CSE 373 Practice Midterm Exam #1

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 = 1; i <= N + 3; i++) {
    for (int j = 1; j <= N * N; j++) {
        sum++;
    }

    sum++;
}
System.out.println(sum);
```

b)
```
int sum = 0;
for (int i = 1; i <= N - 3; i++) {
    for (int j = 1; j <= i + 4; j += 2) {
        sum++;
    }

    sum++;
}

for (int i = 1; i <= 100; i++) {
    sum++;
}
System.out.println(sum);
```

c)
```
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= 1000; j += 2) {
        sum++;
    }
}

for (int k = -50; k <= -1; k++) {
    sum++;
}
System.out.println(sum);
```

d)
```
Set<Integer> set = new TreeSet<Integer>();
for (int i = 1; i <= N * 2; i++) {
    set.add(i);
}

for (int k : set) {
    System.out.println(k);
}
System.out.println("done!");
```

e)
```
List<Integer> list =
        new ArrayList<Integer>();
for (int i = 1; i <= N; i++) {
    list.add(i);
}

while (!list.isEmpty()) {
    list.remove(0);
}
System.out.println("done!");
```

f)
```
Deque<Integer> deque =
        new ArrayDeque<Integer>();
for (int i = 1; i <= N / 2; i++) {
    deque.addLast(i);
}
for (int i = 1; i <= N / 2; i++) {
    deque.addFirst(i);
}

while (!deque.isEmpty()) {
    deque.removeFirst();
    deque.removeLast();
}
System.out.println("done!");
```

2. Java / Guava Collection Programming

Write a method named `countDuplicateValues` that accepts two `Map`s from strings to integers as parameters and returns an integer count of all duplicate values between the two maps. A duplicate is a value that occurs more than once across the two maps. The first occurrence of a value does not count as a duplicate, but any further occurrences do. For example, if passed the maps `{a=42, b=17, c=29, d=42}` and `{z=31, a=98, e=17, f=42, g=31, h=31}`, your method would return `5` because there are 2 dupes of 42, 1 dupe of 17, and 2 dupes of 31.

Your code should run in O(*N*) time where *N* is the combined number of key/value pairs in the two maps. You may assume that neither map nor any of its keys/values are `null`.

---

3. Java Class Programming for Collections

Suppose you are given the following class `Appointment`, representing an event on a user's calendar:

```
public class Appointment {
    private Date date;
    private Time startTime;
    private int duration;

    ...

}
```

A `Date` object represents a month, day, and year, such as Feb 14, 2013. A `Time` object represents a time of day, such as 11:30 AM. For the purposes of this problem, you don't need to know the exact state or methods inside those classes, except that they both implement the `Comparable` interface, defining a natural ordering where dates/times that occur earlier are considered "less" than ones that are later and are therefore placed earlier in the natural ordering. Both classes also have proper `equals` and `hashCode` methods that you can call as needed.

```
public class Date implements Comparable<Date> { ... }
public class Time implements Comparable<Time> { ... }
```

Write an `equals` and `hashCode` method for the `Appointment` class. Two appointments should be considered equal if they occur on the same date and time and last for exactly the same duration. Your hash code function should distribute codes effectively among appointments. Also make your `Appointment` class **implement the `Comparable` interface**. Appointments should be ordered by date, from earliest to latest; breaking ties by starting time, earliest to latest; and if the dates and times are both the same, breaking ties by duration, from shortest to longest.

You may assume that none of the fields of the `Appointment` object are `null`, though `null` might be passed to your `equals` method and you should handle it appropriately by returning `false`. If null is passed to your `compareTo` method, your code can throw a `NullPointerException` (in other words, you don't need to check for `null` there).

---

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 **linear probing** for collision resolution
- the hash function returns the integer key's value, mod the size of the table, plus any probing needed
- **rehashing** occurs at the end of an add where the load factor is ≥ **0.6** and doubles the capacity of the hash table
- the table uses **lazy removal** and stores a special value "XX" for an array entry that has been removed

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, String> map = new HashMap<Integer, String>();
map.put(7, "Jessica");
map.put(34, "Tyler");
map.put(17, "Ryan");
map.put(15, "Tina");
map.put(34, "Saptarshi");
map.put(7, "Meghan");
map.put(33, "Kona");
map.remove(17);
map.put(6, "Tina");
map.remove(34);
map.put(15, "Daisy");
```

5. Heaps

Given the following integer elements:

- 30, 65, 22, 40, 15, 70, 80, 60, 55, 10

a) Draw the tree representation of the **binary max-heap** that results when all of the above elements are **added** (in the given order) to an initially empty maximum binary 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-max 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:

- 20, 30, 90, 50, 40, 35, 10, 37, 31, 34

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

---

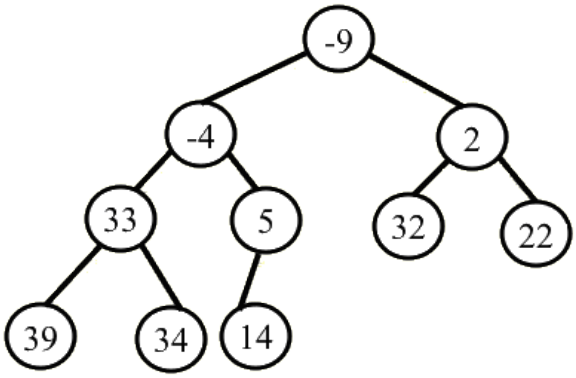
7. Heap Priority Queue Implementation

In lecture, we implemented a class called `HeapPriorityQueue`, a binary min-heap implementation of a priority queue. Add a method to this class called `nodesAtLevel` that accepts as a parameter an integer representing a level in the heap and returns an integer count of the number of values the heap stores at that level, if any. The root level of the heap (i.e. the location of the minimum value) is level 1; the root's direct children are at level 2, and so on. An `IllegalArgumentException` should be thrown if `nodesAtLevel` is called with a level of less than 1, or a level greater than the number of levels in the heap.

This method should run in O(log *N*) time or better, where *N* is the number of total elements stored in the heap.

The table below shows some example calls to your method and the values that would be returned for the given heap:

Call	**Value Returned**
`heap.nodesAtLevel(0)`	`IllegalArgumentException`
`heap.nodesAtLevel(1)`	`1`
`heap.nodesAtLevel(2)`	`2`
`heap.nodesAtLevel(3)`	`4`
`heap.nodesAtLevel(4)`	`3`
`heap.nodesAtLevel(5)`	`IllegalArgumentException`



index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
value	/	-9	-4	2	33	5	32	22	39	34	14	/