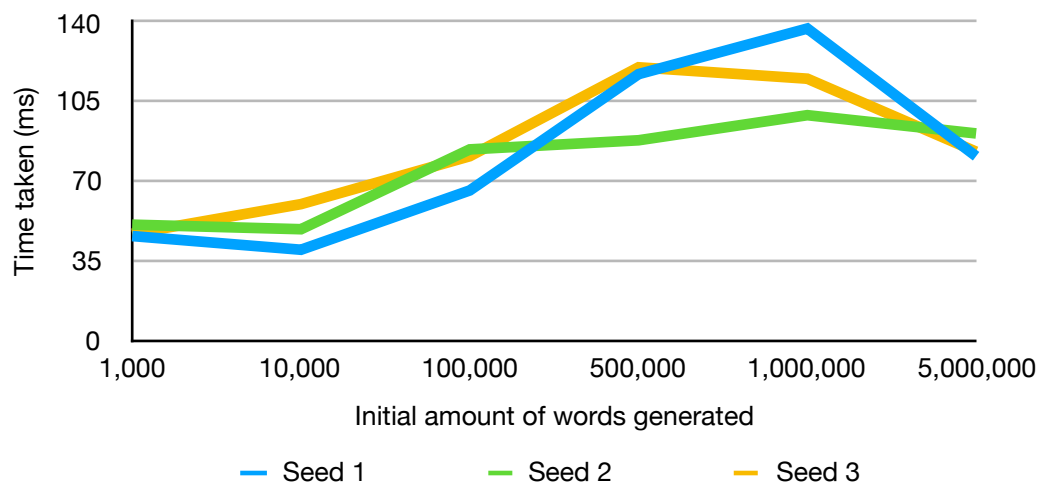


Figure 4 shows that there is a steady increase with the time taken to check occurrences of 10,000 random words with no test exceeding 15 ms.

Figure 5 shows that there is a steady increase with the time taken to check occurrences of 100,000 random words with no test exceeding 150 ms.

The spike decrease after 1,000,000 words could be due to the difference between the two tested numbers of 100,000 and 5,000,000 being a very big gap. This only shows that the algorithm is increasing its successes of $O(1)$ time complexity as the hash table size is becoming larger.

Figure 6 | Adding 10,000 more words

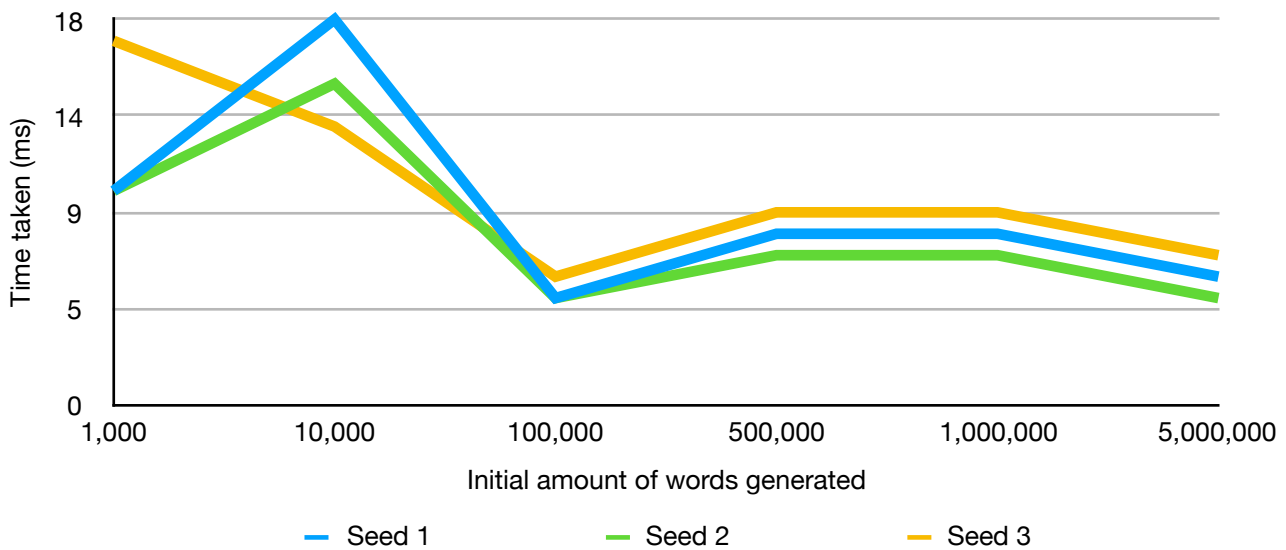


Figure 7 | Adding 100,000 more words

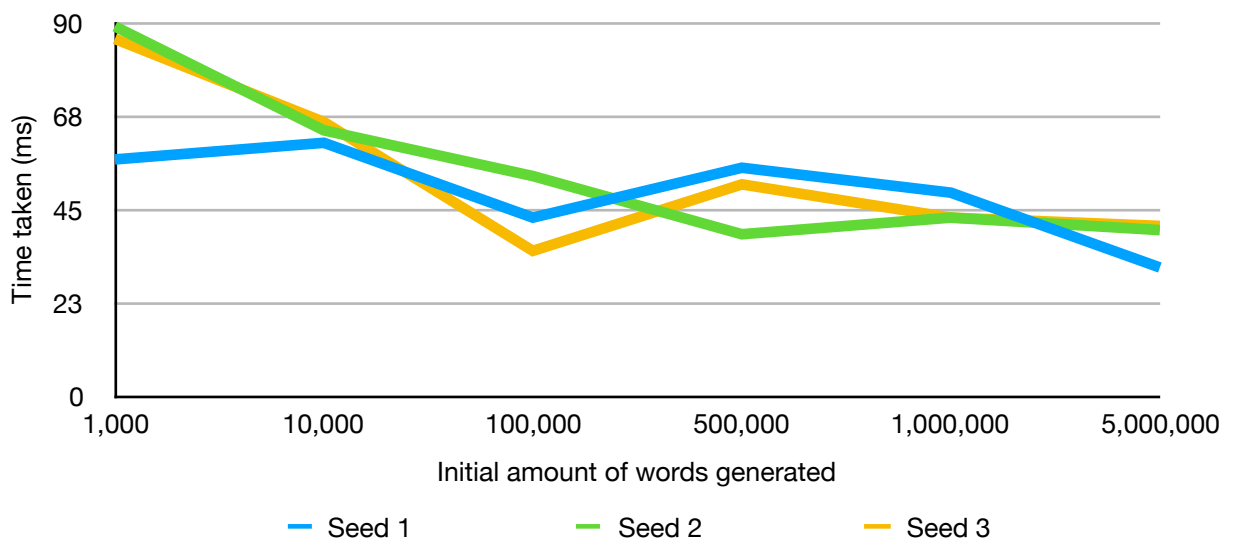


Figure 6 shows that the time taken is higher with the words added being a larger set than the words initially generated.

There is a steep improvement in the time taken past the point of 10,000 initial words generated, as: when initially 100,000 words are generated, the hash table size grows to accommodate that many words, and when 10,000 more is being added, it is a minuscule number compared to the one already generated, and this trend generally follows.

Figure 7 also shows that the time taken is higher with the words added being a larger set than the words initially generated.

Although instead of a spike, the decrease in the time taken is gradual, probably due to the higher amount of ms it takes to add the 100,000 extra words, but the same trend as **Figure 6** still follows.

Figure 8 | Removing 10,000 words

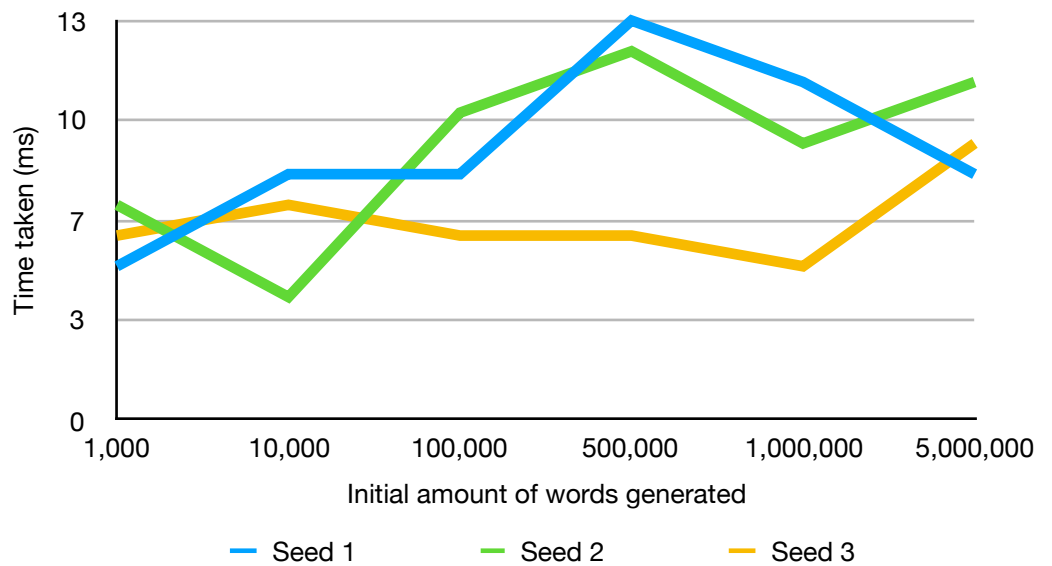


Figure 9 | Removing 100,000 words

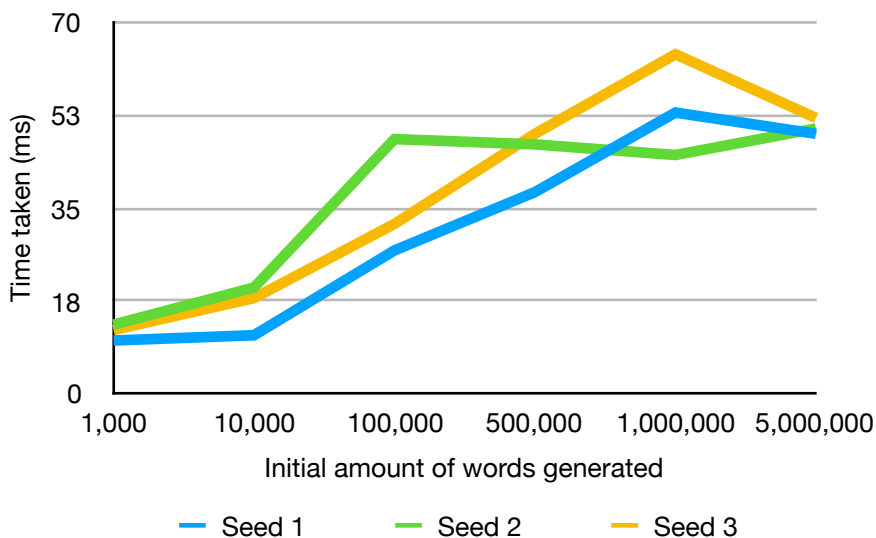
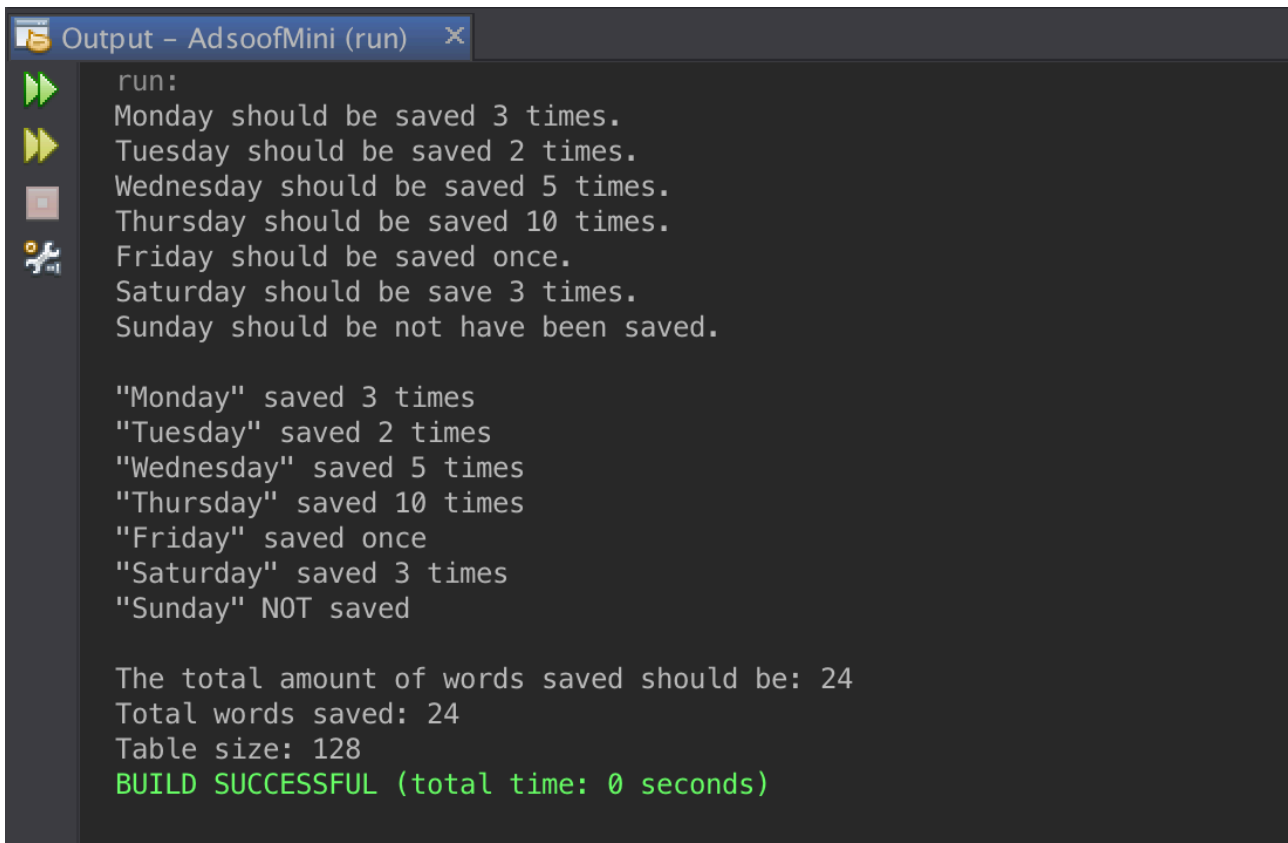


Figure 8 has pretty obscure data as supposed to **Figure 9** possibly due to the seeds themselves, therefore more tests are needed with different seeds to determine the trend.

Figure 9 shows that there is a gradual increase in the time it takes to remove 100,000 words, with it peaking at 1,000,000. Due to the previous test's results, this test also needs to be redone with different seeds.

WordTest5.java



```
Output - AdsoofMini (run) X
run:
Monday should be saved 3 times.
Tuesday should be saved 2 times.
Wednesday should be saved 5 times.
Thursday should be saved 10 times.
Friday should be saved once.
Saturday should be save 3 times.
Sunday should be not have been saved.

"Monday" saved 3 times
"Tuesday" saved 2 times
"Wednesday" saved 5 times
"Thursday" saved 10 times
"Friday" saved once
"Saturday" saved 3 times
"Sunday" NOT saved

The total amount of words saved should be: 24
Total words saved: 24
Table size: 128
BUILD SUCCESSFUL (total time: 0 seconds)
```

As the output console shows, the results from the hash table methods match the numbers.

Future Improvements

Downsizing

One of the features the data structure is currently lacking is the ability to downsize the size of the hash table. This would be a great improvement on memory efficiency if a big amount of data was to be deleted from the hash table.

Handling even bigger data

The next step to handling even bigger data will be to replace linked lists with Binary Trees to further improve efficiency of the hash table.

References

Introduction to hash tables [online]. *Algorithms and Ideas in Java*. 19 October. [viewed 5 December 2017]. Available from: <https://intelligentjava.wordpress.com/tag/hash-table-java-implementation/>

computinglife, 2008. Why do hash functions use prime numbers? [online]. *Computing Life*. 20 November. [viewed 6 December 2017]. Available from: <https://computinglife.wordpress.com/2008/11/20/why-do-hash-functions-use-prime-numbers/>

Drumm, K., 2017. *Hash Tables and Hash Functions* [video]. 5 March 2017. [viewed 4 December 2017]. Available from https://www.youtube.com/watch?v=KyUTuwz_b7Q

Huctbach, M., 2017. Implementing a string collection class [online]. *QMPLUS*. 10 November. [viewed 10 December 2017]. Available from: https://qmplus.qmul.ac.uk/pluginfile.php/1197927/mod_page/content/64/MiniProject.html