CSE 373, Spring 2011
Homework #6: Hashing It Out (40 points)
Due Friday, May 20, 2011, Beginning of Class

In this assignment you practice different hashing concepts including different collision resolution techniques, implementing `hashCode` for different classes, and implementing the Map abstract data type backed by a hash table that uses separate chaining as its collision resolution strategy.

This assignment should be worked on individually. Part A should be turned in on paper in class. Part B should be submitted electronically by submitting a file named `HashMap.java`. For Part B you will also need the additional supporting files: `Map.java` (interface to implement), `HashMap.java` (a partially implemented class to which you should add the `get` and `remove` methods), `HashMapEntry.java` (a class that models a key → value pair that can be inserted into the `HashMap`), and `HashMapClient.java` (a client program containing a couple of testing methods).

This is considered to be a written homework assignment therefore late submissions can be turned in up to the beginning of lecture on Friday, May 20 for a deduction of 20%. Submitting any portion of this assignment late will incur a deduction of 20% on the entire assignment.

Part A: Hashing Concepts

1. Given input {66, 33, 287, 96, 81, 941, 146}, a hash table of size 10, no rehashing, and a hash function $h(x) = x \pmod{10}$, show the resulting:
   a) Separate chaining hash table
   b) Hash table using linear probing
   c) Hash table using quadratic probing
   d) Hash table with second hash function $h_2(x) = 7 - (x \bmod 7)$

2. What are the advantages and disadvantages of the various collision resolution techniques (i.e. separate chaining, linear probing, quadratic probing, and double hashing)?

3. Override the `hashCode` method for the following two classes. Your `hashCode` methods should use the principles discussed in class and found in the Bloch reading which can be found off of the homework page of our course website. For both classes, if two objects are equal their `hashCode`s must be equal. *Hint*: For fields that have a Java object type, take advantage of the `hashCode` methods implemented for those objects by Java when you are constructing your own `hashCode` method. Read the API to discover how `hashCode` is implemented for different Java objects.

```java
public public class BigNum {
    private List<Integer> digits;
    private boolean isNegated;

    ...

    public boolean equals(Object other) {
        if (!(other instanceof BigNum)) {
            return false;
        }

        BigNum o = (BigNum) other;

        if (digits.isEmpty() && o.digits.isEmpty()) {
            return true;
        }
        else {
            return digits.equals(o.digits) && isNegated == o.isNegated;
        }
    }
}


public class ProjectPartners {
    private int year;
    private int quarter; // 0..3 -> AU,WI,SP,SU
    private int prjNum;
    private String student1, student2;

    ...

    public boolean equals(Object other) {
        if (!(other instanceof ProjectPartners)) {
            return false;
        }

        ProjectPartners o = (ProjectPartners)other;

        boolean sameStudents =
            (student1.equals(o.student1) && student2.equals(o.student2))
            || (student1.equals(o.student2) && student2.equals(o.student1));

        return sameStudents && prjNum == o.prjNum
            && quarter == o.quarter && year == o.year;
    }
}
```

Part B: Hash Map Implementation

For this portion of the assignment you will finish implementing the map data structure that we began in lecture.

The goal is to complete the implementation of the instructor-provided `Map` interface, which represents a map of keys into values.

Here are its methods:

```
public interface Map<K, V> {
    public boolean containsKey(K key);
    public V get(K key);
    public void print();
    public void put(K key, V value);
    public V remove(K key);
    public int size();
}
```

In lecture we implemented `containsKey` and `put` and the `print` and `size` methods were already provided.

Methods to Implement:

Your job is to implement the `get` and the `remove` methods. These methods are specified below. Both of these methods should have constant (O(1)) expected runtime assuming rehashing were properly implemented to keep the load factor at an acceptable ratio; you do not need to implement rehashing to ensure this is the case.

`public V get(K key)`

Returns the value mapped to by the given key, if any. If the given key is not contained in this map, returns `null`. Your implementation should correctly fetch the value mapped to a `null` key.

`public V remove(K key)`

Removes the mapping for the given `key` from this map. If `key` is found in your `HashMap`, you should remove the mapping and return the value that was associated with the `key`. If the `key` is not found in your `HashMap`, your `HashMap` should not be altered and `null` should be returned. `null` is a valid `key` value so your method should behave the same way with a `null key` as it would with any other value. In other words, if `null` is passed as the `key` and it is in the `HashMap` you should remove the mapping with the `null key` and return the value that was mapped to `null`; otherwise, your `HashMap` should not be altered and `null` should be returned.