CSE 373 Practice Midterm Exam #3
(Section Handout #7)

1. Big-Oh Analysis

Give a tight bound of the runtime complexity class for each of the following code fragments in Big-Oh notation, in terms of the variable *N*.

a)
```
int sum = 0;
for (int i = 0; i < N; i++) {
    if (i == N - 1) {
        for (int j = 0; j < N; j++) {
            sum++;
        }
    }
}
System.out.println(sum);
```

b)
```
Map<Integer, Integer> map =
    new TreeMap<Integer, Integer>();
for (int i = 1; i < N; i++) {
    map.put(i, N * N);
}
map.clear();
System.out.println("done!");
```

c)
```
int sum = 0;
for (int i = 0; i < N; i++) {
    sum++;
}
for (int i = 100*N; i >= 0; i--) {
    sum++;
}
System.out.println(sum);
```

d)
```
List<Integer> list =
        new LinkedList<Integer>();
for (int i = 0; i < N; i++) {
    list.add(i);
}
int sum = 0;
for (int i = 0; i < N; i++) {
    sum += list.get(i);
}
System.out.println("done!");
```

2. Java / Guava Collection Programming

You are writing code for a game show called Family Fracas. Write a method named `rankFamilies` that accepts a Guava `Table` of strings and strings to integers. The rows represent last names and the columns represent first names; the values are the number of points earned by the person with that first and last name. Your method should return a `List` of last names, ordered by how many points were earned by the people in that family (the people with that last name). Families that earned more points should appear earlier in the list. For example:

	Bobby	**Jackie**	**Alex**	**Jordan**	**Mary**
Smith	30		17		67
Chen	75	14			26
Rivera			32	49	25
Mitchell	18		107	3	

According to the table above, Alex Mitchell earned 107 points, Mary Rivera earned 25 points, etc. The Smith family earned 114 points, the Chens earned 115 points, the Riveras earned 106 points, and the Mitchells earned 128 points, so your method will return the list `[Mitchell, Chen, Smith, Rivera]`. You may assume that the `Table` and its elements are not `null` and that no two families will earn exactly the same number of points. Your code should not modify the table passed in. Your method must run in O(*N* log *N*) time or better, where *N* is the number of people in the `Table`.

---

3. Java Class Programming for Collections

Suppose you are given the following class `Car`, representing an automobile:

```
public class Car {
    private String make;        // such as "Toyota"
    private String model;       // such as "Camry"
    private int year;           // such as 2011
    private Color color;

    ...
}
```

Write an `equals` and `hashCode` method for the `Car` class. Two cars should be considered equal if they have the same state: make, model, year, and color. Your hash code function should distribute codes effectively among cars.

Also give `Car` objects a natural ordering by modifying the class to **implement the `Comparable` interface**. Cars should be arranged by make in ABC order (e.g. Honda before Toyota), breaking ties by year (e.g. 2004 before 2011), then breaking ties by model (e.g. Camry before Prius), and finally breaking ties by color. You may assume that `Color` objects implement the `Comparable` interface.

You may assume that the fields' values are not `null`. You may assume that `null` is not passed to your `compareTo` method, though `null` might be passed to `equals` and you should handle it appropriately by returning `false`.

---

4. Hashing

Simulate the behavior of a **hash set** as described in lecture. Assume the following:

- the hash table array has an initial capacity of **10**
- the hash table uses **separate chaining** for collision resolution
- the hash function returns the integer key's value, mod the size of the table
- **rehashing** occurs at the *end* of an add where the load factor is ≥ **0.5** and doubles the capacity of the hash table

Draw an array diagram to show the final state of the hash table after the following operations are performed. Leave a box empty if an array element is `null` or is unused. Also write the size, capacity, and load factor of the final hash table. You do not have to redraw an entirely new hash table after each element is added or removed, but since the final answer depends on every add/remove being done correctly, you may wish to redraw the table at various important stages to help earn partial credit in case of an error.

```
Set<Integer> set = new HashSet<Integer>();
set.add(47);
set.add(97);
set.add(52);
set.add(-34);
set.remove(4);
set.remove(52);
set.add(44);
set.add(77);
set.add(-232);
set.add(-4);
set.add(444);
```

5. Heaps

Given the following string elements:

- Chuck, Sarah, Casey, Morgan, Beckman, Awesome, Ellie, Jeff, Lester

a) Draw the tree representation of the **binary min-heap** that results when all of the above elements are **added** (in the given order) to an initially empty heap. Circle the final tree that results from performing the additions. Also show the final array representation of the heap.

b) After adding all the elements, perform **2 remove-min operations** on the heap. Circle the tree that results after the two elements are removed. Also show the final array representation of the heap.

Please show your work. You do not have to draw an entirely new tree after each element is added or removed, but since the final answer depends on every add/remove being done correctly, you may wish to show the tree at various important stages to help earn partial credit in case of an error.

---

6. AVL Trees

Given the following string elements:

- Drew, Jared, Matt, Alex, Bethy, Sara, Corey, Grace, Brad, Ben

a) Draw the **AVL tree** that results when all of the above elements are added (in the given order) to an initially empty AVL tree.

Please show your work. You do not have to draw an entirely new tree after each element is added or removed, but since the final answer depends on every add/remove being done correctly, you may wish to show the tree at various important stages to help earn partial credit in case of an error.

b) Draw the AVL tree from part (a) after all of the following elements are **removed**:

- Jared, Sara, Matt, Corey

c) Write the **balance factor** of every node of the AVL tree that you drew for part (b), to the right of that node.

---

7. Hash Set Implementation

In lecture, we implemented a class called `HashSet`, a set of elements implemented using a hash table with separate chaining. Assume that the class is implemented in the following way:

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

    private class Node {
        private E data;
        private Node next;
        ...
    }
}
```

Add a method to this class named `removeInRange` that accepts a minimum and maximum value and removes from the set any elements that are between the given minimum and maximum, inclusive. For example, if a hash set in a variable `set` stores `[31, 12, 22, 45, 6, 28, 18, 59]`, after calling `set.removeInRange(10, 30);` , the set should store `[31, 45, 6, 59]`. You should assume that the element type `E` is comparable and cast it to type `Comparable` so that you can determine which elements are in the given range. You should not create any arrays or temporary data structures. This method should run in O(*N*) time, where *N* is the number of total elements stored in the set.
