QUEENS COLLEGE Department of Computer Science
CSCI 313 Practice problems on Hashing and Hash Tables

Instructor: Alex Ryba

These are some of the solutions to practice problems. Not all problems have solutions here.

Solutions to some older problems might not make use of generics. Generics are now required in this course.

Problem 1 (a) Assume that a chained hash table is implemented with the following instance variables.

```
public class ChainedHashTable<K, V> implements Map<K, V> {
   int size;
   int capacity;
   ArrayList<Pair<K, V>> bucket[];
   // method implementation code omitted
}
```

Write a Java implementation for the method *get*

Answer:

```
public E get(K k) {
   int h = k.hashCode() % capacity;
   if (bucket[h] != null) {
      for (int i = 0; i < bucket[h].size(); i++) {
         Pair<K,V> p = bucket[h].get(i);
 if (p.getKey().equals(k)) return p.getEntry();
      }
   }
   return null;
}
```

(b) Consider these three possibilities for a hash function to be applied to String data (representing words in text). The data is to be stored in a chained table with capacity 20. For each function either describe any reason(s) why it would not work well or state that it is a good choice.

(i) The function converts the first character in the String to lower case and returns its ASCII code modulo 20.

Answer: This is not a good hash function because it does not use the key's full data to calculate its code.

(ii) The function generates a random number x that indexes a position in the String. It then converts character number x in the String to lower case and returns its ASCII code modulo 20.

Answer: This will not even work as a hash function since it will return different codes on different invocations of the method.

(iii) The function converts all characters in the String to lower case and returns the sum of their ASCII codes.

Answer: This is a good hash function.

Problem 2 The generic class ChainedHashTable has the following partial implementation.

```
public class ChainedHashTable<K, V> implements Map<K, V> {
  int size;
  int capacity;
  ArrayList<Pair<K, V>> bucket[];

  ChainedHashTable(int cap) {
    capacity = cap;
    bucket = (ArrayList<Pair<K, V>>[]) new ArrayList[cap];
    size = 0;
    for (int i = 0; i < cap; i++)
      bucket[i] = null;
  }
```

```
  // standard methods omitted

}
```

Give a complete implementation of a standard method called *put*. Assume that the class K has a useful method called *hash* that returns a positive integer result.

```
public void put(K k, E e) {
   int h = k.hashCode() % capacity;
   if (bucket[h] == null) bucket[h] = new ArrayList<>();
   for (int i = 0; i < bucket[h].size(); i++) {
      Pair<K, V> p = bucket[h].get(i);
      if (p.getKey().equals(k)) {
         bucket[h].set(i, new Pair<>(k, e));
         return;
      }
   }
   bucket[h].add(new Pair<>(k, e));
   size++;
}
```

Give a complete implementation of a non-standard method called *loadFactor* that returns the average number of elements in each bucket.

```
public double loadFactor() {
   return size / ((double) capacity);
}
```