Algorithms for Games Development

Assignment 1 - Open Addressing Collision Handling in Hash Tables

Due Date: 26th February 2023

Provided Framework

To help you get started with the assignment, a significant amount of starter code is provided. You should study this very carefully before proceeding.

First, the `SimpleMap` interface defines the core operations on a Map like data structure - in this assignment we are considering HashMap implementations but there are other possibilities such as Binary Search Trees.

```
public interface SimpleMap<K, V> extends Iterable<K>{
	// Returns the number of key, value pairs in this map
	public int size();
	// Returns true if this map is empty, false otherwise
	public boolean isEmpty();
	// Returns true if the specified key is in this map
	public boolean containsKey(K key);
	// Removes all key, value pairs from this map
	public void clear();
	// Returns the value associated with the specified key
	// if it is present, returns null otherwise
	public V get(K key);
	// Inserts the specified key, value pair into this map.
	// If the key is already present, returns the old value
	// associated with the key if if is present,
	// returns null otherwise
	public V put(K key, V value);
	// Removes the key, value pair with the specified key
	// from this this map
	public V remove(K key);
}
```

Next, the `SimpleHashMap` class implements this interface.

```
public class SimpleHashMap<K, V> implements SimpleMap<K, V>
```

This class is a concrete implementation that fully implements all the operations of the `SimpleMap` interface.

It defines an inner enumeration that specifies constants to represent the three specific collision handling strategies that we are going to consider.

```
public static enum CollisionStrategy{
      LINEAR_PROBING, QUADRATIC_PROBING, DOUBLE_HASHING
}
```

It also provides numerous constructors to create and initialise a HashMap depending on different initial conditions. These constructors are chained together to provide default values if none are specified - the last constructor does all the work.

The defaults are to create a table with an initial capacity of 7, a load factor of 1.0 (i.e. the table is allowed to become completely full before resizing) and to use linear probing as the collision handling strategy.

```
// Create a HashMap with capacity "initCapacity", loadFactor "1.0" and
// collision handling strategy "CollisionStrategy.LINEAR_PROBING"
public SimpleHashMap(int initCapacity)
// Create a HashMap with capacity "7", loadFactor "loadFactor" and
// collision handling strategy "CollisionStrategy.LINEAR_PROBING"
public SimpleHashMap(double loadFactor)
// Create a HashMap with capacity "7", loadFactor "1.0" and
// collision handling strategy "strategy"
public SimpleHashMap(CollisionStrategy strategy)
// Create a HashMap with capacity "initCapacity", loadFactor "1.0" and
// collision handling strategy "strategy"
public SimpleHashMap(int initCapacity, CollisionStrategy strategy)
// Create a HashMap with capacity "initCapacity", loadFactor "1.0" and
// collision handling strategy "CollisionStrategy.LINEAR_PROBING"
public SimpleHashMap(double loadFactor, CollisionStrategy strategy)
// Create a HashMap with capacity "initCapacity", loadFactor "loadFactor"
// and collision handling strategy "CollisionStrategy.LINEAR_PROBING"
public SimpleHashMap(int initCapacity, double loadFactor)
// Create a HashMap with capacity "initCapacity", loadFactor "loadFactor"
// and collision handling strategy "strategy"
public SimpleHashMap(int initCapacity, double loadFactor, CollisionStrategy
strategy)
```

Notes:

1. The default constructor is not available and should not be used.
2. The initial capacity of the table may not actually be the value you pass in - the actual capacity is computed automatically by the provided code to be a prime number (congruent to 3 mod 4) that is at least as large as the value you pass in.
3. You will not be making use of all of these constructors in completing this assignment.

Using these to create a `HashMap` is fairly straightforward if rather clunky (particularly the mechanism to define the collision handling strategy).

Some examples:

```
// Create a HashMap that stores key, values pair where the keys are Strings
// and the values are Integers with an initial capacity of at least 1000, a
// load factor of 1.0 and using the Linear Probing collision strategy
SimpleHashMap<String, Integer> m = new SimpleHashMap<String,Integer>(1000);

// This is equivalent to the above
SimpleHashMap<String, Integer> m = new SimpleHashMap<String,Integer>(1000, 1.0,
SimpleHashMap.CollisionStrategy.LINEAR_PROBING);

// Create a HashMap that stores key, values pair where the keys are Strings
// and the values are Integers with an initial capacity of at least 1000, a
// load factor of 1.0 and using the Quadratic Probing collision strategy
SimpleHashMap<String, Integer> m = new SimpleHashMap<String,Integer>(1000,
SimpleHashMap.CollisionStrategy.QUADRATIC_PROBING);
```

Using the other core operations is equally straightforward:

```
// Inserts the key "Hello" with a value of 2 into the HashMap
m.put("Hello", 2);
// Searches the HashMap for the key "Hello" and returns the associated
// value - in the case the Integer object with the value 2
Integer i = m.get("Hello");
// Removes the key values pair "Hello", 2 from the HashMap
m.remove("Hello");
```

Note: The `remove()` operation has not actually been fully implemented yet and will currently throw an `UnsupportedOperationException`

The `SimpleHashMap` provides additional implemented operations that relate to `HashMap`s rather than general `Map`s that are used similarly to those above. Of particular interest are:

```
// Returns the capacity of the HashMap, as opposed to its size
public int capacity()
// Returns the number of probes from the LAST call to either
// put (insert) or get(search). The number of probes measures the
// number of attempts made to insert or find a key
public int probeCount()
// Returns the current load of the HashMap - this is a measure of
// how full the HashMap is. Computed via size / capacity
public double loadLevel()
```

There is also a method `printTable()` that can be used to print the underlying array that stores the HashMap - this might be useful for debugging when working with `HashMap`s with small capacities.

Internally, the `SimpleHashMap` works by delegating the implementation of some operations (`put()`, `get()` and `remove()`) to subclasses (`LinearProbingHashMap`, `QuadraticProbingHashMap` and `DoubleHashingHashMap`) that determine how each collision handling strategy is actually implemented. These operations (except `remove()`) have been implemented for the `LinearProbingHashMap`. Implementing them for the others is part of the assignment.

Finally, the `SimpleHashMap` class has two other inner classes: A `MapEntry<K, V>` class that encapsulates individual key, value pairs and a `KeyIterator` that is returned by the `SimpleHashMap`'s `iterator()` method.

To iterate over the keys of a `HashMap` and retrieve all values you can use something similar to:

```
for (String s : m){
    int x = m.get(word);
}
```

A small number of sample programs are provided that illustrate the usage of the provided code.

The Assignment

Task 1 - Hand Worked Examples

Simulate, by hand, inserting the keys `14,17,25,37,34,16,26` in order into a table of size 11, showing the table and calculating the number of probes that occur after each insertion for each of the three collision handling methods as described as in the examples above.

If you do this correctly, the final output should be:

```
Linear Probing:          [---, 34,---, 14, 25, 37, 17, 16, 26,---,---]
Quadratic Probing:       [ 16, 34, 26, 14, 25, 37, 17,---,---,---,---]
Double Hashing:          [ 26, 34,---, 14, 37, 16, 17,---,---, 25,---]
```

(actually the output for Double Hashing may differ - it depends on the second hash function that you use)

Task 2 - Complete HashMap Implementations

Implement the `put(K key, V value)` and `get(K key)` methods of the `QuadraticProbingHashMap` and `DoubleHashingHashMap` implementations as described in the lecture notes.

You can use the corresponding operations in the provided `LinearProbingHashMap` implementation as a guide for this. This should not be too difficult - the only real difference is in how the index for the next probe is updated but great care needs to be taken to ensure it is done correctly.

In particular, for Quadratic Probing with alternating signs, watch out for the updated index becoming negative - your code will crash with an exception. To correct this, adding the capacity onto the computed index if it is negative should bring it back into range.

For Double Hashing, care is needed to ensure the second hash function never computes 0 - the code will enter an infinite loop. This can be guarded against by adding 1 to the value computed by the second hash function.

You will probably find it helpful to print out the values of the internal variables for each probe to see what is going on and remove the print statements when all is working.

You will need to test these methods extensively - use your worked example above and other small inputs. Print the hash table after each iteration to see if everything is working. The provided file `tests.txt` contains fully worked output of a number of runs of randomly generated sets of keys (7 keys inserted into a table of size 11) that might be useful - these were generated with the program `Tests.java`

Task 3 - Frequency Counts / Insertion Probe Counts

Using the provided text files (`GreatExpectaions.txt`, `OliverTwist.txt`, `MobyDick.txt` and `TheHoundOfTheBaskerviles.txt`), for each file determine the total number of words and the number of distinct words in each file (convert words to all lower case before inserting them).

Also, for each of the three HashMap implementations determine the average number of probes needed to insert all words into the HashMap.

Use the following delimiters to a `Scanner` object, it will strip out a lot (but not all) punctuation (such as trailing , or ") from words giving a cleaner output:

```
Scanner input = new Scanner(filename);
input.useDelimiter("-{2,}|[^\\p{IsAlphabetic}'-]+");
```

To do this, insert all words from a given file into a `HashMap` one at a time, using the word as the key and the number of times it appears as its value. On each insertion, check if the word is already in the `HashMap`, if it is present, increment its value by 1, otherwise insert it with a value of 1. In either case, increment the total number of probes by the number of probes from the last insertion. Finally, iterate over the entire map, find the word with the greatest value and calculate the average number of probes.

Sample output of one run might look something like below (note: this output was produced using a `LinearProbingHashMap` and a minimum word length of 6, also here I used two maps: one for each file on its own and second one for all files). You will only need to consider one file at a time.

```
File: /home/ekenny/projects/HashMap/src/MobyDick.txt
Size: 14704 Capacity: 28111 Load Factor: 0.5230692611433246
Average no. of probes: 27.98696932940565
Words: 57710 Distinct: 14704
The most common word is "though" appearing 384 times
-------------------------------------------------

File: /home/ekenny/projects/HashMap/src/OliverTwist.txt
Size: 8938 Capacity: 14051 Load Factor: 0.6361113088036439
Average no. of probes: 13.891611664466579
Words: 41665 Distinct: 8938
The most common word is "oliver" appearing 759 times
-------------------------------------------------

File: /home/ekenny/projects/HashMap/src/GreatExpectations.txt
Size: 9091 Capacity: 14051 Load Factor: 0.6470002135079354
Average no. of probes: 15.0273660473092
Words: 42571 Distinct: 9091
The most common word is "little" appearing 371 times
-------------------------------------------------
```

```
File: /home/ekenny/projects/HashMap/src/TheHoundOfTheBaskervilles.txt
Size: 4385 Capacity: 7019 Load Factor: 0.6247328679299046
Average no. of probes: 18.48557857782061
Words: 14527 Distinct: 4385
The most common word is "holmes" appearing 186 times
--------------------------------------------------

All Files
Size: 22360 Capacity: 28111 Load Factor: 0.795418163708157
Average no. of probes: 11.741533683127441
Words: 156473 Distinct: 22360
The most common word is "before" appearing 973 times
```