Data Structures & Algorithms II

Assignment 1 - Open Addressing Collision Handling in Hash Tables

Part 1 - Theory

Hashing

The basic idea behind hashing is to take a field in a record, known as the key, and convert it through some fixed process to a numeric value, known as the **hash key**, which represents the position to either store or find an item in the table. The numeric value will be in the range of `0` to `n-1`, where `n` is the maximum number of slots in the table.

The fixed process to convert a key to a hash key is known as a **hash function**. This function will be used whenever access to the table is needed.

One common method of determining a hash key is the division method of hashing. The formula that is used is:

hash key = key % capacity the table

The division method is generally a reasonable strategy, unless the key happens to have some undesirable properties. For example, if the table size is 10 and all of the keys end in zero. In this case, the choice of hash function and table size needs to be carefully considered. The best table sizes are generally prime numbers.

No matter what the hash function, there is the possibility that two keys could resolve to the same hash key. This situation is known as a **collision**.

When this occurs, there are two straightforward solutions: chaining and open addressing, which is the subject of this assignment

Linear Probing

When using linear probing, the item will be stored in the next available slot in the table, assuming that the table is not already full. This is implemented via a linear search for an empty slot, from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there. So the probe sequence is given by:

```
hash(key) + 1, hash(key) + 2, ...
```

or

```
hash(key) + i,
```

where `i` is the number of unsuccessful probes so far

If an empty slot is not found before reaching the point of collision, the table is full.

A problem with linear probing is that it is possible for blocks of data to form when collisions are resolved. This is known as **primary clustering**. This means that any key that hashes into the cluster will require several attempts to resolve the collision.

```
Linear Probing
[---,---,---,---,---,---,---,---,---,---,---]
[---,---,---, 14,---,---,---,---,---,---,---] Inserting 14  Probes: 1
[---,---,---, 14,---,---, 17,---,---,---,---] Inserting 17  Probes: 1
[---,---,---, 14, 25,---, 17,---,---,---,---] Inserting 25  Probes: 2
[---,---,---, 14, 25, 37, 17,---,---,---,---] Inserting 37  Probes: 2
[---, 34,---, 14, 25, 37, 17,---,---,---,---] Inserting 34  Probes: 1
[---, 34,---, 14, 25, 37, 17, 16,---,---,---] Inserting 16  Probes: 3
[---, 34,---, 14, 25, 37, 17, 16, 26,---,---] Inserting 26  Probes: 5
Inserted 7 values, 7 distinct in 139 ms
Average no. of probes: 2.142857142857143 Longest probe: 5
```

Quadratic Probing

To resolve the primary clustering problem, quadratic probing can be used. With quadratic probing, rather than always moving one spot, move i^2 spots from the point of collision, where `i` is the number of attempts to resolve the collision.

That is we look in slots:

```
hash(key) + 1, hash(key) + 4, hash(key) + 9
```

A major limitation of this scheme is that it there is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime. This means that even if there are empty slots in the table, the probe sequence may not find them.

If the sign of the offset is alternated (e.g. +1, -4, +9, -16 etc.), and if the number of buckets is a prime number `p` congruent to 3 modulo 4 (i.e. one of 3, 7, 11, 19, 23, 31 and so on), a free slot will always be found as long as one exists.

So the probe sequence becomes:

```
hash(key) + 1, hashkey) - 4, hash(key) + 9, hash(key) - 16
```

and so on.

In either case, quadratic probing suffers from **secondary clustering** because elements that hash to the same hash key will always probe the same alternative cells.

```
Quadratic Probing
[---,---,---,---,---,---,---,---,---,---,---]
[---,---,---, 14,---,---,---,---,---,---,---] Inserting 14  Probes: 1
[---,---,---, 14,---,---, 17,---,---,---,---] Inserting 17  Probes: 1
[---,---,---, 14, 25,---, 17,---,---,---,---] Inserting 25  Probes: 3
[---,---,---, 14, 25, 37, 17,---,---,---,---] Inserting 37  Probes: 3
[---, 34,---, 14, 25, 37, 17,---,---,---,---] Inserting 34  Probes: 1
[ 16, 34,---, 14, 25, 37, 17,---,---,---,---] Inserting 16  Probes: 6
[ 16, 34, 26, 14, 25, 37, 17,---,---,---,---] Inserting 26  Probes: 5
Inserted 7 values, 7 distinct in 19 ms
Average no. of probes: 2.857142857142857 Longest probe: 6
```

Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions from the point of collision to insert.

Like linear probing, it uses one hash value as a starting point and then repeatedly steps forward a fixed interval until the desired value is located, an empty location is reached, or the entire table has been searched; but this interval is decided using a second, independent hash function (hence the name double hashing). Unlike linear probing and quadratic probing, the interval depends on the data, so that even values mapping to the same location have different bucket sequences; this minimizes repeated collisions and the effects of clustering.

Here the probe sequence is given by:

```
hash(key) + (i * hash2(key))
```

There are a couple of requirements for the second function; it must never evaluate to 0 (it will never move from the home slot) and it must ensure that all cells in the table can be probed.

A popular second hash function is: `hash2(key) = R - ( key % R )` where `R` is a prime number that is smaller than the size of the table.

The second hash function in my implementation is: `1 + (key % (capacity - 1))`
This gives a potentially fairly long stride which helps space elements out.

```
Double Hashing
[---,---,---,---,---,---,---,---,---,---,---]
[---,---,---, 14,---,---,---,---,---,---,---] Inserting 14  Probes: 1
[---,---,---, 14,---,---, 17,---,---,---,---] Inserting 17  Probes: 1
[---,---,---, 14,---,---, 17,---,---, 25,---] Inserting 25  Probes: 2
[---,---,---, 14, 37,---, 17,---,---, 25,---] Inserting 37  Probes: 1
[---, 34,---, 14, 37,---, 17,---,---, 25,---] Inserting 34  Probes: 1
[---, 34,---, 14, 37, 16, 17,---,---, 25,---] Inserting 16  Probes: 1
[ 26, 34,---, 14, 37, 16, 17,---,---, 25,---] Inserting 26  Probes: 2
Inserted 7 values, 7 distinct in 4 ms
Average no. of probes: 1.2857142857142858 Longest probe: 2
```

Rehashing
Once the hash table gets completely full, the running time for operations will start to take too long and may fail. To solve this problem, a table at least twice the size of the original will be built and the elements will be transferred to the new table.

The new size of the hash table should also be prime and existing keys will reinserted into the table based on the new table size.

This is a very expensive operation! O(N) since there are N elements to rehash and the table size is roughly 2N. This is okay though since it shouldn't happen too often.

The question becomes when should the rehashing be applied? Some possible choices are: when the table becomes half full, once an insertion fails or once a specific load factor has been reached, where load factor is the ratio of the number of elements in the hash table to the table size.

Deletion
The method of deletion depends on the method of insertion. In any of the cases, the same hash function(s) will be used to find the location of the element in the hash table to be deleted.

It is not possible to simply delete elements by marking their location as free - subsequent searches are likely to break. Instead, we leave deleted keys in the table with a marker or flag that they are not valid elements of the table. Now, subsequent searches will not terminate prematurely. When inserting a new key, it overwrites a deleted key.

However, for a large number of deletions and a relatively small number of new insertions, the table can become overloaded with deleted keys, increasing the search time.

Part 2 - 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 initialize a HashMap depending of different initial conditions. These constructors are chained together to provide default values if none are specified - the last construtor 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.

Part 3 - 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 previously.

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 removing 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 to 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
```

Task 4 - Searching (Successful and Unsuccessful) Probe Counts

Next, we are going to examine the effect of searching under various load levels by counting the average number of successful and unsuccessful searches for each implementation under specific loads.

To do this, initialize each `HashMap` to have a size close to the number of distinct words that you found above in Task 3 - this will means the table is large enough to store all the words without needing to be resized / rehashed.

Then, fill each map in increments of 5% - i.e. insert words until the table is about 5% full, 10% full and so on. At the end of each pass, search the map for all the words found in the provided `dictionary.txt`, keeping track of the number of successful and unsuccessful searches and their corresponding probe counts.

Sample output is given below - again this is for all files, you only need to do this for one fill at a time. Try to produce a similar table if you can. The `System.out.printf` statement may be helpful in producing formatted output like this.

```
All Files:
-----------------------------------------------------------
      | Linear Probing | Quadratic Prob.| Double Hashing
------|----------------|----------------|---------------
 Load | Success  Fail  | Success  Fail  | Success   Fail
------|----------------|----------------|---------------
 0.05 |  1.03    1.05  |  1.03    1.05  |  1.02     1.05
 0.10 |  1.05    1.08  |  1.05    1.08  |  1.04     1.08
 0.15 |  1.07    1.12  |  1.07    1.12  |  1.06     1.11
 0.20 |  1.09    1.16  |  1.09    1.15  |  1.08     1.14
 0.25 |  1.11    1.20  |  1.11    1.20  |  1.10     1.18
 0.30 |  1.14    1.25  |  1.14    1.24  |  1.12     1.22
 0.35 |  1.17    1.31  |  1.16    1.29  |  1.14     1.26
 0.40 |  1.20    1.38  |  1.19    1.35  |  1.17     1.31
 0.45 |  1.23    1.46  |  1.22    1.41  |  1.19     1.37
 0.50 |  1.27    1.56  |  1.25    1.48  |  1.22     1.43
 0.55 |  1.32    1.69  |  1.29    1.55  |  1.26     1.49
 0.60 |  1.38    1.85  |  1.33    1.64  |  1.29     1.57
 0.65 |  1.45    2.06  |  1.37    1.75  |  1.33     1.66
 0.70 |  1.52    2.35  |  1.42    1.88  |  1.36     1.77
 0.75 |  1.62    2.78  |  1.47    2.04  |  1.41     1.91
 0.80 |  1.76    3.52  |  1.53    2.25  |  1.46     2.09
 0.85 |  1.95    4.80  |  1.60    2.54  |  1.52     2.33
 0.90 |  2.31    7.83  |  1.69    2.98  |  1.60     2.71
 0.95 |  3.09   19.41  |  1.83    3.90  |  1.72     3.52
 1.00 | 11.57  459.54  |  2.17   64.94  |  2.15    64.49
```

Task 5 - Deletion

Implement the `remove()` operation for the `QuadraticProbingHashMap` and `DoubleHashingHashMap` implementations.

To do this, don't actually delete the keys but simply mark them as removed. There is a variable called `removed` in the inner class `MapEntry<K,V>` of the `SimpleHashMap` class that you can use for this purpose.

This will also require updates the the corresponding `put()` and `get()` operations - searches should ignore deleted entries, they help guide the searches. Insertions should overwrite deleted entries.

Test the effects of deletion by performing a large number of insertions followed by a series of smaller deletions and insertions. Finally, rerun Task 4 over the resulting maps to see what effect this has on the average probe counts for successful and unsuccessful searches under various load levels.

Task 6 - Reporting

The final task is to produce a brief (3-5 page) report summarizing the results of the various tasks in the assignment. Include any hand worked examples and test data that you have produced and the output of the programs you have written.

Describe your implementations of the `put()`, `get()` and `remove()` operations noting in particular the formulas used to update the index to determine the probe sequence used and how these were tested.

Make observations on and draw conclusions from the output of your programs - what can you say about the performance of each of the different collision handling implementations.