INFO1105 Assignment 2

In this assignment, you'll be implementing a **hash map** which you'll use to create a password management system. In Section A, you'll implement and compare different implementations of hash maps and in Section B, you'll implement a simple password manager.

Marking

This assignment is worth 12% of your final grade for this course and will be marked out of 12. You will need to make both a code submission to **PASTA** and a written submission to **eLearning**.

Section	*PASTA*	*Hand Marking*	*Report*
Linear probing	1		-
Impact of hash function	1		1
Double hashing	1		1
Chaining	1		1
Password Manager	1		2
Total	5	2	5

Your mark will come from automatic and hand marking of your code and hand marking of your report.
Automatic marking will be performed by **PASTA**, using pre-defined test cases, some of which are hidden. Hand marking of your code will reflect its quality (e.g. overall style, use of comments, selection of variable names) and efficiency (i.e. is this the fastest way to compute the solution?). Your written report will be assessed on the correctness and depth of your discussion. Note that your code and your report will be screened for originality using *Moss* and *Turnitin*.

Submitting

This assignment is due on **Tuesday 13 October, 2015 at 9am**.
You may work and submit individually or in pairs (with a different partner than in assignment 1).

Please submit two things:

- a zip of your `src` directory to "INFO1105 Assignment 2" on PASTA
- a PDF of your written report to the "INFO1105 Assignment 2 – Report" task on eLearning

Please read each section carefully and submit your code to PASTA and report to eLearning as described:

- use the same class and method names as those in the code skeletons provided
- your report should not be longer than 2 A4 pages *including any tables, graphs and references* — if you write more than this, only the first two pages will be marked
- structure your report in sections with names matching the section names used in this handout
- you may write each section as a series of dot points, or in full sentences

Data sets

In this assignment, you'll be testing your implementations on two data sets, which are available to download from eLearning.

Bot IP Addresses

- Use this data set for **Section A** of the assignment.
- **datasetA.txt** is a list of 1800 IP addresses used by spider bots used to crawl the web by search engines
- The original source data for these lists is found here: http://www.iplists.com/nw/
- Additional synthetic data has been attached to each address
- The data is presented in tab separated columns
    1. IP address
    2. Frequency of accesses
    3. Cumulative frequency
    4. Rank (by frequency)

Common Words (passwords)

- Use this data set for **Section B** of the assignment.
- **datasetB.txt** is a list of 10150 alphanumerical words.
- The words were created from a subset of American English words which can be found here **:** http://www.wordfrequency.info/files/entriesWithoutCollocates.txt
- Short words in particular have been padded with digits
- The data is presented in a single column, one password per line

A: Hash Map Implementations

In this section, you will implement and compare three hash maps. Each differs in the method used to resolve collisions. The three methods we will investigate are linear probing, double hashing, and chaining. For all implementations, you may assume that you do not have to perform any resizing.

You will remember from lectures that a hash map is a data structure which stores key-value pairs by using a hash function to map the key of each to an index in a table. In particular, you will use `HashMapNode` to store your key-value pairs and will define a hash function to tell you where in the map to store the node.

For your linear probing and double hashing hash map implementations, nodes should be implemented with class name `HashMapNode` and this should have the following structure.

```
public class HashMapNode<K extends Comparable<K>, V> {

    // construction
    public HashMapNode(K key, V value)

    // get methods
    public K getKey()
    public V getValue()

    // set method
    public void setValue(V newValue)
}
```

For your chaining hash map implementation, nodes should be implemented with class name ChainingHashMapNode and this should additionally store a pointer to the next node in its chain.

```
public class ChainingHashMapNode<K extends Comparable<K>, V> {

    // construction
    public ChainingHashMapNode(K key, V value)

    // get methods
    public K getKey()
    public V getValue()
    public ChainingHashMapNode<K, V> getNext()

    // set methods
    public void setValue(V newValue)
    public void setNext(ChainingHashMapNode<K, V> next)
}
```

Method 1: Linear probing (2 marks)

Code

Your first task is to implement a class called `HashMap`, which stores `HashMapNode` entries and uses **linear probing** to resolve collisions.

The hash function you need to implement is:

$$hash(key) = abs(multiplier \cdot hashCode(key)) \bmod modulus$$

where *abs* is absolute value and *hashCode* is Java's `.hashCode()` method (see Week 8 tutorial).

Hint: to use the output of `hash` as your index into the table, you will need to perform mod `hashMapSize`.

```
public class HashMap<K extends Comparable<K>, V> {

    // construct a HashMap with 4000 places and given hash parameters
    public HashMap(int multiplier, int modulus)

    // construct a HashMap with given capacity and given hash parameters
    public HashMap(int hashMapSize, int multiplier, int modulus)

    // hashing
    public int hash(K key)

    // size (return the number of nodes currently stored in the map)
    public int size()
    public boolean isEmpty()

    // interface methods
    public List<K> keys()
    public V put(K key, V value)
    public V get(K key)
    public V remove(K key)
}
```

Instructions for construction:

- your node store should be initialised an empty array, typed to have elements which are `HashMapNode`
- its capacity will either be the default value of 4000, or given as `hashMapSize` to the constructor
- `multiplier` and `modulus` are your hashing parameters; store these in instance variables so that you can access them when the `hash` method is called

Instructions for the interface methods:

- keys
  - return an array list of the keys of the nodes currently stored in the map
- put
  - insert a node into the map with its key and value set to the objects given
  - if an element already exists with the given key, return the original value and update to the value given
  - return `null` if the key was not already stored
  - collisions should be handled using the linear probing method
- get
  - return the value of the node whose key matches the given argument
  - return `null` if the key is not stored in the map
- remove
  - remove the value of the node whose key matches the given argument and return it
  - place a `defunct` marker in the map in its place
  - return `null` if the key is not stored in the map

Hint: Since we are using a linear probe, put, get, remove may need to scan the cells adjacent to the index given by hash, until either the required node, or an empty cell, is found. If you reach the end of the array storing the nodes, wrap back around to the start.

Impact of hash function on linear probing

In this section, you will explore how your choice of hash function impacts on the performance of your hash map. In particular, you will investigate how often collisions occur and how far your linear probe needs to scan as you change the values of `multiplier`, `modulus`, `hashMapSize`. To do this, add three additional methods to your HashMap which output statistics about the collisions encountered by any call to the put method, and one which resets them.

```
public class HashMap<K extends Comparable<K>, V> {

    ....

    public int putCollisions()

    public int totalCollisions()

    public int maxCollisions()

    public void resetStatistics()
}
```

- putCollisions
  - return the number of calls to the put method which encountered a collision, resulting in the entry not being place in the index indicated by `hash`, but instead one found by moving the linear probe
  - if multiple collisions occur on a single put call, this number should only go up once, **not** once per subsequent collision
  - return 0 if no collisions have occurred
- totalCollisions
  - return the (accrued) number of probing steps needed in put method calls to add entries to the hash map; i.e. increment on each probing step needed within any call to `put`
  - return 0 if no collisions have occurred
- maxCollisions
  - return the maximum number of probing steps that has been needed to add a new entry to the table on any call to `put`
  - return 0 if no collisions have occurred
- resetStatistics
  - reset all three statistics to be zero

Hint: we recommend you use instance variables which you increment during processing to keep track of the relevant statistics and return when these methods are called.

Report

Store the bot IP address data (datasetA.txt) in your hash map, using the parameters presented below, and report on the collision statistics observed. In particular, instantiate hash maps using the given parameters which have `String` keys and `Double` values; these will map each IP address in column 1 of datasetA.txt, to its observed frequency in column 2. To help you, we have provided a code snippet for reading this data file in Appendix A.

In your PDF report, fill in the following table and answer the following questions. If you run further experiments with different settings, please include them in your report.

Settings			Collisions		
`mult`	`mod`	`size`	`put`	`total`	`max`
1	4000	4000			
10	4000	4000			
1	4271	4000			
5	4271	4000			
1	4271	2000			

1. How does the choice of `multiplier`, `modulus`, and `hashMapSize` affect the number of collisions seen and the number of hops the probe needs to make?
2. Can you explain why these trends are observed?
3. Why are these factors important for implementations which will be used by real world applications?
4. What improvements would you suggest to improve the choice of hash function your map relies on?

Method 2: Double Hashing

In this section you will explore performance of using double hashing as an alternative method of handling collisions. You will remember that, in double-hashing, instead of the probe being moved along the store in hops of 1 place, it is instead moved along in hops of size determined by a secondary hash function. Updates to the index are given by the formula $i + j \cdot d(k)$ where i is the output of your primary hash function, $d(k)$ is your secondary hash function applied to the key k, and j increments from 0 until an available cell is found to store the new entry.

Code

Create a new class called `DoubleHashMap` which re-uses the methods you implemented for `HashMap` but instead handles collisions using double hashing. You will need to edit your constructor methods to allow users to specify a `secondaryModulus` and add one further public method, `secondaryHash`, which implements the secondary hash function:

secondaryModulus - (abs(hashCode(key)) mod *secondaryModulus)*

```
public class DoubleHashMap<K extends Comparable<K>, V> {

    // updated construction
    // construct a HashMap with 4000 places and given hash parameters
    public DoubleHashMap(int multiplier, int modulus, int secondaryModulus)

    // construct a HashMap with given capacity and given hash parameters
    public DoubleHashMap(int hashMapSize, int multiplier, int modulus, int
secondaryModulus)

    ....

    public int secondaryHash(K key)
}
```

Note that `maxCollisions` now represents the maximum value j ever reaches in resolving a collision.

Statistics

In certain circumstances, it is possible for this approach to fail to locate an empty position. Add the following statistics method, so you can also include statistics on whether and how often this happens this in your report:

```
    public int putFailures()
```

The `put` method should increment the failure statistic, then throw the following exception.

```
throw new RuntimeException("Double Hashing failed to find a free position");
```

In testing, you should catch that exception and continue the test (after checking that the exception message was the expected one – you don't want to inadvertently exceptions with other causes.) You may use the following snippet of code for this:

```
try {
    //do a single put operation here
} catch(RuntimeException e) {
    if(!e.getMessage().equals("Double Hashing failed to find a free position")) {
        throw e;
    }
}
```

Report

Report on the effect of using double hashing compared to linear probing in storing the bot IP address data (datasetA.txt) by filling in the following table and answering the following questions. If you run further experiments with different settings, please include them in your report.

mult	mod	secMod	size	put	total	max	fails
1	4271	1	2000				
1	4271	223	2000				
1	4271	647	2000				
1	4271	1	4000				
1	4271	223	4000				
1	4271	647	4000				

1. What does the case of `secondaryModulus = 1` correspond to?
2. How does the use of double hashing affect the number of collisions seen and the distance the probe needs to be displaced compared to linear probing?
3. Can you explain why these trends are observed?
4. Were all the keys successfully entered in the map using the above secondary hash function? If not, explain the reason for these failures and discuss how you would modify this double hashing method to prevent them (you do not need to implement these ideas).
5. What do your findings mean for how hash maps should be implemented for real world applications?

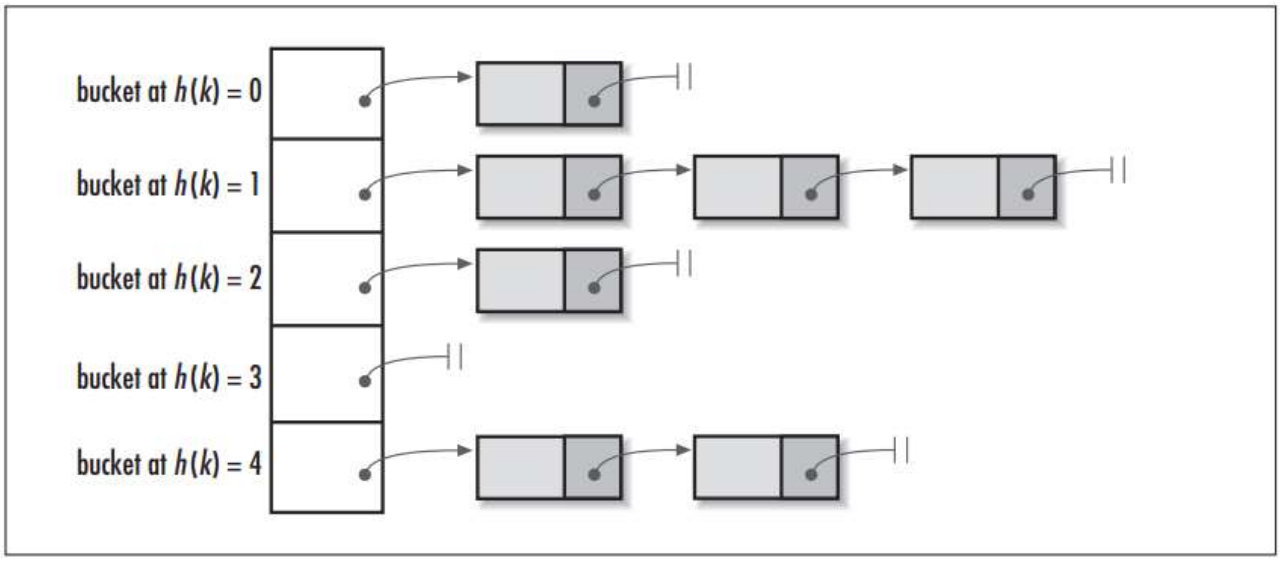
Method 3: Chaining

In this section you will explore the performance of using chaining instead of open addressing in your hash map. In particular, where cells of your hash map have been elements in an array in which store up to one object of type `HashMapNode`, they will now hold one element on type `ChainingHashMapNode`.

Code

Create two new classes, called `ChainingHashMapNode` and `ChainingHashMap`, which borrow your core implementation from `HashMapNode` and `HashMap` with the following changes

- `ChainingHashMapNode` has a pointer to another `ChainingHashMapNode`, and so, acts like a node in a singly linked list
- in place of the collision statistics in `HashMap` (i.e. `putCollisions`, `totalCollisions`, `maxCollisions`, and `resetStatistics`), implement the method `getFullestBuckets` which returns an `int[]` of size two, with the first element being the number of nodes stored in the fullest hash map cell and the second element being the number of hash map cells containing this number of nodes. For instance, `getFullestBuckets` should return {3; 1} given the hash map shown below.




```
public class ChainingHashMap<K extends Comparable<K>, V> {

    // construct a HashMap with 4000 places and given hash parameters
    public ChainingHashMap(int multiplier, int modulus)

    // construct a HashMap with given capacity and given hash parameters
    public ChainingHashMap(int hashMapSize, int multiplier, int modulus)

    // hashing
    public int hash(K key)

    // size (return the number of nodes currently stored in the map)
    public int size()

    public boolean isEmpty()

    // interface
    public int[] getFullestBuckets()
    public List<K> keys()
    public V put(K key, V value)
    public V get(K key)
    public V remove(K key)
}
```

Report

Report on the effect of using a hash map which resolves collisions by chaining instead of using open addressing to store the bot IP address data (datasetA.txt). In particular, fill in the following table and answer the following questions. In your answer to the questions, you should consider all three methods for resolving collisions linear probing, double hashing, and chaining.

`mult`	`mod`	`size`	`getFullestBuckets`
1	4000	4000	
10	4000	4000	
1	4271	4000	
5	4271	4000	
1	4271	2000	

1. How do your quantitative results here relate to those you obtained in your open addressing experiments on linear probing and double hashing?
2. What advantages and disadvantages do you see in the three methods for real world applications?

B Password Manager

You will now implement `SimplePasswordManager`, which stores and manages users' passwords. For security, we do not want to store the actual password string used by each user but, rather, the *hash* of the password string: in the case that our data store is compromised, we don't want nefarious individuals to have free access to user passwords. The result is that, instead of checking at authentication that the password entered is the same as the password stored, we will instead need check that the hash of the password entered is the same as the (hashed) password representation stored.

Report

Secure hash functions for password storage

Do some research to answer the following questions:

1. What properties does a good hash function for storing password data have?
2. Why are each of these properties important for security?
3. Select a hash function designed for storing passwords (e.g. djb2, sdbm, lose lose) and:
    a. give the algorithm used to calculate the hash in **pseudocode**
    b. describe what the algorithm does, **in your own words**

Hints:

- remember to cite all sources you consult in your research
- Wikipedia is not a suitable resource to use
- strong cryptographic hash functions such as SHA-1 can be very difficult to understand and implement; these are considered outside the scope of this assignment

Code

Implement a class called `SimplePasswordManager` which uses a hash map to store user passwords and has a series of methods to manage their use. You may use any one of the hash map implementations from Section A. Your hash map will need to have String keys which are usernames and Long values which reflect the hashed value of that user's password. Since we used a type-safe implementation of HashMap, the key-value types need to be specified to instantiate a HashMap to store users.

```
public class SimplePasswordManager {

    // construct a SimplePasswordManager with 4000 places and default hash parameters
    // multiplier = 1 and modulus = 4271
    public SimplePasswordManager()

    // construct a SimplePasswordManager with the supplied parameters
    public SimplePasswordManager(int size, int multiplier, int modulus)

    // hashing
    public Long hashPassword(String password)

    // interface methods

    // return an array of the usernames of the users currently stored
    public List<String> listUsers()

    public String authenticate(String username, String password)
    public String addNewUser(String username, String password)
    public String deleteUser(String username, String password)
    public String resetPassword(String username, String oldPassword, String newPassword)
}
```

Instructions for hashing:

- implement the hash function you found in your research, that is:
    - you may not import a library which implements hash functions – the implementation must be your own
    - the hash function on `SimplePasswordManager` will be different to the hash function you implemented on your hash maps – the hash function on your hash maps were used to index the entries in the map while this hash function on `SimplePasswordManager` will be used to securely store passwords
    - no test on PASTA will assume a particular hash function, only that `hashPassword` takes a single `String` argument and returns a `Long`

Instruction for the interface methods:

- `authenticate`
  - Check that the hash of the input password is the same as the stored password for the given user.
  - If no user exists in the store with the given username, return "No such user exists."
  - If the hashed password does not matches, return "Failed to authenticate user."
  - Otherwise, return the username
- `addNewUser`
  - Add a new user to your `HashMap` store with the given username and password.
  - If a user with the given username is already stored, just return "User already exists."
  - Otherwise, store the user and password hash, then return the username.
- `deleteUser`
  - Delete the given user from the store of users after authenticating the given password.
  - If no user exists in the store with the given username, return "No such user exists."
  - If the given password fails authentication, print the message "Failed to authenticate user."
  - Otherwise, delete the user and then return the username.
- `resetPassword`
  - Update the stored password for the given user to the (hash of) the `newPassword` after authenticating the `oldPassword`
  - If no user exists in the store with the given username, just return "No such user exists."
  - If the `oldPassword` fails authentication, just return "Failed to authenticate user."
  - Otherwise, update the password hash and then return the username.

Report

Performance of <*name of hash function*>

Report on how well the hash function you chose uniquely identifies passwords. In particular, report how many of the 10150 common passwords in the Password data set (datasetB.txt) have the same value when hashed by your hash function. You may find the code snippet in Appendix B helpful for reading the password file.

Discuss your findings with reference to the following questions:

1. What are the practical implications of having multiple passwords having the same hash representation?
2. What improvements could you make to the hash function you chose to improve security?

Appendix A: Reading the bot IP addresses data set (datasetA.txt)

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

....

    public void exploreData(String pathToFile) throws FileNotFoundException, IOException {
        // instantiate hash maps
        BufferedReader br = new BufferedReader(new FileReader(pathToFile));
        try {
            String line = br.readLine();
            while (line != null) {
                String[] pieces = line.trim().split("\\s+");
                if (pieces.length == 4){
                     // TODO: put data into hash maps
                }
                line = br.readLine();
            }
        } finally {
            br.close();
        }
        // TODO: print collision statistics
    }

....
```

Appendix B: Reading the Password data set (datasetB.txt)

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

....

    public void printHashCollisions(String pathToFile) throws FileNotFoundException,
                                                              IOException {
        HashMap<Long, List<String>> map = new HashMap<Long, List<String>>(50000, 1,
56897);
        SimplePasswordManager spm = new SimplePasswordManager();
        BufferedReader br = new BufferedReader(new FileReader(pathToFile));
        try {
            String line = br.readLine();
            while (line != null) {
                String password = line.trim();
                Long passwordHash = spm.hash(password);
                // TODO: if passwordHash is in a, add password to its list value
                // else, instantiate a new ArrayList and add password to it
                line = br.readLine();
            }
        } finally {
            br.close();
        }
        List<Long> hashes = map.keys();
        for (Long hash : hashes){
            List<String> passwords = map.get(hash);
            if (passwords.size() > 1){
                // all passwords in this list have the same hash representation
            }
        }
    }

....
```