CSE 373 Practice Midterm Exam #2

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 * 2; i++) {
    for (int j = 0; j < i / 3; j++) {
        for (int k = 0; k < j*j; k+=3) {
            sum++;
        }
    }
}
System.out.println(sum);
```

b)
```
int sum = 0;
for (int i = 1; i < N; i *= 2) {
    for (int j = 1; j < N; j *= 2) {
        sum++;
    }
}
System.out.println(sum);
```

c)
```
Set<Integer> set1 =
        new HashSet<Integer>();
for (int i = 0; i < N; i++) {
    set1.add(i);
}
Set<Integer> set2 =
        new TreeSet<Integer>();
set2.addAll(set1);
System.out.println("done!");
```

d)
```
List<Integer> list =
        new LinkedList<Integer>();
for (int i = 0; i < N; i++) {
    list.add(0, i);
}
Set<Integer> set = new TreeSet<Integer>();
Iterator<Integer> itr = list.iterator();
while (itr.hasNext()) {
    set.add(itr.next());
}
System.out.println("done!");
```

e)
```
List<Integer> list1 =
        new ArrayList<Integer>();
for (int i = 0; i < N; i += 2) {
    list1.add(i);
}
List<Integer> list2 =
        new ArrayList<Integer>();
for (int i = 0; i < N; i++) {
    list2.add(0, list1.remove(0));
}
System.out.println("done!");
```

f)
```
int sum = 0;
for (int i = 0; i < N * 2; i++) {
    for (int j = 0; j < 10000; j++) {
        for (int k = 0; k < j*j; k++) {
            sum++;
        }
    }
}
System.out.println(sum);
```

2. Java / Guava Collection Programming

Write a method named `friends` that checks whether a group of Facebook users are all friends with each other. The method accepts two parameters: a `Multimap` from Facebook user names (strings) to their friends' user names (strings), and a `List` of user names (strings) to check. If all of the users in the list are friends with all of the other users, your method should return `true`. If any user is not friends with any other user, your method should return `false`. For example, if passed the multimap `{Joe=[Bill], Ed=[Joe], Bill=[Joe, Ed, Sue], Sue=[Bill, Joe, Ed]}` and the list `[Sue, Joe, Bill]`, your method would return `false` because Joe is not friends with Bill. If the same collections were passed except Joe's friends were `Joe=[Bill, Sue]`, the method would return `true`. Note that you must check friendship in both directions; e.g. Sue must be friends with Joe, and Joe must be friends with Sue.

Your code should run in no worse than $O(N^2)$ time where N is the number of names in the list. You may assume that the collections passed and their elements are not `null`.

---

3. Java Class Programming for Collections

Suppose you are given the following class `Person`, representing a person in the Stable Marriage simulation:

```
public class Person {
    private String name;
    private String gender;              // either "M" or "F"
    private Person fiancee;             // null if single
    private Queue<String> preferences;

    ...
}
```

Write an `equals` and `hashCode` method for the `Person` class. Two persons should be considered equal if they have the same name, gender, preferences, and fiancee. Your hash code function should distribute codes effectively among persons. Be mindful that your methods should not produce infinite recursion by calling them on other `Person` objects. To get around this, you can compare fiancees using `==` in the `equals` method, and simply incorporate whether the person has a fiancee as a true/false value in your `hashCode`.

Also **write a `Comparator`** that arranges persons by gender (men first, then women), arranging in alphabetical order by name within each gender.

You may assume that the name, gender, and preferences are not `null`, but the fiancee could be. You may assume that `null` is not passed to your `Comparator`, though `null` might be passed to your `equals` method and you should handle it appropriately by returning `false`.

---

4. Hashing

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

- the hash table array has an initial capacity of **5**
- 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 ≥ **0.6** 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.

```
Map<Integer, Integer> map = new HashMap<Integer, Integer>();
map.put(17, 42);
map.put(21, 8);
map.put(2, 8);
map.remove(21);
map.put(31, 17);
if (map.containsKey(8)) {
    map.remove(31);
}
map.put(72, 5);
map.remove(17);
map.put(2, 3);
```

5. Heaps

Given the following integer elements:

- 17, 63, 40, 95, 13, 10, 12, 43, 47, 15, 82

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 integer elements:

- 65, 30, 40, 48, 73, 51, 45, 47, 42, 43, 20, 35, 10

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**:

- 40, 45, 30, 47, 42, 43

---

7. Deque Implementation

In lecture, we implemented a class called `ArrayDeque`, a circular buffer array representing a double-ended queue. Assume that the class is implemented in the following way:

```
public class ArrayDeque<E> implements Deque<E> {
    private E[] elements;
    private int size;
    private int front;

    ...
}
```

Add a method to this class named `reverse` that reverses the order of the elements in the deque. For example, if a deque in a variable `d` stores `[42, 17, -9, 83, 25]`, after calling `d.reverse();` , the deque should store `[25, 83, -9, 17, 42]`. The reversal should be done "in place" without creating any new arrays or temporary data structures. This method should run in O(*N*) time, where *N* is the number of total elements stored in the deque.
