CMP 338 (Fall 2011)

Exam 2, 11/10/11

Name (sign) ______________________________

Name (print) ______________________________

email ______________________________

Question	Score
1	20
2	5
3	5
4	5
5	5
6	10
7	10
8	10
9	10
10	10
TOTAL	90

Name:

1) For the following questions:

A is ~ c;
B is ~ c lg N;
C is ~ c N;
D is ~ c N lg N; and
E is ~ c N^2.

a) After N key-value pairs have been inserted in a Binary Search Tree, how many comparisons are required to perform a `get()` operation in the ***worst*** case? **C**

b) After N key-value pairs have been inserted in a Binary Search Tree, how many comparisons are required to perform a `max()` operation in the ***average*** case? **B**

c) After N key-value pairs have been inserted in a 2-3 Tree, how many comparisons are required to perform a `deleteMax()` operation in the ***worst*** case? **B**

d) After N key-value pairs have been inserted in a 2-3 Tree, how many comparisons are required to perform a `select()` operation in the ***average*** case? **B**

e) After N key-value pairs have been inserted in a 2-3 Tree, how many comparisons are required to perform a `isEmpty()` operation in the ***worst*** case? **A**

f) After N key-value pairs have been inserted in a *left-leaning* Red/Black Tree, how many comparisons are required to perform a `keys()` operation in the ***average*** case? **C**

g) After N key-value pairs have been inserted in a *left-leaning* Red/Black Tree, how many comparisons are required to perform a `min()` operation in the ***worst*** case? **B**

h) After N key-value pairs have been inserted in a *left-leaning* Red/Black Tree, how many comparisons are required to perform a `floor()` operation in the ***average*** case? **B**

i) After N key-value pairs have been inserted in a Hash Table, how many comparisons are required to perform a `get()` operation in the ***worst*** case?**C**

j) After N key-value pairs have been inserted in a Hash Table, how many comparisons are required to perform a `contains()` operation in the ***average*** case? **A**

2) A Binary Search Tree has small integer keys between 1 and 10. Searching for a key of 5, comparisons are made with the keys of the nodes in the tree. Which of the following sequences of keys could ***NOT*** have been encountered during the search?

a) 10, 9, 8, 7, 6, 5
b) 2, 6, 9, 4, 5
c) 10, 1, 8, 2, 3, 7, 4, 6, 5
d) 5
e) 1, 2, 8, 4, 5

3) What does the operation **`floor(Key key)`** do on an ordered symbol table?

Returns the largest Key in the symbol table less than or equal to key (or null).

4) In a *left-leaning* Red/Black Tree, what is the best upper-bound on the ratio of the length of the longest path from the root to a leaf to the length of the shortest path from the root to a leaf? 2 : 1

5) The following table, from a published road map, purports to give the length in miles of the shortest routes connecting cities. Correct the error in the table. What kind of graph does this table represent?

An edge-weighted undirected graph.

	Providence	**Westerly**	**New London**	**Norwich**
Providence		53	54	48
Westerly	53		18	~~101~~ 30
New London	54	18		12
Norwich	48	~~101~~ 30	12	

6) Below is (part of) the declaration of a **`BinarySearchTree`** class. Complete the implementation of the **`rank()`** operation for this class.

```
public class BinarySearchTree<Key extends Comparable<Key>, Value> {
    protected class Node {
        protected Key key;
        protected Value val;
        protected Node left;
        protected Node right;
        protected int N;
        Node(Key k, Value v) { ... }
    }
    protected Node root;

    public int rank(Key key) {
        return rank(root, key);
    }

    private int rank(Node n, Key key) {
        if (null == n)
            return 0;
        int cmp = key.compareTo(n.key);
        if (cmp < 0)
            return rank(n.left, key);
        if (0 < cmp)
            return size(n.left) +
                   (null == n.val ? 0 : 1) +
                   rank(n.right, key);
        return size(n.left);
    }
```

7) Below is (part of) the declaration of a **`Part`** class. Complete the implementation of **`hashCode()`** in a way that is consistent with **`equals()`**. Also, implement a **`hash()`** method that maps any **`Part`** to an integer between 0 and 100.

```java
public final class Part {
    final String name;
    final double weight;
    final Part[] subparts;
    public Part (String n, double w, Part[] s) {
        name = n; weight = w; subparts = s; }
    public boolean equals(Object o) {
        if (null == o) return false;
        if (this == o) return true;
        if (this.getClass() != o.getClass()) return false;
        Part p = (Part) o;
        if (!name.equals(p.name)) return false;
        if (weight != p.weight) return false;
        if (subparts.length != p.subparts.length) return false;
        for (int i=0; i<subparts.length; i++) {
            if (!subparts[i].equals(p.subparts[i])) return false;
        }
        return true;
    }
    public int hashCode() {
        int hash = 17;
        hash = 31*hash + name.hashCode();
        hash = 31*hash + ((Double) weight).hashCode();
        for (int i=0; i<subparts.length; i++) {
            hash = 31*hash + subparts[i].hashCode();
        }
        return hash;
    }

    public int hash() {
        return hashCode() & 0x7FFFFFFF % 101;
    }
```

8) Below is (part of) the declaration of a Student class. Complete the implementation of the getSongs() method to return the set of songs that a student might obtain through some chain of friends.

```
public class Student {
    private Set<Student> friends = new HashSet<Student>();
    private Set<Song> songs      = new HashSet<Song>();
    public Student(Set<Student> f, Set<Song> s) {
        friends = f;
        songs   = s;
    }
    public Set<Song> getSongs() {
        Set<Student> visited = new HashSet<Student>();
        return getSongs(visited);
    }
    private Set<Song> getSongs(Set<Student> visited) {
        Set<Song> songs = new HashSet<Song>();
        if (visited.contains(this))
            return songs;
        visited.add(this);
        for (Student s : friends) {
            songs.addAll(s.getSongs(visited));
        }
        return songs;
    }
}
```

Name:

9) The graphs in the following questions have ||V|| vertices and ||E|| edges. Let **a** be ~ c (||E|| + ||V||),

b be ~ c (||E|| + ||V|| lg ||V||),

c be ~ c (||E|| lg ||E||), and

d be ~ c (||E|| ||V||). What are the running times of the following?

a) Depth-first search. **a**

b) Breadth-first search. **a**

c) Prim's minimum spanning tree algorithm. **b**

d) Kruskal's minimum spanning tree algorithm. **c**

e) Dijkstra's shortest path algorithm. **b**

10) Briefly explain how Dijkstra's Shortest Path algorithm works. What is the problem it solves? What are the data structures it relies upon? How are they used?

Dijkstra's Shortest Path algorithm finds the <u>shortest path</u> between a <u>source</u> node **s** and a destination node **d** in an <u>edge-weighted</u> <u>undirected graph</u> with <u>non-negative edge-weighs</u>.

The algorithm repeatedly <u>relaxes</u> the <u>closest</u> unrelaxed node from **s** (starting with **s** itself) until it relaxes **d**. It associates with a node **n**, a non-negative **distance** and a **previous** node. If **previous** is non-null, it is the <u>next to last node on a path from **s** to **n**</u> and **distance** is <u>the length of that path</u>. If **n** has been relaxed, this path is the shortest path from **s** to **n**. Relaxing **n** entails determining, <u>for each neighbor</u> **m** of **n,** if there is a path from **s** to **m** through **n** that is shorter than the path (if there is one) currently encoded in the **distance** and **previous** value for **m**. If so, these values are modified to reflect the shorter path.

A <u>priority queue</u> is used to maintain the distances from s to unrelaxed nodes. A <u>map</u> (HashMap) from nodes to <u>sets</u> (HashSets) of nodes may be used to encode the graph. Maps (from nodes to nodes and to Doubles) may also be used to encode **distance** and **previous**. A set can also be used to keep track of which nodes have been relaxed.