Info 2

Laboratory 11

Prof. Weber-Wulff

14.01.2020

Chung-Fan Tsai

(Chung-Fan.Tsai@Student.HTW-Berlin.de)

Muhammad Safarov

(s0570690@htw-berlin.de)

Pavel Tsvyatkov

(s0559632@htw-berlin.de)

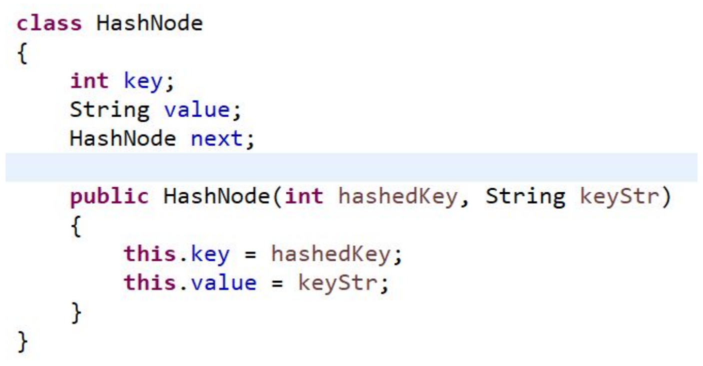
Index

https://people.f4.htw-berlin.de/~weberwu/info2/labs/ExerB.shtml

Lab Exercise

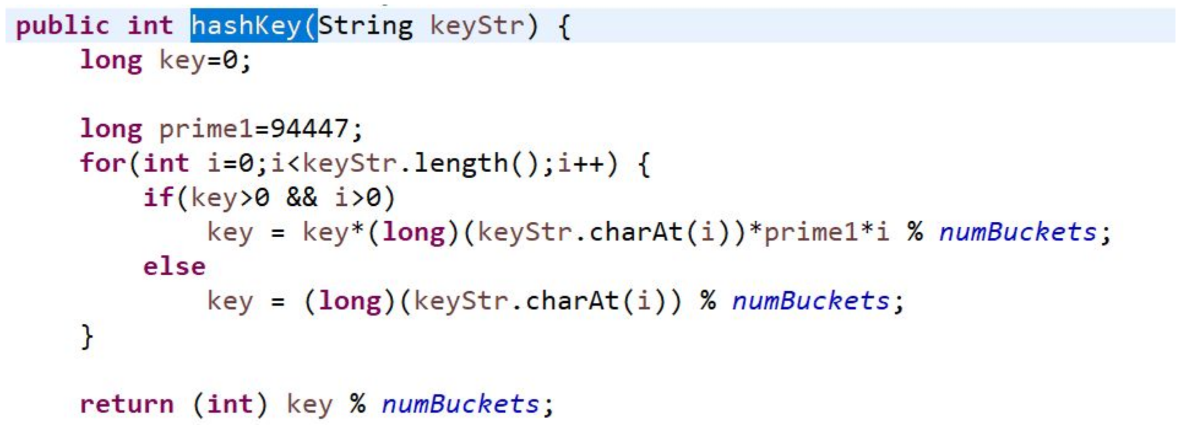
1. **Write a dictionary class that upon instantiation reads in a file of words and creates a hash table for storing them. Use chaining of collisions in your hash table.**

   We first used a small set of words (around 5000) to test, then we created a hash table class using an array of hash nodes (a class we created that has int key, String value and a HashNode next pointer) to store the words.

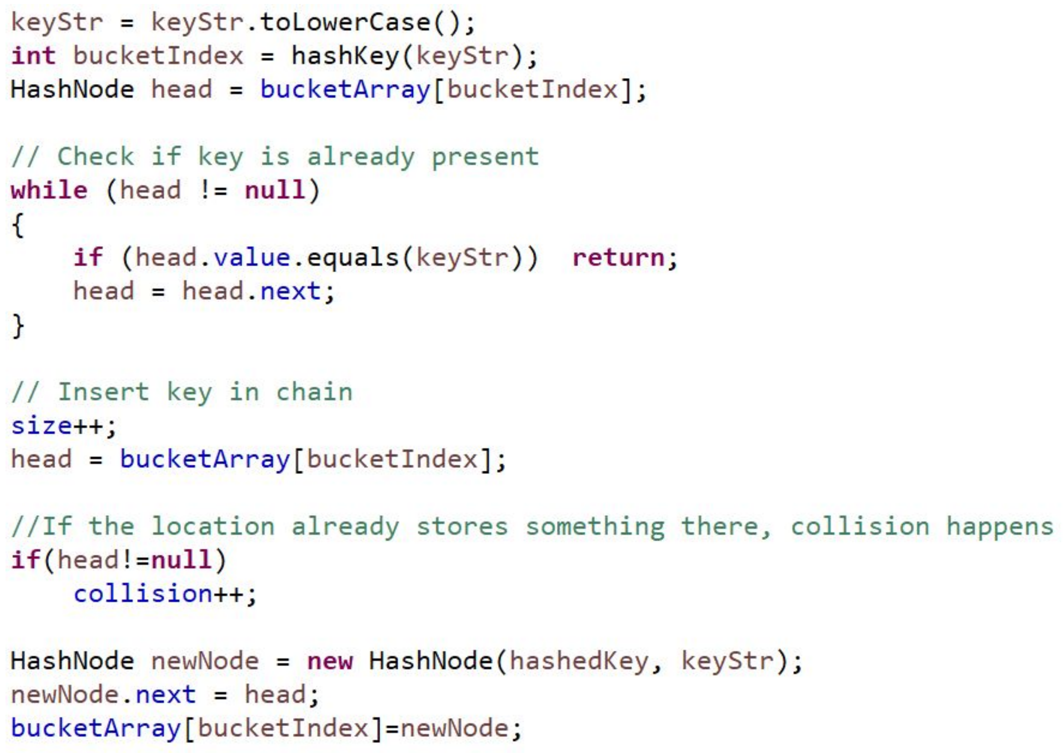


```
class HashNode
{
    int key;
    String value;
    HashNode next;

    public HashNode(int hashedKey, String keyStr)
    {
        this.key = hashedKey;
        this.value = keyStr;
    }
}
```

When we instantiate the class MyHashTable , we give a size/number of buckets to it. After we read from our word file using buffered reader, we call add(String s) that would normalize it (1. To lower case 2. Sorted alphabetically) then hash it using hashKey(String s). In our hashKey method, we use this equation to get to our key that's as unique as possible to that specific string:

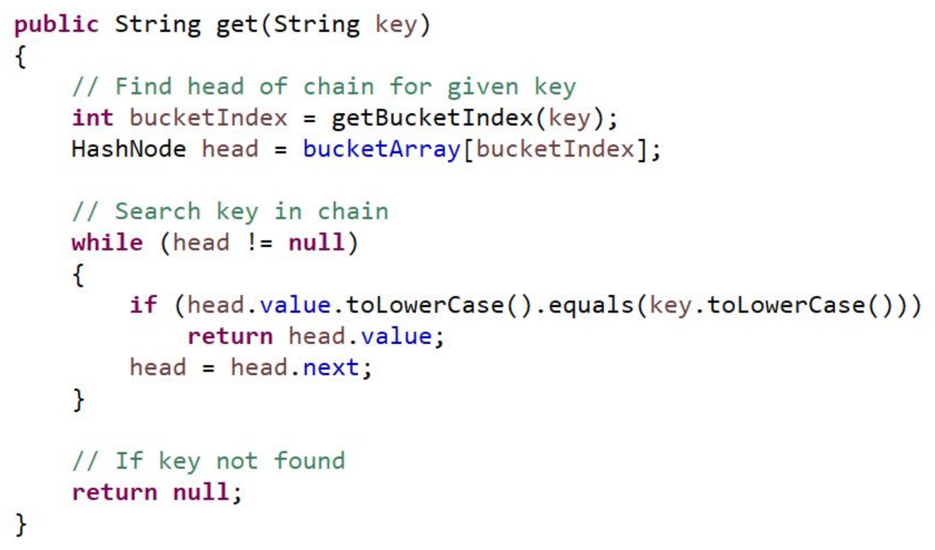

```java
public int hashKey(String keyStr) {
    long key=0;

    long prime1=94447;
    for(int i=0;i<keyStr.length();i++) {
        if(key>0 && i>0)
            key = key*(long)(keyStr.charAt(i))*prime1*i % numBuckets;
        else
            key = (long)(keyStr.charAt(i)) % numBuckets;
    }

    return (int) key % numBuckets;
```

Once we found the right bucket to put it in, in our add() method, we get to that location in our array, if it's empty, create a new node and point the array location to it. If not empty(!! Collision happens), HashNode head = arr[i], then create a new node and assign the node’s next to the head, then lastly, assign arr[i] to our new node, so we are always inserting new node in the beginning/head of the chain just to save time iterating to the end.


```java
keyStr = keyStr.toLowerCase();
int bucketIndex = hashKey(keyStr);
HashNode head = bucketArray[bucketIndex];

// Check if key is already present
while (head != null)
{
    if (head.value.equals(keyStr))  return;
    head = head.next;
}

// Insert key in chain
size++;
head = bucketArray[bucketIndex];

//If the location already stores something there, collision happens
if(head!=null)
    collision++;

HashNode newNode = new HashNode(hashedKey, keyStr);
newNode.next = head;
bucketArray[bucketIndex]=newNode;
```

We are incrementing collision in the add() method so that later on we can print it out and see how many times a word has been hashed to a location where it already stores a word or more.

With get(String s), we hash it and get the bucket index so we know which index of bucket array to look into, then traverse through the chain comparing the value, if we find it, return the value, if not, return null.

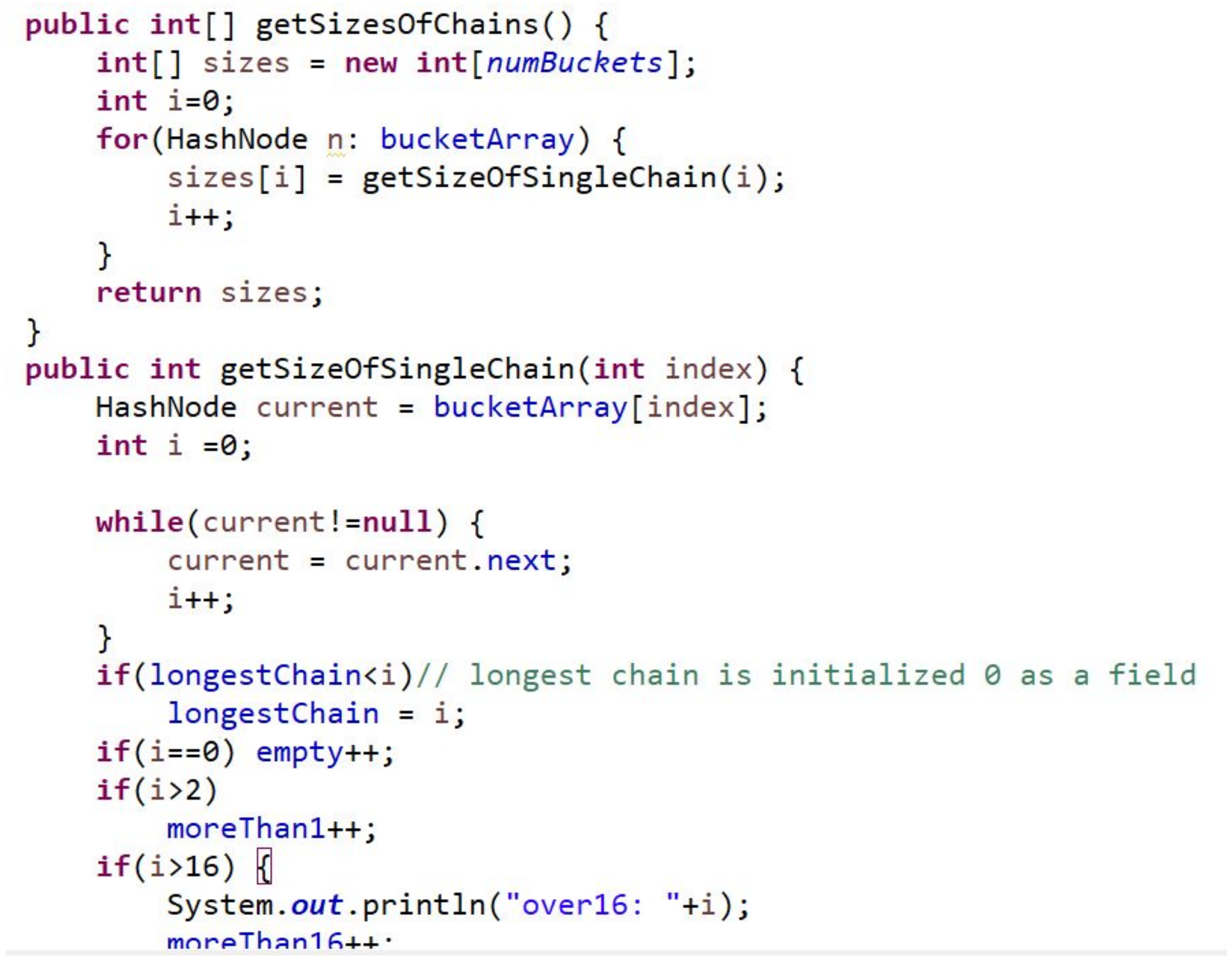


```
public String get(String key)
{
    // Find head of chain for given key
    int bucketIndex = getBucketIndex(key);
    HashNode head = bucketArray[bucketIndex];

    // Search key in chain
    while (head != null)
    {
        if (head.value.toLowerCase().equals(key.toLowerCase()))
            return head.value;
        head = head.next;
    }

    // If key not found
    return null;
}
```

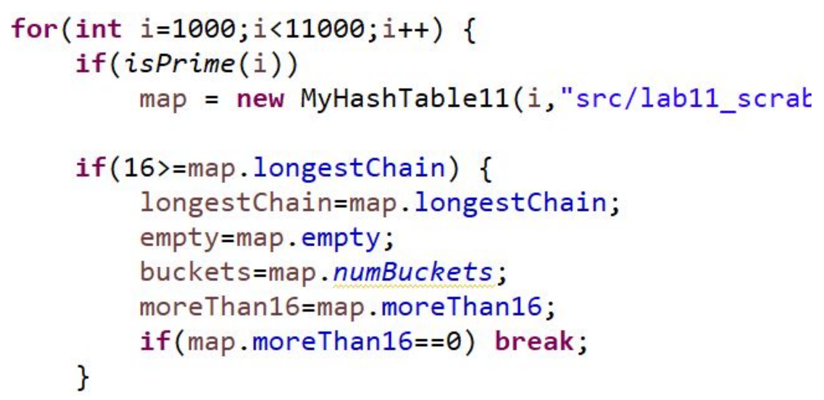
Remove(String s) is quite similar to get, instead of returning the value when found, we set a previous node pointer to remember the node before the node we want to remove, so that it can simply point to its next next node, thus jump the node we want to remove. Of course we'd need to check if the node we look for is at the head position, then we just assign bucket[index]=head.next.

1. **(b)How many entries does your table have? How many collisions were there? What is the longest chain in your hash table? It might be useful to implement some statistical methods in order to see if your hash table is "okay". Can you fix your hash function in order to only have chains of 16 or less?**

Because we increment the global variable size by one every time a node is successfully added to our table, we simply use a getter to get how many nodes/words/entries are in the table. And since a collision occurs whenever a hashed collection already has a word at the location, we increment the variable "collision" by one everytime that happens. But if collision means how many chains there are that have more than one node/word at a location, we simply implement a getBucketSizes() method that iterate through the array and each head node to see how many words are there at each location, so when there are more than one, we increment variable “moreThan1”:

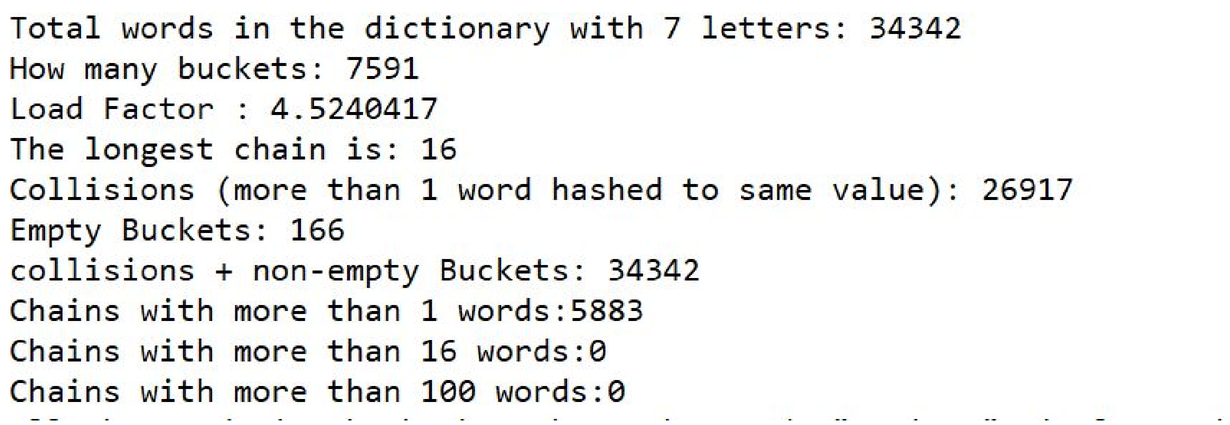

```
public int[] getSizesOfChains() {
    int[] sizes = new int[numBuckets];
    int i=0;
    for(HashNode n: bucketArray) {
        sizes[i] = getSizeOfSingleChain(i);
        i++;
    }
    return sizes;
}
public int getSizeOfSingleChain(int index) {
    HashNode current = bucketArray[index];
    int i =0;

    while(current!=null) {
        current = current.next;
        i++;
    }
    if(longestChain<i)// longest chain is initialized 0 as a field
        longestChain = i;
    if(i==0) empty++;
    if(i>2)
        moreThan1++;
    if(i>16) {
        System.out.println("over16: "+i);
        moreThan16++;
```

As seen in the code above, we have another global variable longestChain that compares to i(how many nodes/words are in the chain), if i is longer, set longestChain to i. Then there’s another problem, how do we find the right number of buckets to initiate the bucketArray’s size so that it’s more evenly distributed, and so that there’s no chains longer than 16? This is where isPrime() comes into use. So we used a loop to test bucket number and when the first time there’s no chain larger than 16, we break the loop and get that number:



```
for(int i=1000;i<11000;i++) {
    if(isPrime(i))
        map = new MyHashTable11(i,"src/lab11_scrat

    if(16>=map.longestChain) {
        longestChain=map.longestChain;
        empty=map.empty;
        buckets=map.numBuckets;
        moreThan16=map.moreThan16;
        if(map.moreThan16==0) break;
    }
```

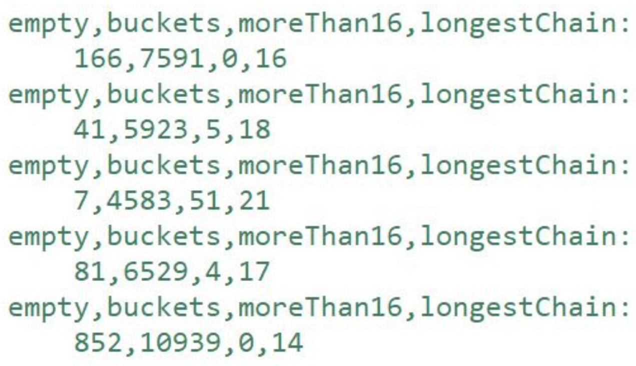
That is how we got the following statistics from the desired table:



```
Total words in the dictionary with 7 letters: 34342
How many buckets: 7591
Load Factor : 4.5240417
The longest chain is: 16
Collisions (more than 1 word hashed to same value): 26917
Empty Buckets: 166
collisions + non-empty Buckets: 34342
Chains with more than 1 words:5883
Chains with more than 16 words:0
Chains with more than 100 words:0
```

Load factor means with the current size of 7591(that we got from looping through prime numbers), the perfectly distributed table will have 4-5 words at each index/location.

However, while the longest chain is 16 words, we have so many empty buckets with this number. So we continued testing different numbers of buckets and got the following result:

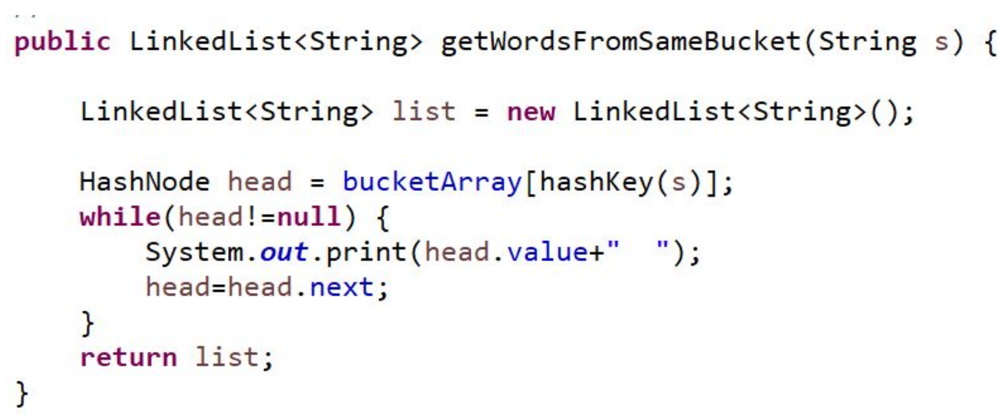


```
empty,buckets,moreThan16,longestChain:
    166,7591,0,16
empty,buckets,moreThan16,longestChain:
    41,5923,5,18
empty,buckets,moreThan16,longestChain:
    7,4583,51,21
empty,buckets,moreThan16,longestChain:
    81,6529,4,17
empty,buckets,moreThan16,longestChain:
    852,10939,0,14
```

As one can see, if we use 4583 buckets, we'll only have 7 of them empty, even though there will be 51 chains with more than 16 words. But it's not the end of the world since the longest of these are 21 top, meaning when we iterate through the chain, it doesn't really take much longer. And with 4583 buckets, it needs only about ⅗ buckets than 7591 buckets.

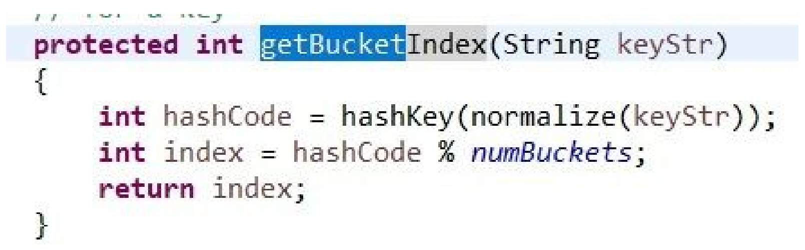
2. **You will need to have a lookup method in your class that takes a word (i.e. a String) and returns an array of Strings corresponding to all the words at the hash location, if any. You may need to normalize the word to look up, depending on your hash function.**

   In order to solve this problem we created a new method called getWordsFromSameBucket.



```
public LinkedList<String> getWordsFromSameBucket(String s) {

    LinkedList<String> list = new LinkedList<String>();

    HashNode head = bucketArray[hashKey(s)];
    while(head!=null) {
        System.out.print(head.value+"  ");
        head=head.next;
    }
    return list;
}
```

After receiving a String as a parameter, our getBucketIndex() method will take care of normalizing the String and it will return an int index of it in our hash table.

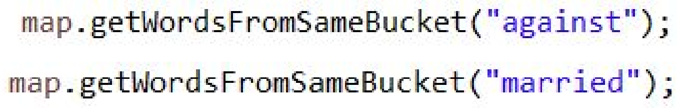


```
protected int getBucketIndex(String keyStr)
{
    int hashCode = hashKey(normalize(keyStr));
    int index = hashCode % numBuckets;
    return index;
}
```

The next step is assigning this index number to the head of the particular chain in our hash table and counting up the words. If there are any words they will be printed out.

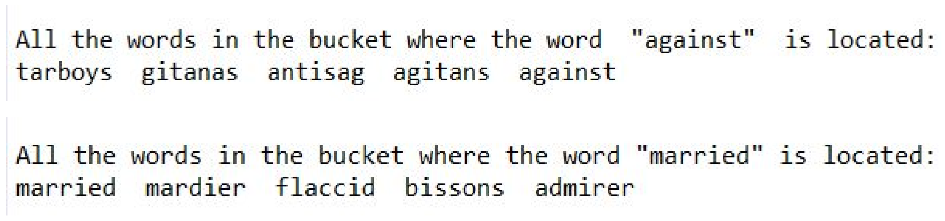
Otherwise there were no words at the particular location of the table.

Now we will try to print out a chain where the words “against” and “married” are located.



```
map.getWordsFromSameBucket("against");
map.getWordsFromSameBucket("married");
```

And we got the following result.

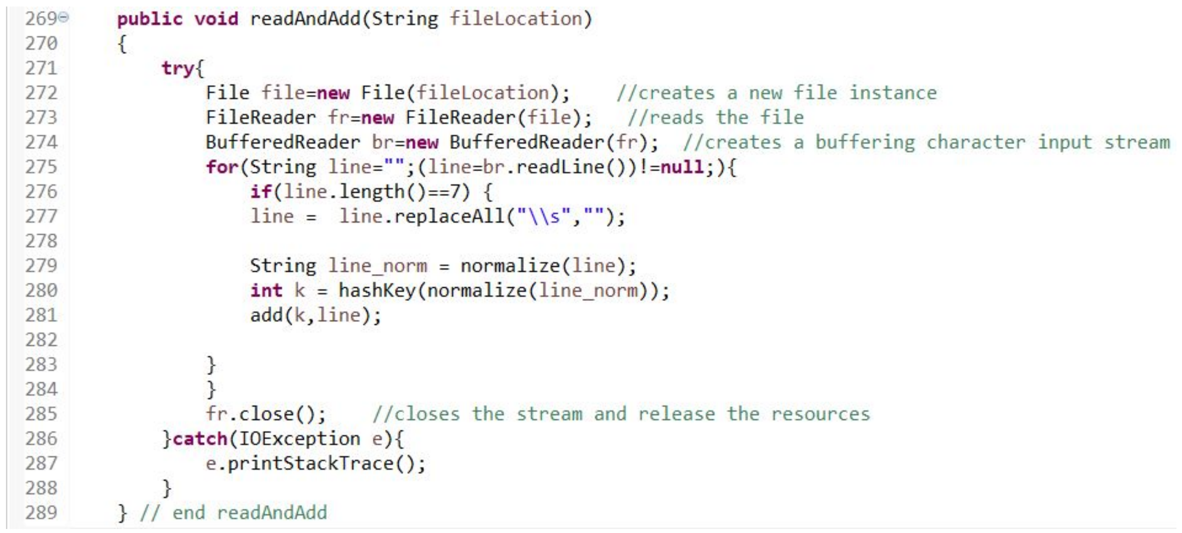


```
All the words in the bucket where the word  "against"  is located:
tarboys  gitanas  antisag  agitans  against

All the words in the bucket where the word "married" is located:
married  mardier  flaccid  bissons  admirer
```

All the chained words that were at the respective index were printed out.

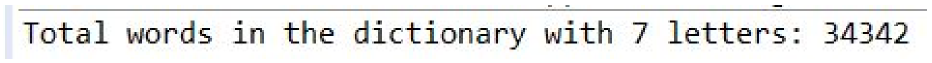
3. **Now make the basic Scrabble cheater: construct a 7-letter-word hash dictionary, set a String to 7 letters, and output the array of Strings found that might be permutations of these 7 letters. Your users can check if there is a permutation to be found. Or you can implement isPermutation and only output the ones that are permutations if you are bored.**

Since we read in a file that has words with different length, in our readAndAdd method we added the line if(line.length() == 7) so that we only add 7-letter words to it.



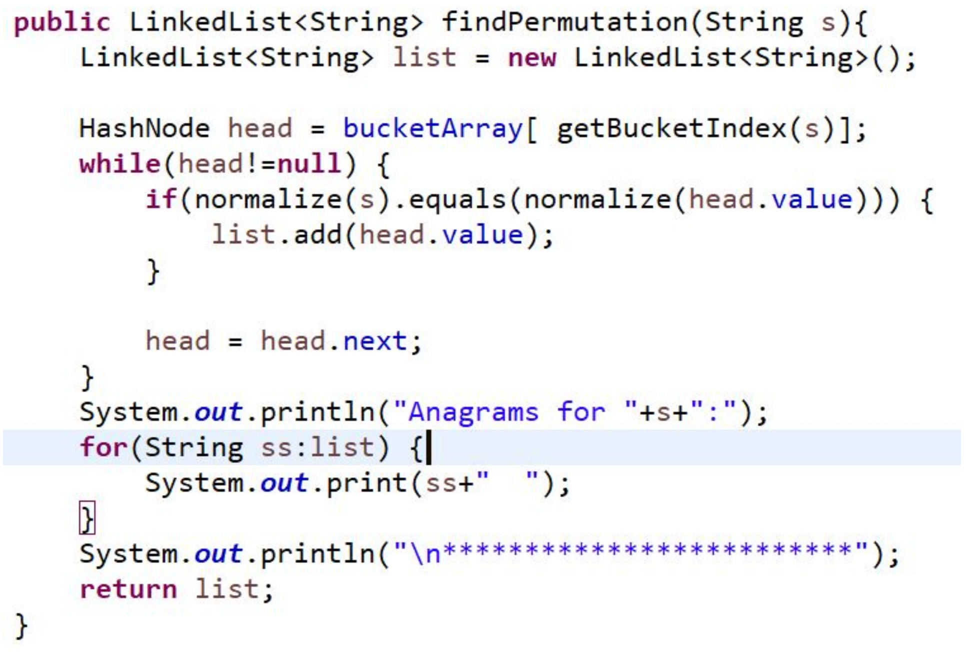
```
    public void readAndAdd(String fileLocation)
    {
        try{
            File file=new File(fileLocation);    //creates a new file instance
            FileReader fr=new FileReader(file);   //reads the file
            BufferedReader br=new BufferedReader(fr);  //creates a buffering character input stream
            for(String line="";(line=br.readLine())!=null;){
                if(line.length()==7) {
                line =  line.replaceAll("\\s","");

                String line_norm = normalize(line);
                int k = hashKey(normalize(line_norm));
                add(k,line);

            }
            }
            fr.close();    //closes the stream and release the resources
        }catch(IOException e){
            e.printStackTrace();
        }
    } // end readAndAdd
```

After running our program, we display a proper message about the words.



```
Total words in the dictionary with 7 letters: 34342
```

To output the Strings that might be permutations, we created a new method and called it findPermutation. It takes a String as a parameter and returns a LinkedList.



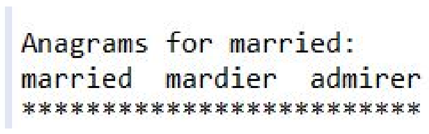
```java
public LinkedList<String> findPermutation(String s){
    LinkedList<String> list = new LinkedList<String>();

    HashNode head = bucketArray[ getBucketIndex(s)];
    while(head!=null) {
        if(normalize(s).equals(normalize(head.value))) {
            list.add(head.value);
        }

        head = head.next;
    }
    System.out.println("Anagrams for "+s+":");
    for(String ss:list) {
        System.out.print(ss+"  ");
    }
    System.out.println("\n***********************");
    return list;
}
```

In the method's body we first create a new LinkedList that we call list. Then we call the method getBucketIndex on the passed string and use it as the index of bucketArray, which gets stored in a HashNode we called head.

After that, to be able to check if the words are permutations, we wrote a while loop that loops until head is not null. We have an if statement where we check if the passed String equals head.value, after they have both been normalized.
In that case we add the "head.value" to the list. Then we store head.next in head to see if there are any other words in the chain.

After we get out of the while loop, we print a proper message to output the permutations for the passed String. We use another for loop to iterate over the list and print out everything that was added. In the end we return the list.

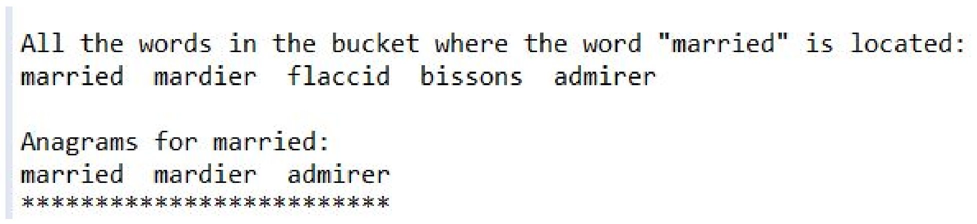
We decided to test our method and passed the word “married” to it. We got the following result.



```
Anagrams for married:
married  mardier  admirer
*************************
```

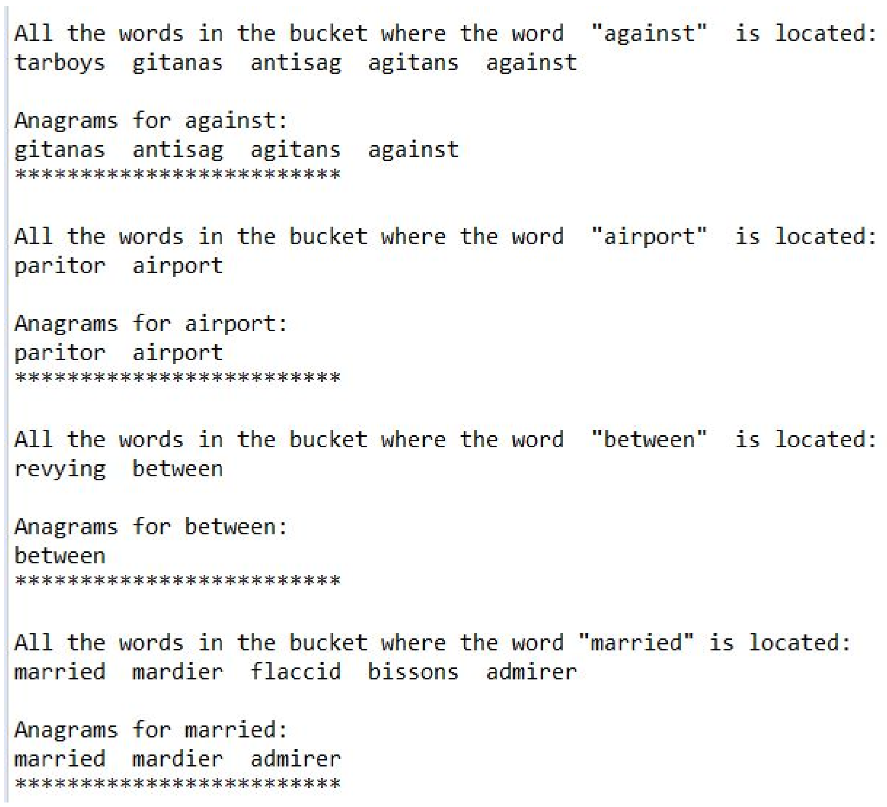
After that we wanted to print out everything together and see the result.

We passed the same String in the method getWordsFromSameBucket and findPermutation.



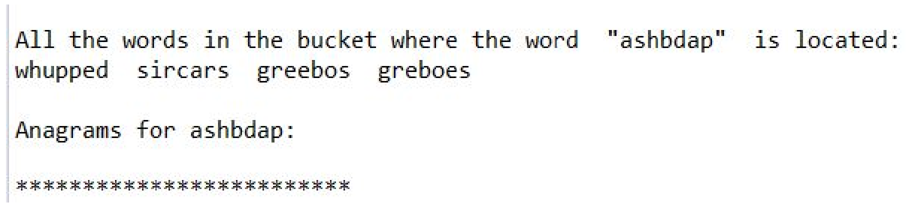
```
All the words in the bucket where the word "married" is located:
married  mardier  flaccid  bissons  admirer

Anagrams for married:
married  mardier  admirer
*************************
```

We also tried with some different words.



```
All the words in the bucket where the word  "against"  is located:
tarboys  gitanas  antisag  agitans  against

Anagrams for against:
gitanas  antisag  agitans  against
*************************

All the words in the bucket where the word  "airport"  is located:
paritor  airport

Anagrams for airport:
paritor  airport
*************************

All the words in the bucket where the word  "between"  is located:
revying  between

Anagrams for between:
between
*************************

All the words in the bucket where the word "married" is located:
married  mardier  flaccid  bissons  admirer

Anagrams for married:
married  mardier  admirer
*************************
```

We decided to try with a String that’s not a known word and got the following result.



```
All the words in the bucket where the word  "ashbdap"  is located:
whupped  sircars  greebos  greboes

Anagrams for ashbdap:

*************************
```

Reflection

Fan:

It is interesting to learn about hashing after learning about different data structures. Now we also need to know why hashmap and hashtable can quickly fetch the data we want even when the total amount stored is huge. We are also already thinking about how we can better improve this program to get a more interesting or useful result. As we find a lot of the times we are also wondering what words like “antisag” (an anagram/permutation we found for “against”) mean, we’d maybe implement definition and store the definition into HashNode and maybe implement a method to look up a word.

Muhammad:

This was one very interesting and fun lab exercise. I have to admit that working in groups of 3 is more fun than in groups of 2. Each one of us had a lot of ideas that we wanted to test, but unfortunately we had too little time to do that. Working with Pavel and Fan was a big pleasure. Great team workers and collaborators. Looking forward to the next lab.

Pavel:

Working on this lab exercise was interesting, because we were able to go step by step about creating the hash table. I was able to see the different results when we were trying out different prime numbers and I saw how the hashing function works. It was nice to play around with the dictionary and see all the different words we were able to find. Showing statistical information about our hash table was also interesting, for example even with a small change to our hash function, the number of collisions in the hash table was sometimes changing by a lot.

Collaboration: https://www.youtube.com/watch?v=jtMwp0FqEcg - Since we had no lecture on hashing before, we had to make a little research and this was just an amazing video that REALLY helped us out to understand the whole idea behind hashing, and made the lecture after our lab more than clear to us.

Code

MyHashTable11

```java
 package lab11_scrabble;

import java.io.FileReader;

import java.io.IOException;

import java.util.Arrays;

import java.util.HashSet;

import java.util.LinkedList;

import java.util.Map;

import java.util.Set;



class MyHashTable11

{

    class HashNode

    {

      int key;
   int empty=0;


      String value;

      HashNode next;



      public HashNode(int hashedKey, String keyStr)

      {

        this.key = hashedKey;

        this.value = keyStr;

      }

    }

   // bucketArray is used to store array of chains

   private HashNode[] bucketArray;

    // Size of array ( prime number )

   static int numBuckets;

    Map<String,Integer> points;

   // Current size of array list

   private int size;



   // Collisions

   int collision = 0;



   // empty buckets

   int empty=0;
```

```
// Collisions with more than 1 elements

   int moreThan1=0;



   // Collisions with more than 16 elements

   int moreThan16=0;



   // Collisions with more than 100 elements

   int moreThan100=0;



   // Finding the longest chain

   int longestChain = 0;



   // Constructor

   // Initializes capacity, size and empty chains.

   public MyHashTable11(int size, String filelocation)

   {

       bucketArray = new HashNode[size];

       numBuckets = size;




       // Create empty chains
       for (int i=0;i<numBuckets;i++)

           bucketArray[i]=null;


       readAndAdd(filelocation);


      System.out.println("Total words in the dictionary with 7 letters: 
"+size());

      System.out.println("How many buckets: "+numBuckets);




      //   System.out.println("Array Size ( the prime number we used ) : " + 
numBuckets);

       System.out.println("Load Factor : "+((float)size())/numBuckets);





       getSizesOfChains();

       System.out.println("The longest chain is: " + longestChain);

       System.out.println("Collisions (more than 1 word hashed to same value): " 
+collision );

       System.out.println("Empty Buckets: " +empty);

       System.out.println("collisions + non-empty Buckets: " +(numBuckets - empty 
+ collision));

       System.out.println("Chains with more than 1 words:" +moreThan1);
```

```
        System.out.println("Chains with more than 16 words:" +moreThan16);

        System.out.println("Chains with more than 100 words:" +moreThan100);



    }



    public int size() { return size; }

    public boolean isEmpty() { return size() == 0; }

     // Method to remove a given key

    public String remove(String key)

    {

        // Apply hash function to find index for given key

        int bucketIndex =  getBucketIndex(key);

         // Get head of chain

        HashNode head = bucketArray[bucketIndex];

         // Search for key in its chain

        HashNode prev = null;

        while (head != null)

        {   // If Key found

            if (head.value.equals(key))

                break;

            // Else keep moving in chain

            prev = head;
            head = head.next;

        }



        // If key was not there

        if (head == null)

            return null;

         // Reduce size

        size--;

         // Remove key

        if (prev != null)

            prev.next = head.next;

        else

            bucketArray[bucketIndex]=head.next;

         return head.value;

    }

     // Returns value for a key

    public String get(String key)

    {

        // Find head of chain for given key

        int bucketIndex =  getBucketIndex(key);

        HashNode head = bucketArray[bucketIndex];

         // Search key in chain
        while (head != null)

        {

            if (head.value.toLowerCase().equals(key.toLowerCase()))

                return head.value;

            head = head.next;

        }

         // If key not found

        return null;

    }


     // Adds a key value pair to hash

    public void add(int hashedKey, String keyStr)

    {

        // Find head of chain for given key

      keyStr = keyStr.toLowerCase();

        int bucketIndex =  getBucketIndex(keyStr);

        HashNode head = bucketArray[bucketIndex];


        // Check if key is already present

        while (head != null)

        {

            if (head.value.equals(keyStr))  return;
            head = head.next;

        }


        // Insert key in chain

        size++;

        head = bucketArray[bucketIndex];


        //If the location already stores something there, collision happens

        if(head!=null)

          collision++;


        HashNode newNode = new HashNode(hashedKey, keyStr);

        newNode.next = head;

        bucketArray[bucketIndex]=newNode;

    }


    // This implements hash function to find index

    // for a key


     public int[] getSizesOfChains() {

      int[] sizes = new int[numBuckets];

      int i=0;
      for(HashNode n: bucketArray) {

        sizes[i] = getSizeOfSingleChain(i);

        i++;

      }

      return sizes;

    }

    public int getSizeOfSingleChain(int index) {

      HashNode current = bucketArray[index];

      int i =0;



      while(current!=null) {

        current = current.next;

        i++;

      }

      if(longestChain<i)// longest chain is initialized 0 as a field

        longestChain = i;

      if(i==0) empty++;

      if(i>2)

        moreThan1++;

      if(i>16) {

        System.out.println("over16: "+i);

        moreThan16++;
      }
      if(i>100) moreThan100++;
      return i;
    }

    public LinkedList<String> findPermutation(String s){
      LinkedList<String> list = new LinkedList<String>();

      HashNode head = bucketArray[ getBucketIndex(s)];
      while(head!=null) {
        if(normalize(s).equals(normalize(head.value))) {
          list.add(head.value);
        }

        head = head.next;
      }
      System.out.println("Anagrams for "+s+":");
      for(String ss:list) {
        System.out.print(ss+"  ");
      }
      System.out.println("\n*************************");
      return list;
```

```
}


// Looks up a single index in the hash table and

// returns the words in it

public LinkedList<String> getWordsFromSameBucket(String s) {


  LinkedList<String> list = new LinkedList<String>();


  HashNode head = bucketArray[ getBucketIndex(s)];

  while(head!=null) {

    System.out.print(head.value+"  ");

    head=head.next;

  }

  return list;

}


// Takes a string and lowercases it.

// Creates a new temp char array out of the string

// Sorts the char array and returns a new string out of it

public String normalize(String s) {

  s=s.toLowerCase();

    char temp[] = s.toCharArray();
```

```
        Arrays.sort(temp);

        return new String(temp);

    } // end normalize


    public void readAndAdd(String fileLocation)

   {

     try{

       File file=new File(fileLocation);    //creates a new file instance

       FileReader fr=new FileReader(file);   //reads the file

       BufferedReader br=new BufferedReader(fr);  //creates a buffering character
input stream

       for(String line="";(line=br.readLine())!=null;){

         if(line.length()==7) {

         line =  line.replaceAll("\\s","");


         String line_norm = normalize(line);

         int k = hashKey(normalize(line_norm));

         add(k,line);



       }

       }

       fr.close();    //closes the stream and release the resources
     }catch(IOException e){

       e.printStackTrace();

     }

   } // end readAndAdd

    public int getBucketIndex(String s) {



      return hashKey(normalize(s));

    }

    // Hashing function. Maps an Integer to

    // an index in the array and returns it

    public int hashKey(String keyStr) {

      long key=0;



      long prime1=94447;

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

        if(key>0 && i>0)

          key = key*(long)(keyStr.charAt(i))*prime1*i % numBuckets;

        else

          key = (long)(keyStr.charAt(i)) % numBuckets;

      }



      return (int) key ;
   } // end hashKey
```

ScrabbleCheater11

```
package lab11_scrabble;


public class ScrabbleCheater11 {


  public ScrabbleCheater11() {


  }

  public static boolean isPrime(int num) {

        if(num<=1)

            return false;

        for(int i=2;i<=Math.sqrt(num);i++){

            if(num%i==0)

                return false;

        }

        return true;

  }
  /*

   *Driver method for the Scrabble Cheater Basic Edition (Lab 11).

   *Instantiates a new Dictionary, HashTable and Scrabble Cheater.

   *Prints statistical information about the HashTable and then

   *looks for words in the same bucket(index) as a given word.

   *In the end prints only the permutations(anagrams) of that word.

   */



   public static void main(String[] args)

      {



  //   ScrabbleCheater11 cheat = new ScrabbleCheater11();

  //   Dictionary11 dict = new Dictionary11(1,"C:\\words\\words-279k.txt");



      MyHashTable11 htable = new
MyHashTable11(7591,"src/lab11_scrabble/wordsList_collins2019.txt");



            System.out.print("All the words in the bucket where the word "+ "
\"against\" " + " is located: ");

            System.out.println();

            htable.getWordsFromSameBucket("against");

            System.out.println();
            System.out.println();

            htable.findPermutation("against");



            System.out.println();

            System.out.print("All the words in the bucket where the word "+ "
\"airport\" " + " is located: ");

            System.out.println();

            htable.getWordsFromSameBucket("airport");

            System.out.println();

            System.out.println();

            htable.findPermutation("airport");



            System.out.println();

            System.out.print("All the words in the bucket where the word "+ "
\"between\" " + " is located: ");

            System.out.println();

            htable.getWordsFromSameBucket("between");

            System.out.println();

            System.out.println();

            htable.findPermutation("between");



              System.out.println();
```

```
             System.out.print("All the words in the bucket where the word "+ 
"\"married\"" + " is located: ");

             System.out.println();htable.getWordsFromSameBucket("married");

             System.out.println();

             System.out.println();

             htable.findPermutation("married");



             System.out.println();

            System.out.print("All the words in the bucket where the word "+ " 
\"ashbdap\" " + " is located: ");

            System.out.println();

            htable.getWordsFromSameBucket("ashbdap");

            System.out.println();

            System.out.println();

            htable.findPermutation("ashbdap");



            //test remove and size()

            System.out.println(htable.remove("speaned"));

            System.out.println(htable.get("speaned"));

            System.out.println(htable.size());

      }

}
```