ADSOOF Mini-Project Report

By Mohammed Sadeq Rahman 160500741

Contents

[code of Data structure 2](#_Toc500817270)

[code for testing data structure 6](#_Toc500817271)

[Data structure explanation 8](#_Toc500817272)

[Code explanation 10](#_Toc500817273)

[Testing of correctness 15](#_Toc500817274)

[Efficiency of code 17](#_Toc500817275)

[Conclusion 20](#_Toc500817276)

[Bibliography 21](#_Toc500817277)

## code of Data structure

**public** **class** WordStoreImp **implements** WordStore {

**public** WordCellTable sd;

**public** WordStoreImp(**int** \_size) {

sd = **new** WordCellTable(\_size);

}

**public** **void** add(String word) {

sd.add(word);

}

**public** **int** count(String word) {

**return** sd.count(word);

}

**public** **void** remove(String word) {

sd.remove(word);

}

}

**public** **class** WordCell {

**public** String word;

**public** **int** occurrence;

**public** WordCell next;

**public** WordCell(String word) {

**this**.word = word;

occurrence++;

**this**.next = **null**;

}

}

**public** **class** WordCellTable {

**public** WordCell[] LinkedList;

**private** **int** index;

**public** **int** size;

**public** **int** collision;

**public** **int** initialWords = 0;

**public** **int** wordsCapacity = 0;

**public** **int** counter = 0;

**public** **int** oldSize = 0;

**public** **boolean** allowResize = **true**;

**public** String previousWord = "";

**public** **int** converToPow2(**int** \_num){  
 **int** n = 2;

**while** (n < \_num){  
 n \*= 2;

}

**return** n;

}

**public** WordCellTable(**int** \_size) {

**if** (\_size < 0) {

**try** {

**throw** **new** Exception("Cannot initiate negative tables");

} **catch** (Exception e) {

e.printStackTrace();

}

}

**this**.initialWords = \_size;

**this**.size = converToPow2(\_size);

LinkedList = **new** WordCell[**this**.size];

**this**.wordsCapacity = **this**.size - **this**.initialWords;

}

**public** **void** resize(**int** \_size) {

WordCell[] newLinkedList = **new** WordCell[\_size];

**for** (**int** i = 0; i < oldSize; i++) {

WordCell currentPtr = LinkedList[i];

**if** (currentPtr != **null**) {

**for** (**int** z = 0; z < currentPtr.occurrence; z++) {

add(currentPtr.word, newLinkedList);

}

WordCell ptr = currentPtr.next;

**while** (ptr != **null**) {

**for** (**int** p = 0; p < ptr.occurrence; p++) {

add(ptr.word, newLinkedList);

}

ptr = ptr.next;

}

}

}

LinkedList = newLinkedList;

size = \_size;

}

**public** **void** add(String word) {

**if** (allowResize && !word.equals(previousWord)) {

previousWord = word;

/\* Used for resizing the HashTable \*/

counter++;

**if** (counter >= **this**.wordsCapacity) {

counter = 0;

**this**.oldSize = **this**.size;

**if** (**this**.size <= 0)

**this**.size = 1;

**this**.size \*= 2;

**this**.wordsCapacity = **this**.size - **this**.oldSize;

resize(size);

}

} **else** **if** (allowResize && count(word) <= 0) {

previousWord = word;

/\* Used for resizing the HashTable \*/

counter++;

**if** (counter >= **this**.wordsCapacity) {

// Reset the counter

counter = 0;

**this**.oldSize = **this**.size;

**if** (**this**.size <= 0)

**this**.size = 1;

**this**.size \*= 2;

/\* Find out the word capacity \*/

**this**.wordsCapacity = **this**.size - **this**.oldSize;

resize(size);

}

}

**int** wordHash = hashCode(word);

index = hashInd(wordHash);

**if** (LinkedList[index] == **null**) {

LinkedList[index] = **new** WordCell(word);

**return**;

} **else** {

WordCell pointer = LinkedList[index];

**for** (; pointer != **null**; pointer = pointer.next) {

**if** (wordHash == pointer.word.hashCode() && pointer.word.equals(word)) {

pointer.occurrence++;

**return**;

}

}

}

WordCell addWord = **new** WordCell(word);

addWord.next = LinkedList[index];

LinkedList[index] = addWord;

collision++;

}

**public** **void** add(String word, WordCell[] LinkedList) {

**int** wordHash = hashCode(word);

index = hashInd(wordHash);

**if** (LinkedList[index] == **null**) {

LinkedList[index] = **new** WordCell(word);

**return**;

} **else** {

WordCell pointer = LinkedList[index];

**for** (; pointer != **null**; pointer = pointer.next) {

**if** (wordHash == pointer.word.hashCode() && pointer.word.equals(word)) {

pointer.occurrence++;

**return**;

}

}

}

WordCell addWord = **new** WordCell(word);

addWord.next = LinkedList[index];

LinkedList[index] = addWord;

collision++;

}

**public** **int** count(String word) {

**if** (size <= 0)

**return** 0;

**int** wordHash = hashCode(word);

index = hashInd(wordHash);

**if** (LinkedList[index] != **null**) {

WordCell pointer = LinkedList[index];

**for** (; pointer != **null**; pointer = pointer.next) {

**if** (wordHash == pointer.word.hashCode() && pointer.word.equals(word)) {

**return** pointer.occurrence;

}

}

}

**return** 0;

}

**public** **void** remove(String word) {

**int** wordHash = hashCode(word);

index = hashInd(wordHash);

WordCell pointer = LinkedList[index];

**if** (pointer != **null**) {

**for** (; pointer.next != **null** && !pointer.word.equals(word); pointer = pointer.next) {}

**if** (wordHash == pointer.word.hashCode() && pointer.word.equals(word)) {

**if** (pointer.occurrence == 1) {

pointer = pointer.next;

LinkedList[index] = pointer;

} **else** **if** (pointer.occurrence > 1) {

pointer.occurrence--;

}

}

}

}

**public** **int** hashCode(String word) {

**int** val = 0;

**for** (**int** i = 0; i < word.length(); i++)

val = val \* 31 + word.charAt(i);

**return** val;

}

**public** **int** hashInd(**int** hashWord) {

**int** key;

key = hashWord % size;

key = makePositive(key);

**return** key;

}

**public** **int** makePositive(**int** key) {

**if** (key < 0)

key = key \* -1;

**return** key;

}

}

## code for testing data structure

This code below checks the time for adding an amount of words to an initial generated words.

**import** java.util.Date;

**import** java.util.Scanner;

**class** WordTest5 {

**public** **static** **void** main(String[] args) {

Scanner input = **new** Scanner(System.***in***);

WordGen.*initialise*(input);

/\*

\* Test how long (after constructor) takes to fill the HashTable

\*/

System.***out***.print("Enter the number of words you wish to generate initially: ");

**int** n = input.nextInt();

**long** time5, time6;

WordStore words = **new** WordStoreImp(n);

/\*

\* Generate the words of the constructor size

\*/

**for** (**int** i = 0; i < n; i++)

words.add(WordGen.*make*());

/\*

\* Call resize by adding more words

\*/

System.***out***.print("Enter number of words you wish to add: ");

n = input.nextInt();

String[] testWords5 = **new** String[n];

**for** (**int** i = 0; i < n; i++)

testWords5[i] = WordGen.*make*();

**long** time9, time10;

time9 = **new** Date().getTime();

**for** (**int** i = 0; i < n; i++)

words.add(testWords5[i]);

time10 = **new** Date().getTime();

System.***out***.print("Time taken to add " + n + " more words ");

System.***out***.println("is " + (time10 - time9) + "ms");

}

}

This test is being able to manually add a word into the HashTable as many times as the user specifies and then it will use the count method to return an integer of how many is inside this table.

**import** java.util.Scanner;

**public** **class** WordTest7 {

**public** **static** **void** main(String[] args) {

Scanner input = **new** Scanner(System.***in***);

/\*

\* 1. Initial creation

\*/

System.***out***.println("Enter a word");

String s = input.nextLine();

/\*

System.out.print("Enter the initial hashtable size: ");

int n = input.nextInt();

\*/

**int** n = 1;

WordStore words = **new** WordStoreImp(n);

System.***out***.print("Enter the amount of times you want your word ("+ s +") be added into the hashtable: ");

n = input.nextInt();

/\*

\* Generate the words of the constructor size

\*/

**for** (**int** i = 0; i < n; i++)

words.add(s);

System.***out***.println("There is a total of: " + words.count(s) + " added into this hashtable");

}

}

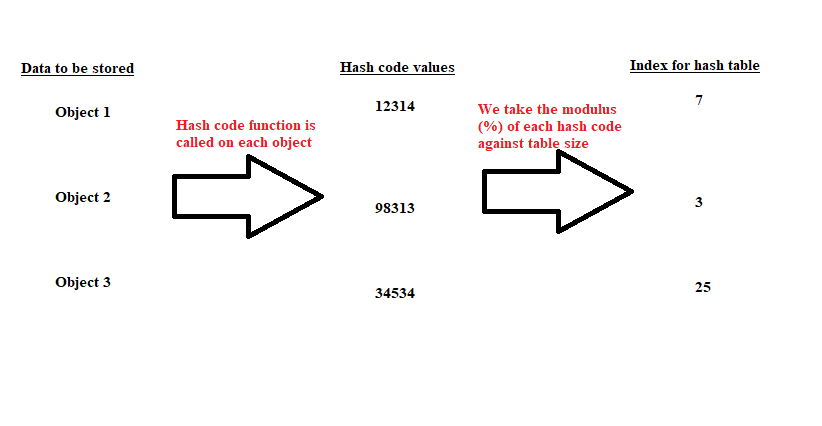
## Data structure explanation

**Data structure used**

For my data structure I have decided to implement a hash table with a (singly) linked list stored inside each index.

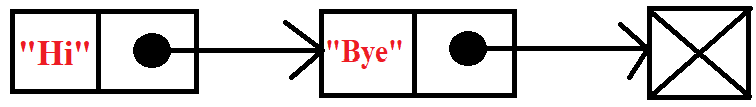
**What is a hash table?**

A hash table is a data structure that is simply an array but where the indexes of the array can be addressed instantaneously (i.e. without having to loop through the array) by using a hash function. The idea is that values, that are to be stored into the data structure, are given a unique number via the hash code function. The hash function is used to generate an index to the array to find the specific value one is looking for. This is done by dividing the hash code values by the size of the array. Indexes can hold singular values or multiple values in some way which (also known as buckets). Here is a visual representation:



**What is linked list?**

A linked list is a data structure that is comprised of singular cells that reference each other (more clearer in the diagram below). Each cell contains a variable of the data element that is to be stored and also a variable which is used reference to the next cell (commonly referred to as *next*). When the last cell’s *next* variable is set to null then the linked list is ended. Here is a visual representation:



**Why I chose to use hash tables with linked list**

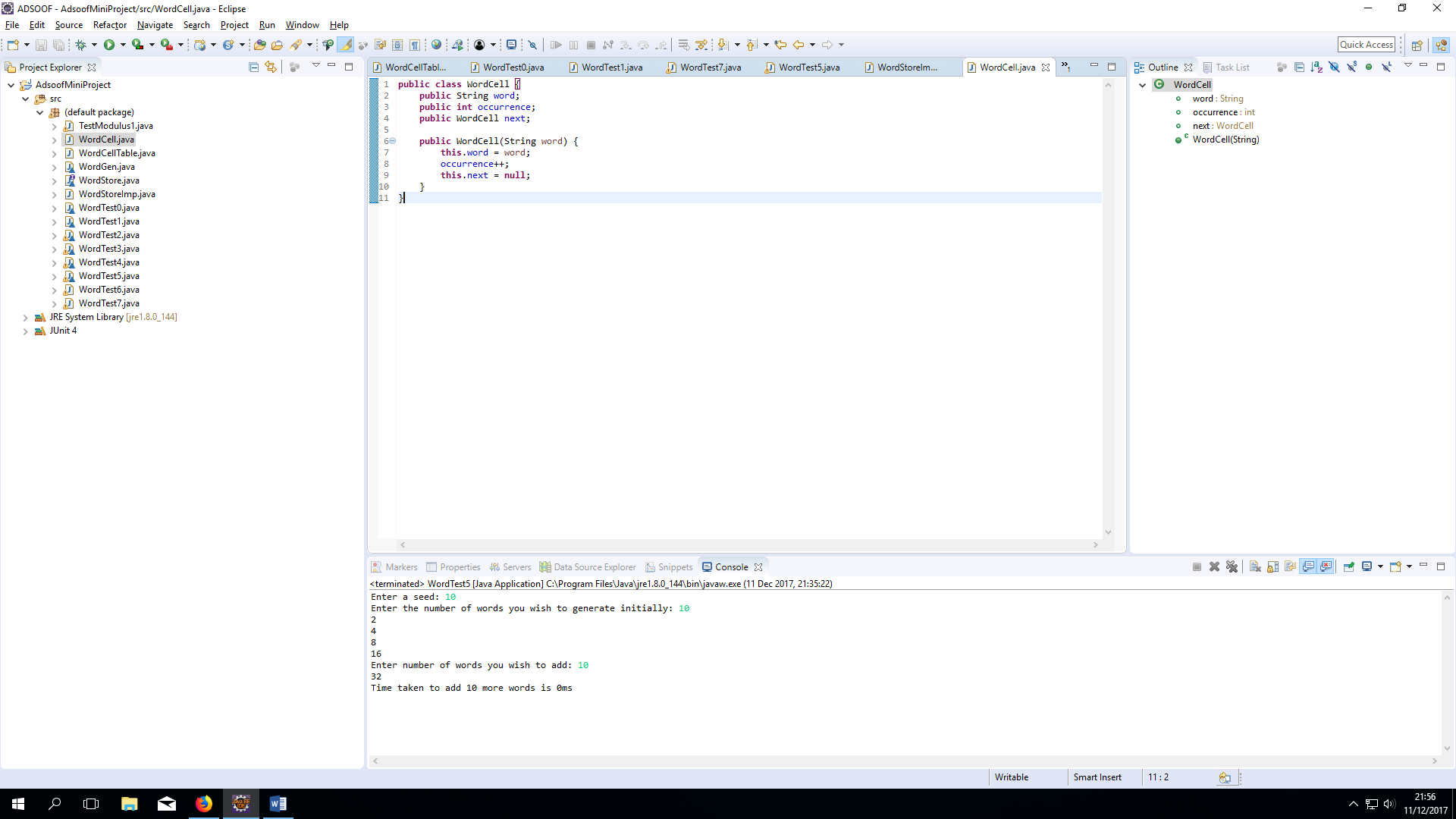
When researching how to implement my data structure, I came across three potential data structures to choose from. Either by using string array to store all the words, using a balanced tree or using a hash table with linked list.

The reason I chose to use hash tables instead of a regular array, is for the fact that you are able to instantaneously retrieve the index of the data element that you are looking for by using a hash code function as mentioned above already. This will save a lot of time when trying to retrieve information and no time is wasted looping through the array when trying to find the target bucket.

When it came to considering balanced trees, I found that balanced trees were argued to be faster than hashing and good for data that needed to be sorted. However, since my data was not required to be sorted and for the fact it showed to be simpler to implement, I chose to implement hash tables with linked lists.

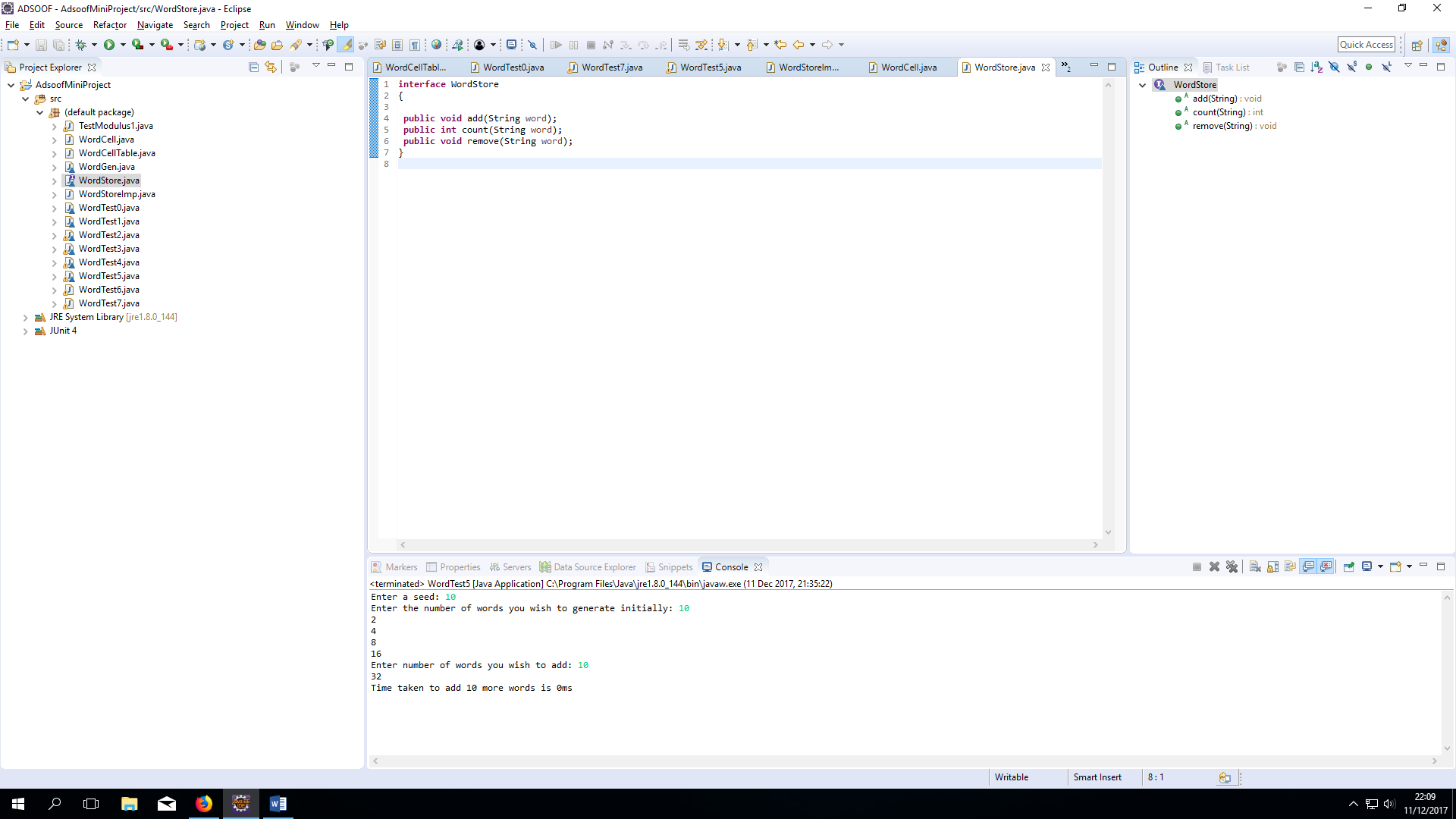
## Code explanation

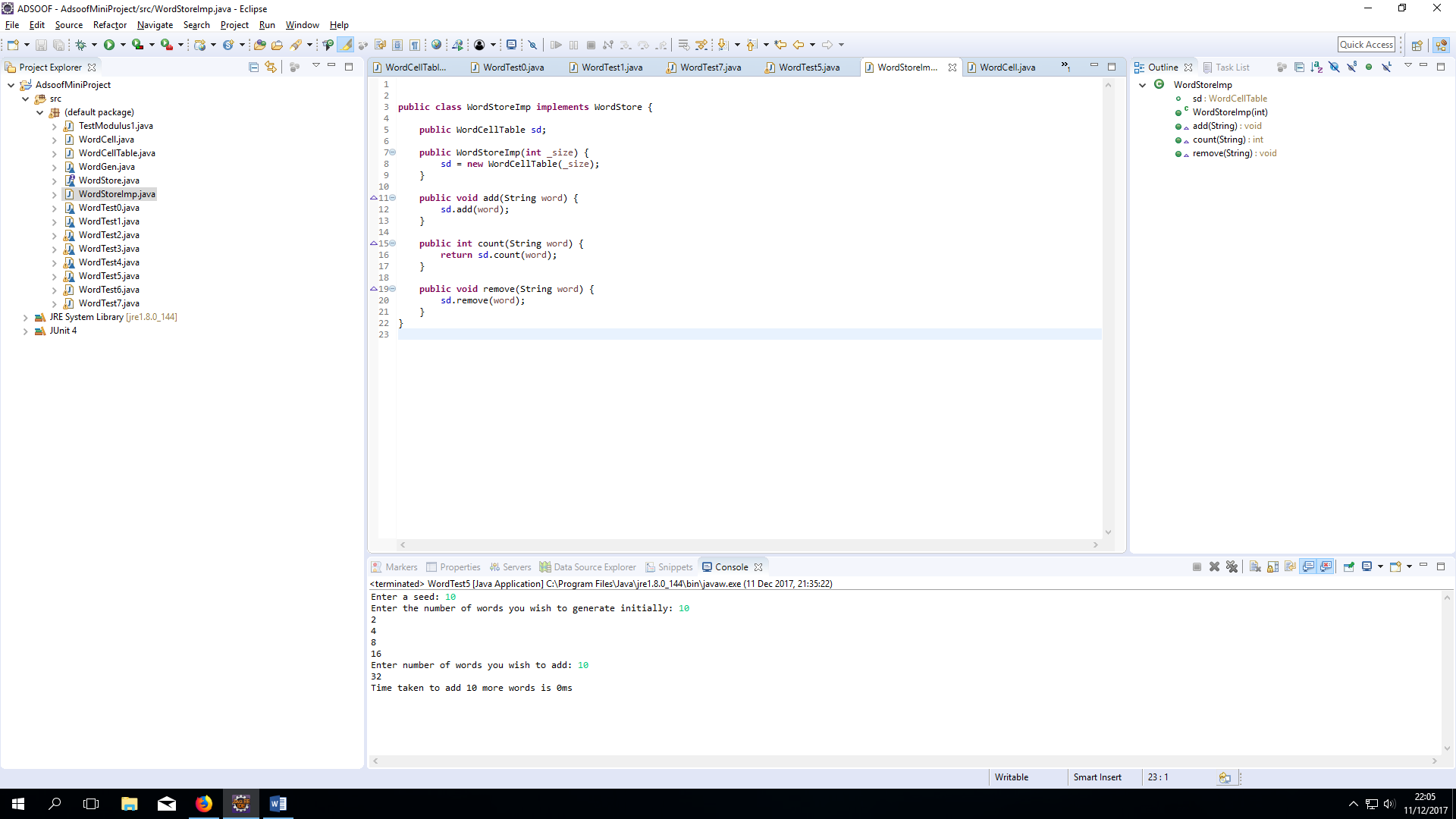
Linked List Cells



WordCell is the class that represents the individual cells that make up my linked list. It will store individual words and count the occurrence of that said word.

WordStore and WordStoreImp Class





The class WordStoreImp implements WordStore and inherits the methods add, count and remove. Inside these methods I call other methods from my data structure class, that execute the required requirements of said method (i.e. add, count or remove) as shown in the code above. add will add a new word to the hash table, remove will remove the said word from the hash table and count will count the number of times the said word has been added to the hash table.

WordCellTable Constructor and convertToPow2 Method

**public** **int** converToPow2(**int** \_num){  
 **int** n = 2;

**while** (n < \_num){  
 n \*= 2;

}

**return** n;

}

**public** WordCellTable(**int** \_size) {

**if** (\_size < 0) {

**try** {

**throw** **new** Exception("Cannot initiate negative tables");

} **catch** (Exception e) {

e.printStackTrace();

}

}

**this**.initialWords = \_size;

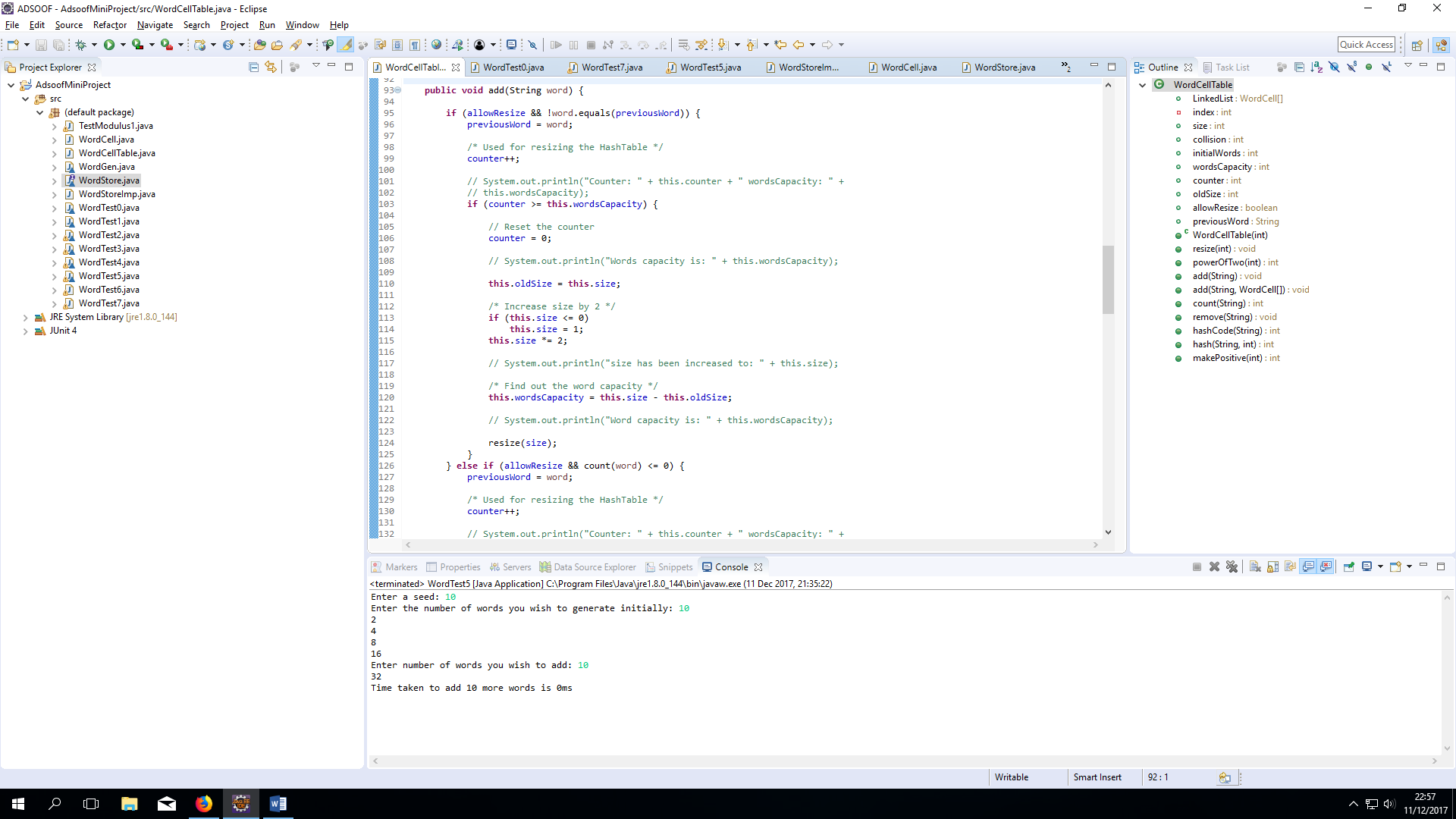
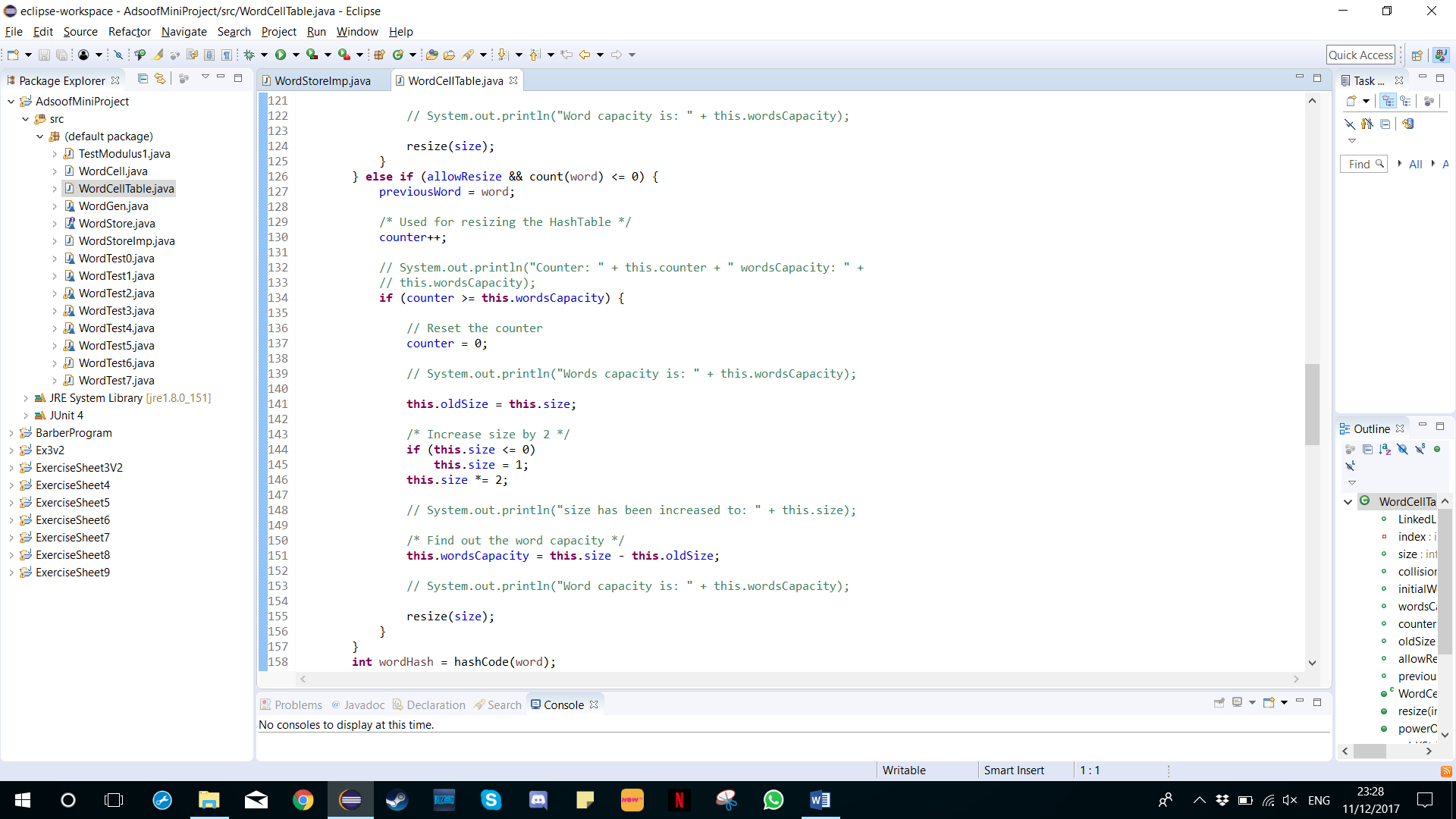
**this**.size = converToPow2(\_size);

LinkedList = **new** WordCell[**this**.size];

**this**.wordsCapacity = **this**.size - **this**.initialWords;

}

The constructor will make sure that if a negative size is passed through in the argument that an exception is thrown, as the argument is used to create the size of the hash table. The size of the array is changed to a number that is the next power of 2 via the method convertToPow2 for efficiency reasons (reasons for convertToPow2 method efficiency will be explained further down in the document). The constructor also records the number of empty indexes in the hash table; I use this to know whether the hash table needs resizing.

Resize and add method

**Adding words**

In my add method, first I check if the hash table needs to be resized (if all the indexes have been used up in the array). The way I have calculated to know whether all the indexes in the hash table store a linked list, i.e. no index in the hash table is null, is by using a count variable.

This count variable is used to record the number of times a new unique word (a word that has not been added into the hash table before) is added to the hash table. As shown above, if the first if or else if condition is met I call the resize method; both conditions are designed to make sure the counter variable only goes up if a new unique word is being added.

A new unique word is added when it is not stored in the cache (previousWord) and the count of this word is less than or equal to 0 which means this word doesn’t exist in the HashTable.

**Resizing prerequisites**

I calculate the wordsCapacity by having the new size which was the original size fitting inside its biggest power of 2 take away the original inputted size for example 1,000,000 was entered for initial size in its constructor and the biggest power of 2 that this number fits in is 1,048,576 so that gives us an extra word capacity of 1,000,000 – 1,048,576 which is 48,576 on top of what the initial size of words was, once we reach other this limit we will resize the HashTable, we can calculate this limit be incrementing a counter on unique words being added.

**The Hash code**

I calculated a hash code which is based off of java’s String.hashCode() which is a number being generated off the number of characters multiplied by a prime number and their ASCII value of individual characters.

**The Hash index**

This is calculated by the hash code modulus the array size so that it fits within the array bounds 0-N.

**The makePositive**

This function basically makes any negative into positive by checking if it’s a negative with less than 0 and then multiplying it by -1

**Removing words**

**public** **void** remove(String word) {

**int** wordHash = hashCode(word);

index = hashInd(wordHash);

WordCell pointer = LinkedList[index];

**if** (pointer != **null**) {

**for** (; pointer.next != **null** && !pointer.word.equals(word); pointer = pointer.next) {}

**if** (wordHash == pointer.word.hashCode() && pointer.word.equals(word)) {

**if** (pointer.occurrence == 1) {

pointer = pointer.next;

LinkedList[index] = pointer;

} **else** **if** (pointer.occurrence > 1) {

pointer.occurrence--;

}

}

}

}

The way we remove words is by calculating the hashcode and then calculating where its hashindex is on the array with its hashcode and then checking if something exists at this hash table index. If something does exist here we need to check if at this linked list object there is a word that equals to the removing word, if it does then we need to check if there is only 1 occurrence of this word, if there is then we need to remove it from the linked list by updating the pointers next with the next node in the linked list. If there is more than 1 occurrence then we need to decrement the amount, for example if there is 5 words of ‘hello’ then when we delete there will be 4 words of ‘hello.

**Counting words**

**public** **int** count(String word) {

**if** (size <= 0)

**return** 0;

**int** wordHash = hashCode(word);

index = hashInd(wordHash);

**if** (LinkedList[index] != **null**) {

WordCell pointer = LinkedList[index];

**for** (; pointer != **null**; pointer = pointer.next) {

**if** (wordHash == pointer.word.hashCode() && pointer.word.equals(word)) {

**return** pointer.occurrence;

}

}

}

**return** 0;

}

This function checks initially if the HashTable size is less than or equal to 0 which would mean that we have no entries to check so we return and don’t continue further. If we do have more than 0 that means we have entries to check. So firstly we again calculate the hash code of the string and then calculate the hashindex of where it’s placed in the array. We then check at this index if it’s not equal to null which will mean there is a LinkedList node here, we then loop through the LinkedList until we find a word that matches, if there is a word that matches then we return the amount of words. If we never find anything then we return 0 i.e. no amount of this word found.

## Testing of correctness

Here I checked that we don’t allow the creation of negative array sizes for the constructor, if they do then we’ll throw an exception.

**public** WordCellTable(**int** \_size) {

**if** (\_size < 0) {

**try** {

**throw** **new** Exception("Cannot initiate negative tables");

} **catch** (Exception e) {

e.printStackTrace();

}

}

**this**.initialWords = \_size;

**this**.size = converToPow2(\_size);

LinkedList = **new** WordCell[**this**.size];

**this**.wordsCapacity = **this**.size - **this**.initialWords;

}

The test below checks the remove function of a particular word found.

**import** java.util.Date;

**import** java.util.Scanner;

**public** **class** WordTest6 {

**public** **static** **void** main(String[] args) {

Scanner input = **new** Scanner(System.***in***);

WordGen.*initialise*(input);

/\*

\* 1. Initial creation

\*/

System.***out***.print("Enter the number of words you wish to generate initially: ");

**int** n = input.nextInt();

WordStore words = **new** WordStoreImp(n);

/\*

\* Generate the words of the constructor size

\*/

**for** (**int** i = 0; i < n; i++)

words.add(WordGen.*make*());

String line1 = input.nextLine();

System.***out***.println("Enter words to test, empty line to exit");

line1 = input.nextLine();

String savedWord = "";

**while** (!line1.equals("")) {

String[] wordlist = line1.split(" ");

**for** (**int** i = 0; i < wordlist.length; i++) {

**int** count1 = words.count(wordlist[i]);

System.***out***.print("\"" + wordlist[i] + "\" ");

**if** (count1 == 0)

System.***out***.println("NOT generated");

**else** **if** (count1 == 1)

System.***out***.println("generated once");

**else**

System.***out***.println("generated " + count1 + " times ");

savedWord = line1;

}

line1 = input.nextLine();

}

System.***out***.println("Removing: " + savedWord);

**int** count1 = words.count(savedWord);

System.***out***.println("now there are " + count1 + " words");

**for**(**int** i=0; i <1E6; i++) {

words.remove(savedWord);

}

count1 = words.count(savedWord);

System.***out***.println("now there are " + count1 + " words left");

}

}

## Efficiency of code

**The efficiency of the adding method**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Words | Adding words | Test 1 | Test 2 | Test 3 | Average (ms) |
| 5,000,000 | 500,000 |  |  |  |  |
| 4,000,000 | 500,000 |  |  |  |  |
| 3,000,000 | 500,000 |  |  |  |  |
| 2,000,000 | 500,000 |  |  |  |  |
| 1,000,000 | 500,000 |  |  |  |  |

**The efficiency of the remove method**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Words | Removing words | Test 1 | Test 2 | Test 3 | Average (ms) |
| 5,000,000 | 500,000 |  |  |  |  |
| 4,000,000 | 500,000 |  |  |  |  |
| 3,000,000 | 500,000 |  |  |  |  |
| 2,000,000 | 500,000 |  |  |  |  |
| 1,000,000 | 500,000 |  |  |  |  |

**Efficiency of the count method**

My count methods Big O Notation is of O (1) lookup as its best case scenario which means it’s the same time regardless of the initial words/counting words.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Initial Words | Counting words | Test 1 | Test 2 | Test 3 | Average (ms) |
| 5,000,000 | 500,000 | 0 | 0 | 0 | 0 |
| 4,000,000 | 500,000 | 0 | 0 | 0 | 0 |
| 3,000,000 | 500,000 | 0 | 0 | 0 | 0 |
| 2,000,000 | 500,000 | 0 | 0 | 0 | 0 |
| 1,000,000 | 500,000 | 0 | 0 | 0 | 0 |

## Conclusion

In conclusion, the hash table is very fast at a cost of increasing memory use, the speeds of the hash table will increase if we increase the overall bucket size, this is because the amount of collisions will decrease, causing us less linked lists being created. If we keep resize on where it will resize, if we have more words than buckets then it will decrease the performance however it will create more memory and keep the overall collision and link list size low making it more memory efficient so that buckets are being used instead of being null.

The hash table speed is also dependent on how fast the hash code is, the less collisions the better, the faster computation time the better. The total bucket size also has an effect on speed because if it is not a power of 2 number then the modulo function of the hash index may have to work extra in some cases, a speed difference of 1-2ms is gained when using a power of 2 array size for the hash table.

If I was to use a hash table for future use, I would use it if I need to work with big data, where I have more than 1million worth of data, an example of this might be trying to collect data from websites and storing them in some database structure like a hash table, the lookup speed is constant O(1) and the insert, deleting are also quite fast.

## Bibliography

I have used the following references as part of my research into the hash table data structure and implementation:

1. <https://www.cs.princeton.edu/courses/archive/spr10/cos226/lectures/10-34HashTables-2x2.pdf>
2. <https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_has.htm>
3. <https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html>
4. <https://www.youtube.com/watch?v=B4vqVDeERhI>