CS 61B Heaps of Hashing Fall 2017

1 Heaps of fun®

(a) Assume that we have a binary min-heap (smallest value on top) data structure called `Heap` that stores integers and has properly implemented `insert` and `removeMin` methods. Draw the heap and its array representation after each of the operations below:

```
Heap h = new Heap(5); // Creates a min-heap with 5 as the root
h.insert(7);
h.insert(3);
h.insert(1);
h.insert(2);
h.removeMin();
h.removeMin();
```

(b) Consider an array-based min-heap with N elements. What is the worst case running time of each of the following operations if we ignore resizing? What is the worst case running time if we take into account resizing? What are the advantages of using an array-based heap vs. using a node-based heap?

	Ignore Resize	With Resize
Insert	_____	_____
Find Min	_____	_____
Remove Min	_____	_____

(c) You are tasked to implement a max-heap data structure using only a min-heap. Could you complete the task? If so, describe your approach. If not, explain why it's impossible.

2 HashMap Modification (from 61BL SU2010 MT2)

(a) When you modify a key that has been inserted into a `HashMap`, will you be able to retrieve that entry again? Explain.

☐ Always ☐ Sometimes ☐ Never

(b) When you modify a value that has been inserted into a `HashMap`, will you be able to retrieve that entry again? Explain.
☐ Always ☐ Sometimes ☐ Never

3 Hash Functions

(a) Here are three potential implementations of the `Integer`'s `hashCode()` function. Categorize each as either a valid or an invalid hash function. If it is invalid, explain why. If it is valid, point out a flaw/disadvantage.

Note: A "valid" `hashCode()` means that any two `Integer`s that are `.equals()` to each other should also return the same hash code value.

Another note: the `Integer` class extends the `Number` class, a direct subclass of `Object`. The `Number` class' `hashCode()` method directly calls the `Object` class' `hashCode()` method.

```
(1) public int hashCode() {
        return -1;
    }

(2) public int hashCode() {
        return intValue() * intValue();
    }

(3) public int hashCode() {
        return super.hashCode();
    }
```

(b) Suppose that we represent Tic-Tac-Toe boards as 3 by 3 arrays of integers (with each integer in the range 0 to 2 to represent blank, 'X', and 'O' respectively). Describe a hash function for Tic-Tac-Toe boards that are represented in this way such that boards that are not equal will never have the same hash code.

(c) Is it possible to add arbitrarily many `String`s to a Java `HashSet` with no collisions? If not, what is the minimum number of distinct `String`s you need to add to a `HashSet` to guarantee a collision?