# CSE 373, Winter 2011
# Written Homework #3: Hash it Up! (40 points)

**Step 0: Due Friday, February 25, 2011, Beginning of Class**
**Step 1: Due Friday, February 25, 2011, 10pm**

Step 0 of this assignment you practice different hashing concepts including different collision resolution techniques, rehashing, and writing the `hashCode` method for different classes. Step 0 of the assignment is due in lecture on Friday, February 25, 2011. You may want to read the portion of the Bloch chapter that gives guidelines on how to implement `hashCode` to implement this portion of the assignment.

Step 1 of this assignment gives you a little bit of experience implementing the Map abstract data type with a hash table that uses separate chaining as its collision resolution strategy. On Friday, February 25, turn in Step 1 electronically by submitting a file named `HashMap.java`. 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) and `HashMapEntry.java` (a class that models a key → value pair that can be inserted into the `HashMap`).

## Step 0: Hashing Concepts (Written)

1. (Weiss 5.1) Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989}, an array of size 10, and a hash function h(x) = x (mod 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 h2(x) = 7 – (x mod 7)

2. (Weiss 5.4) A large number of deletions in a separate chaining hash table can cause the table to be fairly empty, which wastes space. In this case, we can rehash to a table half as large. Assume that we rehash to a larger table when there are twice as many elements as the table size. How empty should the table be before we rehash to a smaller table?

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*: Use the `hashCode` method of objects that are a part of the Java API in your `hashCode` methods. Read the API to discover how `hashCode` is implemented for different objects.

```
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;
    }
}
```

## Step 1: Hash table implementation (Programming)

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. Your job is to implement the `get` and the `remove` methods. 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.

***Methods to Implement:***

The methods you are required to implement in `HashMap` are listed now in detail. Both methods should have O(1) expected runtime if rehashing were properly implemented to keep load factor at an acceptable ratio.

`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 work when fetching of the value of 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.