ADSOOF Mini Project Write-up

By Mikail Tekneci

160212682

WordStoreImp

```
import java.util.*;

class WordStoreImp implements WordStore {
 private LinkedWordList[] words;
 private HashFunction hashfunc;

 public WordStoreImp(int n) {
      if(n == 0)
           words = new LinkedWordList[1];
      else
           words = new LinkedWordList[n];
      for(int i = 0; i < words.length; i++)
           words[i] = new LinkedWordList();
      hashfunc = new HashFunction();
 }

 public void add(String word) {
      int hash = hashfunc.createHashKey(word, words.length);
      LinkedWordEntry wordToAdd = new LinkedWordEntry(word);
      words[hash].addToHash(wordToAdd);
 }

 public int count(String word) {
      int hash = hashfunc.createHashKey(word, words.length);
      LinkedWordEntry wordToCount = new LinkedWordEntry(word);
      return words[hash].countInList(wordToCount);
 }

 public void remove(String word) {
      int hash = hashfunc.createHashKey(word, words.length);
      LinkedWordEntry wordToRemove = new LinkedWordEntry(word);
      words[hash].removeFirstOcc(wordToRemove);
 }
}
```

LinkedWordList

```
class LinkedWordList {
  private LinkedWordEntry wordAtHash = null;

  public LinkedWordEntry getWordAtHash() {
  return wordAtHash;
  }

  public void addToHash(LinkedWordEntry wordToAdd) {
        if(wordAtHash == null)
              wordAtHash = wordToAdd;
        else {
              LinkedWordEntry currentWordAtHash = wordAtHash;
              wordAtHash = wordToAdd;
              wordAtHash.setNextWord(currentWordAtHash);
        }
  }

  public int countInList(LinkedWordEntry wordToCount) {
        int count = 0;
        LinkedWordEntry currentEntryInChain = wordAtHash;

        while(currentEntryInChain != null) {
              if(currentEntryInChain.getWord().equals(wordToCount.getWord()))
              count++;
              currentEntryInChain = currentEntryInChain.getNextWord();
        }

        return count;
  }

  public void removeFirstOcc(LinkedWordEntry wordToRemove) {
        if(wordAtHash == null) return;

        if(wordAtHash.getWord().equals(wordToRemove.getWord())) {
              wordAtHash = wordAtHash.getNextWord();
              return;
        }

        LinkedWordEntry currentEntryInChain = wordAtHash;
        while(currentEntryInChain != null) {
          if(currentEntryInChain.getWord().equals(wordToRemove.getWord())) {
              currentEntryInChain.setNextWord(currentEntryInChain.getNextWord());
              return;
          }
        currentEntryInChain = currentEntryInChain.getNextWord();
        }
  }
}
```

LinkedWordEntry

```
class LinkedWordEntry {
 private String word;
 private LinkedWordEntry nextWord;

 public LinkedWordEntry(String word) {
      this.word = word;
      this.nextWord = null;
 }

 public String getWord() {
      return word;
 }

 public void setWord(String word) {
      this.word = word;
 }

 public LinkedWordEntry getNextWord() {
      return nextWord;
 }

 public void setNextWord(LinkedWordEntry next) {
      this.nextWord = next;
 }
}
```

HashFunction

```
class HashFunction {
 public int createHashKey(String wordToHash, int arraySize) {
 int hashKey = 0;
      for(int i = 0; i < wordToHash.length(); i++) {
           int alphabetCode = wordToHash.charAt(i) - 96;
           hashKey = (hashKey * 27 + alphabetCode) % arraySize;
      }
      return hashKey;
 }
}
```

Code Explanation

The data structure that I have chosen for this project is Hash Tables. Hash tables are array-like structures in that they store given values at given indexes, however the indexing system is different. The indexes for a hash table are provided by a hashing function. A hashing function is an algorithm that will provide any value you add into your table with a "hash" – an index to locate and store said value in the table. To represent hash tables, I used a combination of an array and linked lists.

In my code, there are three custom classes. LinkedWordEntry is a custom data structure that contains all the essentials of a "word". LinkedWordList contains all the operations that will be run on any given LinkedWordEntry. It has a single variable of type LinkedWordEntry that will point to the data structure containing the values of it, and it will be the class representing the hash table. HashFunction contains an algorithm that creates a hash for any String based on each of its characters' ASCII code. The code of each character is taken away by 96, to represent a – z as 1 – 26. The hash key being made is stored in an accumulator that is multiplied by 27 (this value could be any number greater than 1) then added onto by the alphabet code, for each character in the String. To make sure the hash isn't larger than the table size, the modulo of it is taken from the size (Banas, 2013).

Using this hashing function, however, brings up a major issue in hash tables – collisions. Collisions are when two elements, in this case the String in a LinkedWordEntry, are given the same hash value. During my hash table research, I came across two different algorithms to solve collisions: Separate Chaining and Linear Probing. I decided on using Separate Chaining as it was much more versatile – no need to resize tables, meaning much more efficiency.

Linear Probing involves placing the colliding value into the next free hash. The code for this is very simple, however in the worst-case scenario, for N elements, if there are no available spaces in the table, the entire table would have to be resized which would give a big O notation of O(N^2). Every element is searched through (O(N)), and then they are placed into a new table with one extra index (O(N)). This is very inefficient when it comes to very large sets of data.

Separate Chaining allows for the collision to be resolved by linking the colliding words together. This means each element in the table becomes a linked list (Grag, 2016). If there is a collision, the latest LinkedWordEntry at that hash would have its pointer directed to the incoming one to resolve it. This means that in the worst-case scenario, for N elements, every element would be in one linked list at a single hash, giving a big O notation of O(N), as you would go through all LinkedWordEntry to reach the last one. This is much more efficient than Linear Probing and it can deal with very large sets of data, but will use much more memory (*algolist*).

WordStoreImp initiates any add, count or remove call. It does so by creating a hash using a HashFunction object and invokes the corresponding method in LinkedWordList at the given table index. The addToHash method sets the incoming entry to wordAtHash, if wordAtHash is null. Else, it sets the new word to wordAtHash and the previous word is linked to the new one. This is much more efficient than looping through to find latest word with a null pointer. The countInList method runs a loop through a hash and increments a counter for every occurrence of the given word. The removeFirstOcc method removes the given word once, then returns.

Starting the project, I implemented linear search algorithms, with an array as my data structure. This allowed me to understand the project, but quickly showed its inefficiencies, at O(N^2). When I initially attempted hash tables, I made WordStoreImp into what LinkedWordList was and made an array of LinkedWordEntry, but this was super inefficient as each index would have increasing amounts of data for every collision that occurred.

Correctness Testing

Testing add method:

```
public void printWord() {
  if(wordAtHash == null) return;
  LinkedWordEntry temp = wordAtHash;
  System.out.println(temp.getWord());
  while(temp.getNextWord() != null) {
        System.out.println(temp.getNextWord().getWord());
        temp = temp.getNextWord();
  }
}
```

I used a method to print out every word in a linked list at any given index. This code was placed in LinkedWordList and required a few more bits of code.

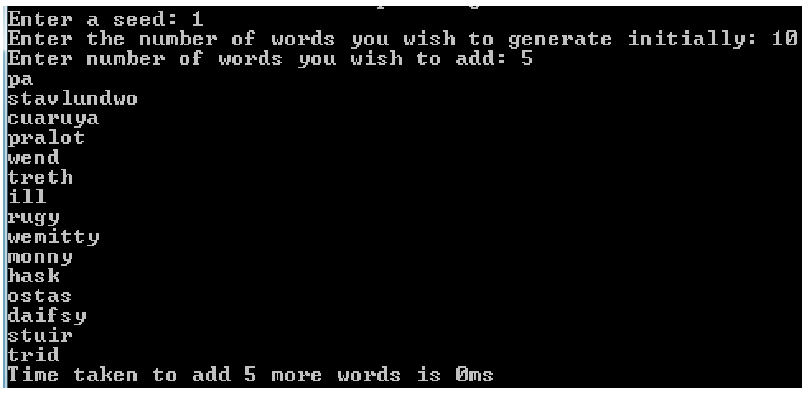
```
public void printWord();

public void printWord() {
  for(int i = 0; i < words.length; i++)
        words[i].printWord();
}
```

The two lines were placed in WordStore and WordStoreImp respectively so that I could actually call the initial method without having to cast the WordStore object in WordTest3.java

```
words.printWord();
```

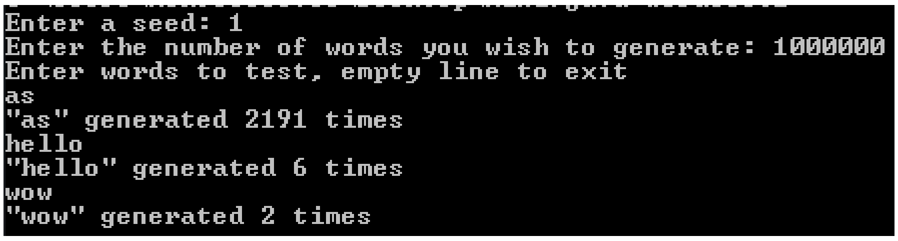
Finally, this line was added to WordTest3 to invoke the method after the words were added. Once all the code was in place, I tested this using seed 1, initially generating 10 words, adding 5 onto that. This produced 15 words on the command prompt screen:



```
Enter a seed: 1
Enter the number of words you wish to generate initially: 10
Enter number of words you wish to add: 5
pa
stavlundwo
cuaruya
pralot
wend
treth
ill
rugy
wemitty
monny
hask
ostas
daifsy
stuir
trid
Time taken to add 5 more words is 0ms
```

Testing count method:

WordTest1 is a perfect test to see if the count method works, and here is the result of using seed 1 with 1 million words generated, testing a few words:



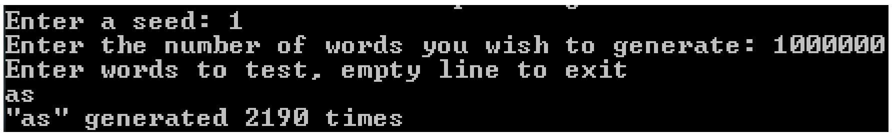
```
Enter a seed: 1
Enter the number of words you wish to generate: 1000000
Enter words to test, empty line to exit
as
"as" generated 2191 times
hello
"hello" generated 6 times
wow
"wow" generated 2 times
```

Testing remove method:

Testing the remove function is as simple as adding one line to the WordTest1 file:

```
words.remove("as");
```

This will remove an occurrence of the LinkedWordEntry that contains the String “as”, and if we run WordTest1 after this code has been added, we can see that with the same seed, the number of “as” generated has decreased by 1:



```
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 2190 times
```

Testing for an initial word number of 0:

To ensure that starting with 0 words initially doesn’t cause exceptions, if the user does type in an initial size of 0, it will just set the starting size to 1.

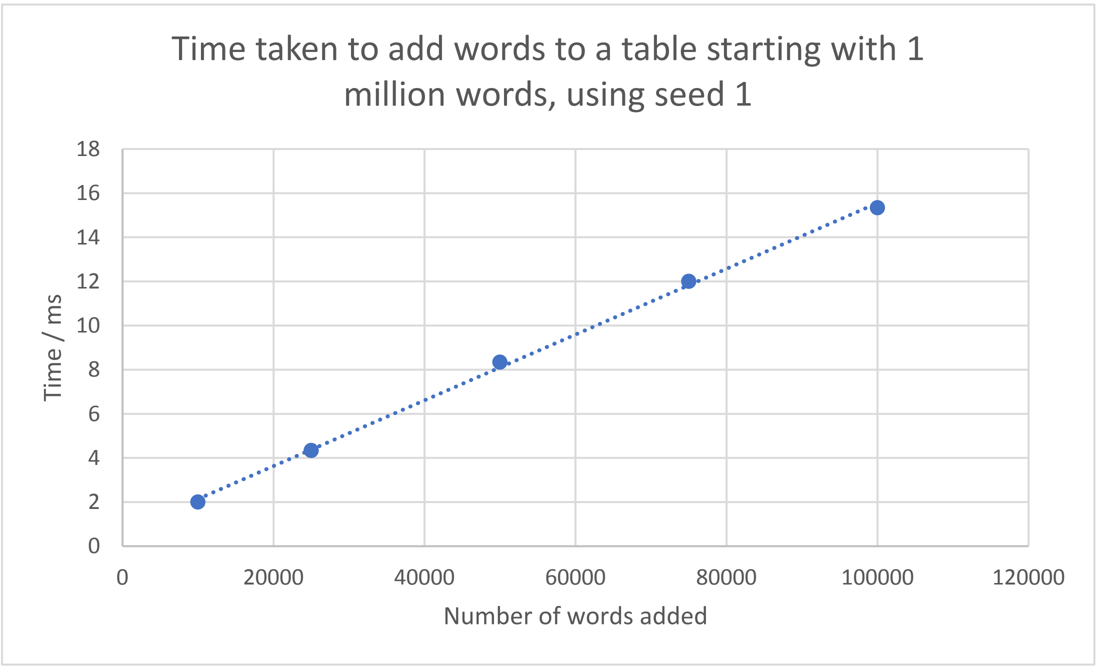
```
if(n == 0)
        words = new LinkedWordList[1];
```

This will mean that any word added after will all be hashed to 0, as anything number modulo 1 is equal to 0.

Efficiency Testing

Time taken to add words to a table starting with 1 million words, using seed 1:

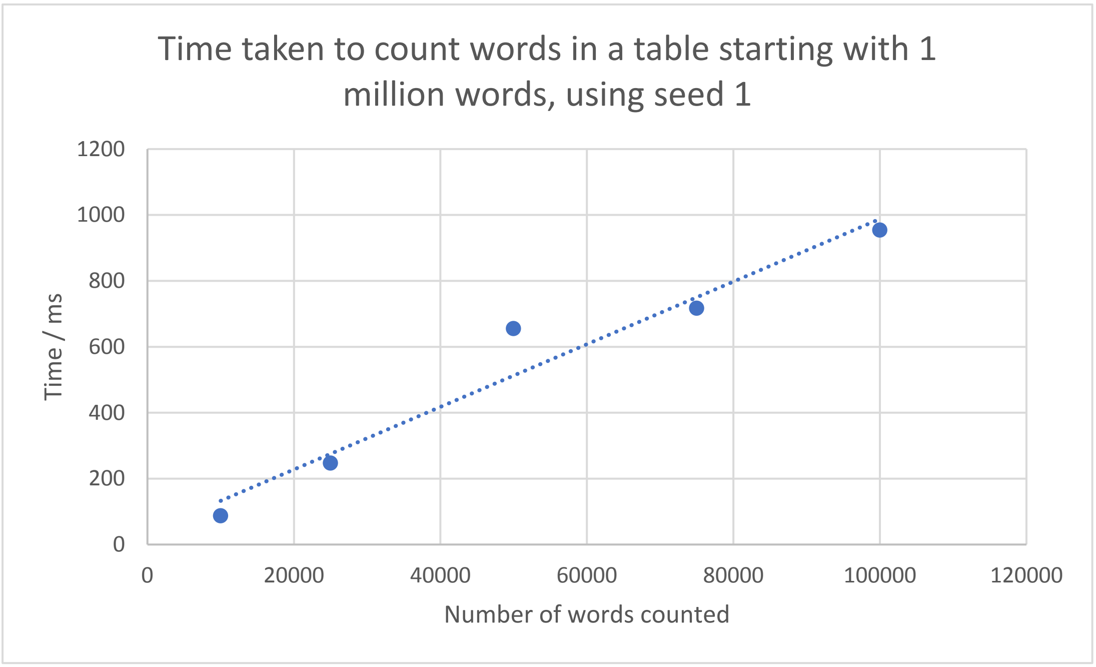
Words	Time / ms				Time per word / ms
	Run 1	Run 2	Run 3	Average	
10000	2	2	2	2	0.0002
25000	4	5	4	4.333333	0.000173333
50000	8	8	9	8.333333	0.000166667
75000	12	12	12	12	0.00016
100000	16	15	15	15.33333	0.000153333

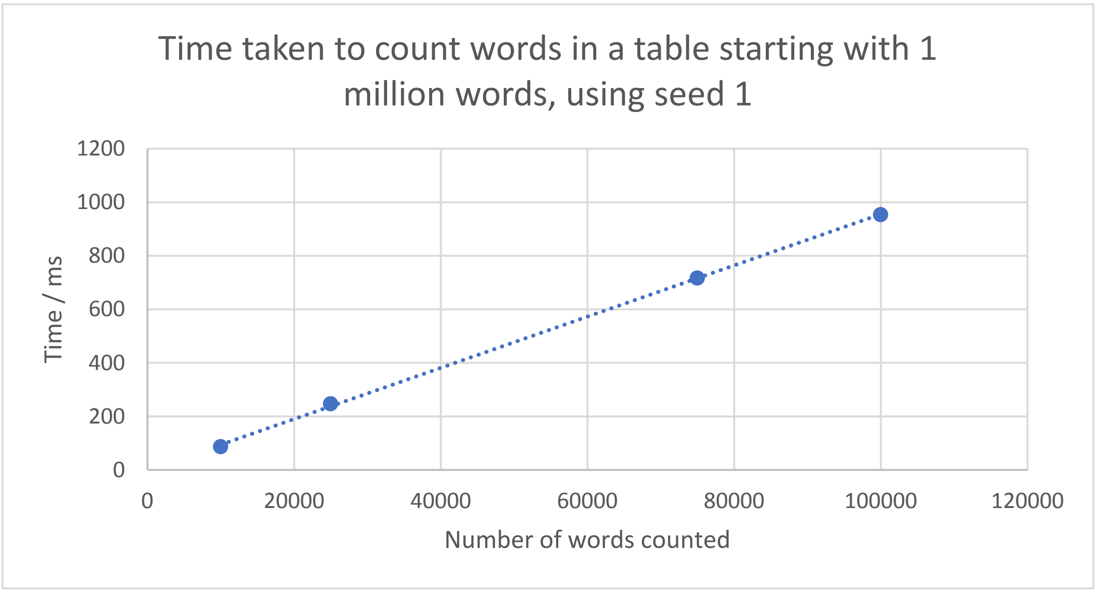


Looking at the results from adding various words to a collection of a million words, this is very much what I expected to see, a linear graph, representing a theoretical complexity of O(N). The time per word, however, is quite a weird sight, as I would expect adding a larger set of words to take more time in general. This variation could be caused by words later generated from the seed 1 that are assigned a hash that is empty, which would add the word in an O(1) time.

Time taken to count words in a table starting with 1 million words, using seed 1:

Words	Time / ms				Time per word / ms
	Run 1	Run 2	Run 3	Average	
10000	100	70	90	86.66667	0.008666667
25000	240	244	256	246.6667	0.009866667
50000	669	633	663	655	0.0131
75000	711	739	700	716.6667	0.009555556
100000	907	1005	949	953.6667	0.009536667

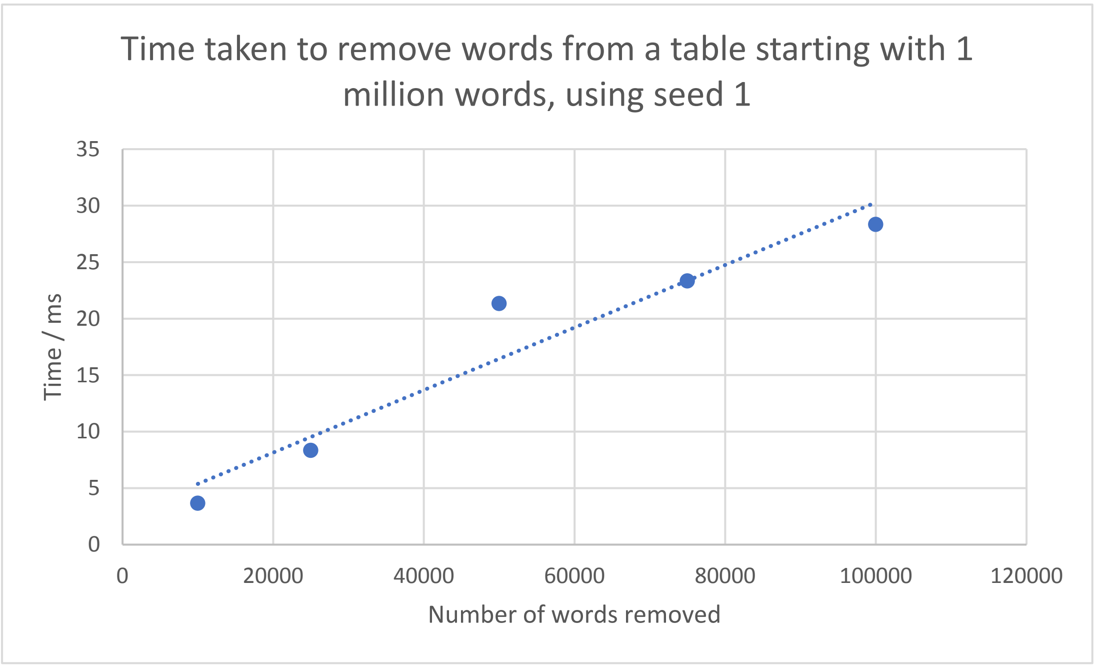


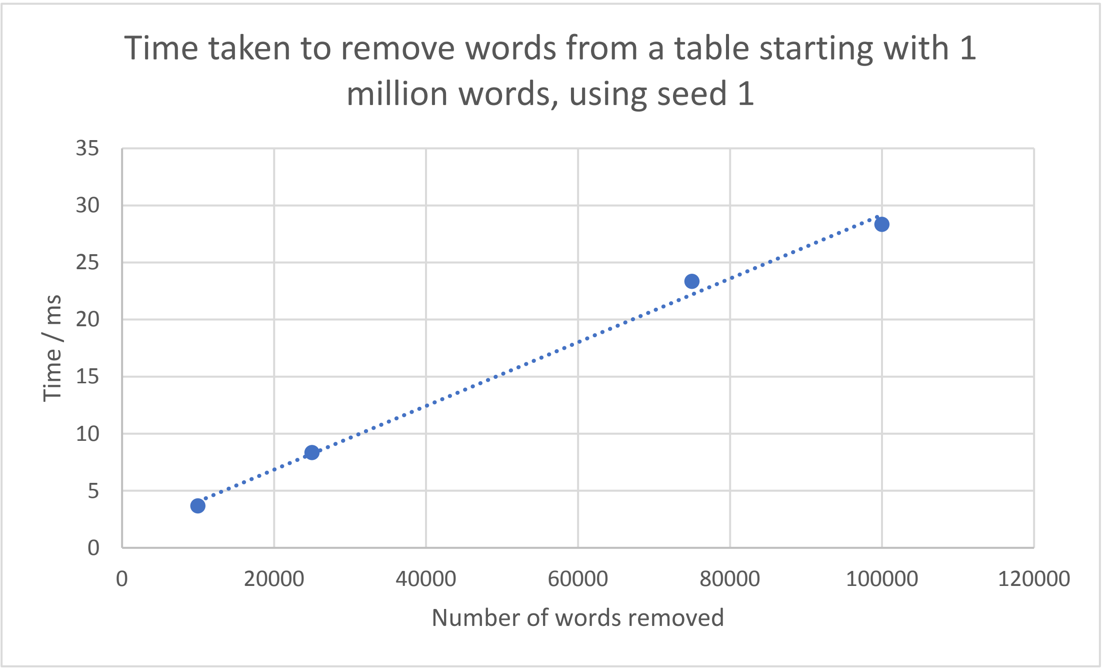


The results for counting the number of occurrences of a given number of words show a somewhat linear graph as expected, however there seems to be an anomaly at testing 50000 words so there are two graphs, one with the anomaly omitted. This gives a theoretical complexity of O(N) as the count method would go through every entry at the given hash to ensure the word exists, and if so, increment the counter for each hit.

Time taken to remove words in a table starting with 1 million words, using seed 1:

Words	Time / ms				Time per word / ms
	Run 1	Run 2	Run 3	Average	
10000	4	4	3	3.666667	0.000366667
25000	9	8	8	8.333333	0.000333333
50000	21	22	21	21.33333	0.000426667
75000	26	22	22	23.33333	0.000311111
100000	29	29	27	28.33333	0.000283333





Testing the remove method also creates a linear graph, with an anomaly at 50000 words again. Two graphs again, one with the anomaly omitted. The theoretical complexity here would also be O(N) as the method, in the worst case, would have to go through the full linked list at a given hash and not find the word to remove.

Bibliography

- Banas D. *et al* (2013). Java Hash Tables 3. *youtube.* Available: URL https://www.youtube.com/watch?v=SVsT7oG4ap8&feature=youtu.be [Accessed 02/12/17].

- Grag, P. *et al* (2016). Basics of Hash Tables. *hackerearth.* Available: URL https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/ [Accessed 01/12/17].

- Hash table. Collision resolution by chaining. *algolist*. Available: URL http://www.algolist.net/Data_structures/Hash_table/Chaining [Accessed 01/12/17].

Note: These were what I used as research material, I did not directly quote any of them apart from the YouTube video, as I used a similar hashing function. The reference points placed in the code explanation section are there to signify which website I learnt the material from.