CSE 373 Section Handout #5

1. **HashSet simulation 1**
   Simulate these adds on an initially empty hash set. Include the final hash table, size, and load factor.

   - *hash function:* abs(i) % length
   - *collision resolution:* linear probing
   - *initial hash table length:* 10 (do not perform any re-hashing)

```
int[] a = {35, 2, 35, 15, 80, 42, 95, 15, 66};
Set<Integer> set = new HashSet<Integer>();
for (int n : a) {
    set.add(n);
}
```

2. **HashSet simulation 2**
   Simulate these operations on an initially empty hash set. Include the final hash table, size, and load factor. (Write "X" in any index where an element is removed and not replaced by another element.)

   - *hash function:* abs(i) % length
   - *collision resolution:* linear probing
   - *initial hash table length:* 10
   - *rehashing:* double in size *after* an add, if load factor is ≥ 0.5

```
int[] a = {73,  43, -3, 33, 24, 94, 73, 86, 24, 24, 20, 60};
Set<Integer> set = new HashSet<Integer>();
for (int n : a) {
    set.add(n);
}
set.remove(13);
set.remove(43);
set.add(54);
set.add(-3);
set.remove(94);
```

3. **HashSet simulation 3**
   Simulate the same operations as in the last problem, but use separate chaining instead of linear probing.

4. **HashMap simulation 1**
   Simulate these operations on an initially empty hash **map**. Include the final hash table, size, and load factor.

   - *hash function:* abs(i) % length
   - *collision resolution:* separate chaining
   - *initial hash table length:* 10
   - *rehashing:* double in size *after* an add, if load factor is ≥ 0.5

```
Map<Integer, String> map = new HashMap<Integer, String>();
map.put(22, "a");
map.put(57, "b");
map.put(-52, "c");
map.put(67, "d");
map.put(22, "e");
if (map.containsKey(2)) {
    map.put(99, "f");
}
map.remove(7);
map.put(75, "a");
map.put(75, "a");
map.put(95, "a");
map.put(57, "88");
map.put(42, "g");
map.remove(67);
map.remove(-52);
```

CSE 373 Section Handout #5

5. **HashMap simulation 2**

   Simulate these operations on an initially empty hash **map**. Include the final hash table, size, and load factor.

   - *hash function:* abs(i) % length
   - *collision resolution:* separate chaining
   - *initial hash table length:* **5**
   - *rehashing:* double in size after an add, if load factor is ≥ **0.8**

```
Map<Integer, Integer> map = new HashMap<Integer, Integer>();
map.put(14, 41);
map.put(24, 42);
map.put(89, 98);
map.remove(4);
map.remove(42);
map.put(-9, 9);
map.put(14, 1337);
if (map.containsKey(89)) {
    map.remove(41);
    map.put(66, 98);
}
map.remove(9);
```

6. **`Point hashCode`**

   Suppose we are trying to write a `hashCode` method for a `Point` class with `x` and `y` fields. For each of the following possible implementations: Is it a legal `hashCode` method for a `Point` class, according to the general contract of that method? Does it distribute the hash codes well between objects? Why or why not?

```
a) public int hashCode() {
       return x + y;
   }
```

```
b) public int hashCode() {
       return x * y;
   }
```

```
c) public int hashCode() {
       return 5 * x;
   }
```

```
d) public int hashCode() {
       return 42;
   }
```

```
e) public int hashCode() {
       return (int)(Math.random()*100);
   }
```

```
f) public int hashCode() {
       return super.hashCode() + 7;
   }
```

7. **`Date hashCode`**

   Write a `hashCode` method for a `Date` class, whose fields are a `year`, `month`, and `day` as integers. Follow the general contract for the `hashCode` method, and try to distribute codes well among all legal dates.

```
public class Date {
    private int year;
    private int month;
    private int day;    ...
```

8. **`PurchaseOrder hashCode`**

   Write a `hashCode` method for a `PurchaseOrder` class representing a purchase of an item from an online store, whose fields are a `itemID` string, a `quantity` as an integer, and a price as a real number. Follow the general contract for the `hashCode` method, and try to distribute codes well among accounts.

```
public class PurchaseOrder {
    private String itemID;
    private int quantity;
    private double price;  ...
```

CSE 373 Section Handout #5

For these problems, recall the `HashSet` *class written in lecture with separate chaining:*

```
public class HashSet<E> implements Set<E> {
   private Node[] elements;
   private int size;

   public HashSet() {...}
   public void add(E value) {...}
   public boolean contains(E value) {...}
   public boolean isEmpty() {...}
   public void remove(E value) {...}
   public int size() {...}
   public String toString() {...}
}
```

```
// inside HashSet class
private class Node {
   public E data;
   public Node next;

   public Node(E data) {...}
   public Node(E data, Node next) {...}
}
```

9. `HashSet addAll`

Write a method named `addAll` that could be added to the `HashSet` class from lecture. This method accepts another `HashSet` as a parameter and adds all elements from that set into the current set, if they are not already present. For example, if a set `s1` contains `[a, b, c]` and another set `s2` contains `[a, x, c, z]`, the call of `s1.addAll(s2);` would change `s1` to store `[a, b, c, x, z]` in some order.

You are allowed to call methods on your set and/or the other set. Do not modify the set passed in. This method should run in O(*N*) time where *N* is the number of elements in the parameter set passed in.

10. `HashSet equals`

Write a method named `equals` that could be added to the `HashSet` class from lecture. This method accepts any object reference `o` as a parameter and returns `true` if `o` refers to another `HashSet` object containing exactly the same set of items as this set. It does not matter if the two sets have the same internal hash table length or element ordering in their linked lists, so long as exactly the same overall elements are stored in both sets; no more, no less. For example, if a set `s1` contains `[a, b, c]` and another set `s2` contains `[b, c, a]`, the call of `s1.equals(s2);` would return `true`. If set `s2` instead contained `[b, c, d]` or `[b, c]` or `[b, c, d, a]` or any other contents not exactly equal to `s1`'s set of elements, you would return `false`.

You are allowed to call methods on your set and/or the other set as needed. Do not modify the set passed in. This method should run in O(*N*) time where *N* is the number of elements in either of the two sets.

11. `HashSet hashCode`

Write a method named `hashCode` that could be added to the `HashSet` class from lecture. This method computes and returns a hash code for an entire set. (Yes, that probably seems odd, but a set itself could be added as an element of another set.) To compute your set's hash code, traverse all of its elements and add up the elements' hash codes into one sum and return this sum. Do not "scale up" the elements' hash codes by multiplying them by some multiplier. For example, if a set `s1` contains `[40, -5, 22]`, the call of `s1.hashCode();` would return 40+(-5)+22 = `57`. If a set `s2` contains `[a, b, c]`, and the hash codes of "a", "b", and "c" are 97, 98, and 99 respectively, the call `s2.hashCode();` would return 97+98+99 = `294`.

You are allowed to call methods on your set as needed (but you should not need to). Do not modify the set's state. This method should run in O(*N*) time where *N* is the number of elements in your set.

CSE 373 Section Handout #5
Solutions

1.
```
   0   1   2   3   4   5   6   7   8   9
[ 80,  0,  2, 42,  0, 35, 15, 95, 66,  0]
size = 7
load = 0.7
```

2.
```
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
[ 20, 60,  0, XX, -3, 24, 86,  0,  0,  0,  0,  0,  0, 73, 33, XX, 54,  0,  0,  0]
size = 8
load = 0.4
```

3.
```
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
[  ↓   /   /   ↓   /   /   /   /   /   /   /   /   /   ↓   /   /   /   /   /   /]
  60          -3  24      86                          73  54
   ↓                                                   ↓
  20                                                  33
size = 8
load = 0.4
```

4.
```
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
[  /   /   ↓   /   /   /   /   /   /   /   /   /   /   /   /   ↓   /   ↓   /   /]
          42=g                                              95=a    57=88
           ↓                                                 ↓
          22=e                                              75=a
size = 5
load = 0.25
```

5.
```
   0   1   2   3   4   5   6   7   8   9
[  /   /   /   /   ↓   /   ↓   /   /   ↓]
                  24=42   66=98       -9=9
                   ↓                   ↓
                  14=1337             89=98
size = 5
load = 0.5
```

6.

a) x+y is a valid hash function; it is mediocre at distributing the hash codes; e.g. (3,2), (2,3) conflict.
b) x*y is a valid hash function; it is a bit better at distributing the hash codes than x+y but not great.
c) 5*x is a valid hash function, but it ignores part of the Point's state and distributes codes poorly.
d) 42 is a valid hash function, but it ignores ALL of the Point's state and distributes codes horribly.
e) A random number is NOT a valid hash function. It returns different results for the same state.
f) Object's hashCode+7 ignores all Point state and is based on the object's memory address, so it could return different results for two different objects with the same state. Therefore it is invalid.

CSE 373 Section Handout #5
Solutions, continued

7.

```
// anything that incorporates day, month, and year is acceptable;
// I multiply by 37 because that outpaces the max day (31)
// and 451 because that outpaces the max month*37 (12*37=444)
// and therefore theoretically spreads Dates out very well
public int hashCode() {
    return day + 37 * month + 451 * year;
}
```

8.

```
// anything that incorporates itemID, quantity, and price is acceptable
public int hashCode() {
    return itemID.hashCode()
         + 151 * quantity
         + 1337 * Double.valueOf(price).hashCode();
}
```

9.

```
// Adds all elements from the given other HashSet into this set.
public void addAll(HashSet<E> other) {
    for (Node front : other.elements) {
        Node current = front;
        while (current != null) {
            add(current.data);
            current = current.next;
        }
    }
}
```

10.

```
// Returns true if o refers to another HashSet with the same elements as this set.
public boolean equals(Object o) {
    if (!(o instanceof HashSet)) {
        return false;
    }
    HashSet<E> other = (HashSet<E>) o;
    for (Node front : elements) {
        Node current = front;
        while (current != null) {
            if (!other.contains(current.data)) {
                return false;
            }
            current = current.next;
        }
    }
    return size == other.size();
}
```

11.

```
// Returns a hash code for storing this set as the element of another hash set.
public int hashCode() {
    int code = 0;
    for (Node front : elements) {
        Node current = front;
        while (current != null) {
            code += current.data.hashCode();
            current = current.next;
        }
    }
    return code;
}
```