Name: NetID:

Solution to the Final Exam

CS 2110, December 12, 2015, 2:00 PM

	1	**2**	**3**	**4**	**5**	**6**	**Total**
Question	True False	Short Answer	Searching Sorting	Trees	Graphs	Concurrency	
Max	20	20	15	15	23	7	100
Score							
Grader							

The exam is closed book and closed notes. Do not begin until instructed.

You have **150 minutes**. Good luck!

1. True / False (20 points)

a)	T	[✓ F]	During execution of a Java program, the call stack contains at most one frame for each method. It contains a frame for each call that has not completed, so if there is an uncompleted recursive call of a method, it contains at least 2 frames for that method.
b)	T	[✓ F]	A binary tree with at most 7 nodes has at most 3 levels. If all left subtrees are empty, the binary tree has 7 levels.
c)	T	[✓ F]	Local variables of a method can be declared static provided the method is declared static. Local variables belong in the frame for the call and cannot be static.
d)	T	[✓ F]	Java has only three kinds of variable: the field (instance variable), the parameter, and the local variable. It has a fourth, the static or class variable
e)	[✓ T]	F	Think of the *median* of a set of integers as *the value that is in the middle when the set is sorted*. We conclude that in a full and complete BST, the root value is the median.
f)	[✓ T]	F	If the class of object `container` implements `Iterable<T>`, the statement "`for (T e: container) {...}`" can be used.
g)	T	[✓ F]	A JUnit testing class can reference the private members of the classes being tested, without using a getter method. Private variables are referenceable only in the class in which they are declared.
h)	T	[✓ F]	Every directed acyclic graph has a unique topological sort. A 3-node graph with edges n1 -> n2 and n1 -> n3 has 2 topological sorts: (n1, n2, n3) and (n1, n3, n2).
i)	T	[✓ F]	A local variable declared at the beginning of a method maintains its value from one call of the method to the next. Space is allocated for the variable in the frame for the call when the method is called and deallocated when the frame is popped at the end of the call.
j)	T	[✓ F]	In a class `L` that implements a list of integers, the following is a good specification of a method `getFirst()`: `/** Return the first element of the list. */` The spec must say what to do if the list is empty.
k)	T	[✓ F]	The job of a method specification is to explain how the code is implemented. No, the spec should say WHAT the method does and perhaps give runtime and space information.
l)	T	[✓ F]	A method `m` that is overridden in a class `C` can never be called from a method in `C`. super.m(...) calls the overridden method.
m)	[✓ T]	F	Instance methods of a class may access private variables of a nested class declared within it.
n)	T	[✓ F]	The average case complexity of every comparison based sorting algorithm is $O(n \log n)$, where n is the number of elements in the array. Selection sort is $O(n * n)$.
o)	[✓ T]	F	Appending a value to an `ArrayList` and appending a value to the linked list you implemented in A3 have the same complexity.
p)	[✓ T]	F	Prepending a value to an `ArrayList` and adding a value to a `HashSet` have the same worst-case complexity.
q)	[✓ T]	F	Using Kruskal's algorithm on an unconnected graph with n components will produce a forest of n spanning trees.
r)	[✓ T]	F	A spanning tree of a connected undirected graph with n vertices ($n > 0$) has $n - 1$ edges.
s)	T	[✓ F]	Dijkstra's algorithm works correctly on graphs with negative edge weights. If there is a cycle (n1, n2, n3, n1) each edge of which has a negative weight, go around again to get a shorter path (n1, n2, n3, n1, n2, n3, n1). The process would never end.
t)	T	[✓ F]	Suppose a try-statement `try {...} catch (Exception e) {...}` in a method `m` throws an `Exception`. After it is caught and processed by the catch-block, it is thrown out to the place where `m` was called. Termination of the catch-block means termination of the try-statement, meaning the statement after the try-statement is executed next.

2. Short Questions (20 points)

2.a Parsing (4 points)

The following grammar describes numbers in scientific notation. Spaces are not significant. <sci> is the start symbol of the grammar.

```
<nz digit> -->  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digit> -->  0 | <nz digit>
<int>      -->  <digit> | <digit><int>
<exp>      -->  E+<int> | E-<int>
<sci>      -->  <nz digit>.<int><exp> | -<nz digit>.<int><exp> | 0.0
```

Indicate which of the following statements are true and which are false.

(i) The following is a sentence of the grammar: 1.0618E-15 true

(ii) The following is a sentence of the grammar: 9.0 false

(iii) The number of sentences is infinite. true

(iv) This is a <digit>: 01 false

2.b Testing (4 points)

Consider function findMedian. (If b.length is odd, the median is the middle value (if b is sorted). If b.length is even, the median is the average of the two middle values (if b is sorted).)

```
/** Return the median of the numbers in b. Precondition: b.length > 0. */
public static double findMedian(int[] b) {...}
```

Complete procedure testFindMedian below, filling in enough test cases to have reasonable confidence that findMedian is correct.

Solution

```
@Test
public void testFindMedian() {
    assertEquals(1.0, findMedian(new int[]{1}); // Single element
    assertEquals(2.0, findMedian(new int[]{1, 2, 3}); // Odd number, sorted
    assertEquals(2.5, findMedian(new int[]{1, 2, 3, 4}); // Even number, sorted
    assertEquals(2.0, findMedian(new int[]{2, 1, 3}); // Odd number, unsorted
    assertEquals(2.5, findMedian(new int[]{2, 1, 4, 3}); // Even number, unsorted
}
```

2.c Exception handling (4 points)

Function `Integer.parseInt(String s)` returns the int value of the integer that is in `s`. But if `s` does not contain an integer, `parseInt` throws a `NumberFormatException`. Write a statement that stores in variable `ans` the value of the function call `Integer.parseInt(someString)` but stores 1 in `ans` if a `NumberFormatException` is thrown.

Solution

```
int ans;
try {
    ans= Integer.parseInt(someString);
} catch (NumberFormatException e) {
    ans= 1;
}
```

2.d Classes and Interfaces (4 points)

Consider the following definitions and interfaces:

```
interface I1{}
interface I2{}
class A implements I2{}
class B implements I1{}
class C extends B implements I2{}
...
A ob1= ...;
B ob2= ...;
C ob3= ...;
```

Say whether each of the following statement is legal or illegal. Consider each assignment by itself.

(i) `ob2= ob3;` legal

(ii) `ob3= ob2;` illegal

(iii) `I1 b= ob3;` legal

(iv) `I2 c= ob1;` legal

2.e Complexity (5 points)

(i) (2 points) Write the definition of "$f(n)$ is $O(g(n))$".
$f(n)$ is $O(g(n))$ iff there exist $c > 0$ and $N > 0$ such that for all $n > N$, $f(n) \leq c * g(n)$

(ii) (3 points) Complete the following statements (giving the tightest bound on the $O(...)$ complexity):

(1) Worst-case time for quicksort on an array of size n is: $O(n^2)$

(2) Worst-cast time for testing membership in a hash set of size n is: $O(n)$

(3) Expected time for testing membership in a hash set of size n when load factor is 1/2 is: $O(1)$

3. Searching / Sorting (15 points)

3.a Loop invariants (7 points)

Suppose we are given an array segment $b[0..n]$ with $0 \le n$. The array segment is sorted in ascending order but possibly contains duplicates. We want to move the elements around so that all of the values in the original array segment are in array segment $b[k..n]$ with duplicates removed and still in ascending order. For example, given array $b[0..n]$ on the left below, we want to produce the array on the right, where $b[k..n]$ contains the values with duplicates removed. We don't care what is in $b[0..k-1]$.

	0							n		0				k			n
b	2	2	4	4	4	5	8	8		2	2	4	4	2	4	5	8

Below are a precondition, postcondition for the problem and a loop invariant. "Don't care" means we do not care what values are in that segment; "no dups" means "no duplicates".

		0 n
Precondition:	b	original $b[0..n]$

		0	k n
Postcondition:	b	don't care	original $b[0..n]$, no dups

		0 t		k n
Invariant:	b	original $b[0..t]$	don't care	original $b[t+1..n]$, no dups

Below, write a loop with initialization for this problem. Your grade depends on how well you use the given loop invariant and the four loopy questions. Hint: $b[0..n]$ contains at least one value, so think of starting with $b[k..n]$ containing one value.

Solution

```
t= n - 1;
k= n;
while (t != -1) {
    if (b[t] != b[k]) {
        k= k - 1;
        b[k]= b[t];
    }
    t= t - 1;
}
```

3.b Selection Sort (2 points)

Write the invariant for `selectionSort` method.

Solution

```
/** Sort b --put its elements in ascending order. */
public static void selectionSort(Comparable[] b) {
    // Invariant: b[0..k-1] is sorted and b[0..k-1] <= b[k..b.length-1]
    ....
}
```

3.c Merge Sort (6 points)

The code for merge sort is given below. It contains errors. Fix the errors. Assume that method `merge` has the specification shown after the method.

Solution

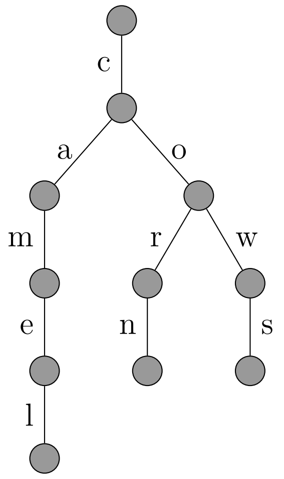
The corrected merge sort is:

```
/** Sort b[h..k]. */
public static void mergeSort(Comparable[] b, int h, int k) {
    if (k+1 - h < 2) return;
    int e= (h + k) / 2;
    mergeSort(b, h, e)
    mergeSort(b, e + 1, k)
    merge(b, h, e, k);
}


/** b[h..j] and b[j+1..k] are sorted.
  *  Merge them together so that b[h..k] is sorted. */
public static void merge(int[] b, int h, int j, int k) {...}
```

4. Trees (15 points)

A *trie* is a tree data structure that can be used to implement a set of `String` values. Each node in a *trie* may have multiple children, which are unordered, and an edge between a given parent and child is annotated with a character. Each element in the set can be found by reading off the characters on some path going from the root to a leaf. For example, here is a trie containing three strings: “camel”, “corn”, and “cows”

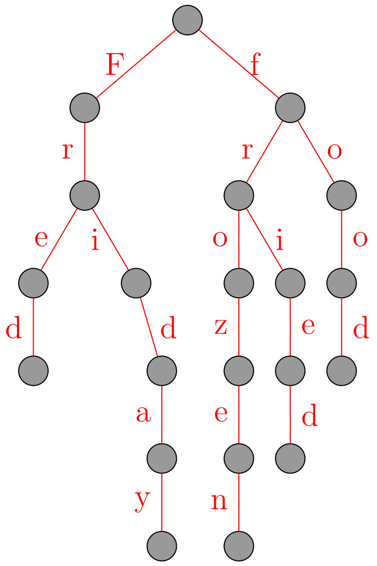


4.a Warmup (5 points)

Draw a trie containing the following entries (note that 'F' and 'f' are different characters):

- "Fred"
- "fried"
- "frozen"
- "food"
- "Friday"

Solution



4.b Implementation (10 points)

In real life, "for" and "form" are both words, which would be allowed in a trie. However, just to keep things simple, in our implementation, only strings given by a path from the root to a leaf are considered to be in the set. Consider the following partial implementation of a trie:

```
/* An instance is the root of a trie that represents a set of strings */
public class Trie {
    /* A key-value pair (c, t) represents the edge labeled with
     * character c from this trie to trie t. */
    private Map<Character, Trie> children;

    /** Constructor: an empty set */
    public Trie() {
        children= new HashMap<Character, Trie>();
    }
   ...
}
```

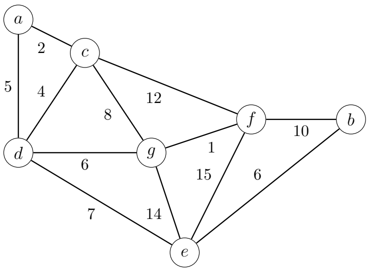
Complete the definition of method `insert`, which of courses goes in class `Trie`. *Hint: use recursion*

Solution

```
/** Insert s into this trie.
  * Precondition: s is not a proper substring of some string in the trie. */
public void insert(String s) {
    if (s.length() == 0) return;
    char c= s.charAt(0);
    Trie child= children.get(c);
     if (child == null) {
        child= new Trie();
        children.put(c, child);
     }
     child.insert(s.substring(1));
}
```

5. Graphs (23 points)

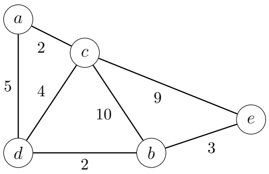
(a) 5 points Consider the following graph:



List the edges of the graph above in the order they would be added to a minimum spanning tree by Prim's algorithm. If a starting node is required, start with node a. An edge should be listed by the pair of vertices it joins. For example, the edge with weight 15 above would be edge (e, f)

Prim's algorithm would add the edges in this order: $(a, c), (c, d), (d, g), (g, f), (d, e), (e, b)$

(b) 7 points Consider the following graph:



Execute Dijkstra's shortest-path algorithm on the graph above with a as the start node. We show the initial state, before iteration 0 of the main loop, in the second column below, giving the frontier set (the blue set in the description of Dijkstra's algorithm given in lecture), and the L value for each node so far.

For each iteration 0, 1, 2, ..., first write in the appropriate place in the table the Selected node, that is, the node that gets removed from the frontier set. Below that, write the new value of the frontier set and the L values that change on this iteration.

Selected Node	N/A	a	c	d	b	e	N/A
Frontier Set	$\{a\}$	$\{c, d\}$	$\{d, e, b\}$	$\{b, e\}$	$\{e\}$	$\emptyset$	$\emptyset$
$L[a]$	0	0	0	0	0	0	0
$L[b]$	∞	∞	12	7	7	7	7
$L[c]$	∞	2	2	2	2	2	2
$L[d]$	∞	5	5	5	5	5	5
$L[e]$	∞	∞	11	11	10	10	10
Iteration	Init	0	1	2	3	4	Final

(c) 11 points Consider the following class representing a node of an undirected graph that has an infinite number of nodes.

```
class Node {
    Set<Node> neighbors; // Neighbors of this node
    boolean hasTreasure; // true iff this Node has a treasure on it
}
```

Complete function `findTreasure` below according to its specification. "REACHABLE" means reachable along a path of unvisited nodes.

Solution

```
/** Return a node REACHABLE from n that has a treasure on it.
  * Precondition: at least one node REACHABLE from n has a treasure on it
  * and that node is a finite distance from n */
public static Node findTreasure(Node n) {
    HashSet<Node> visited= new HashSet<Node>(); // Nodes that have been visited
    LinkedList<Node> queue= new LinkedList<Node>();
    queue.add(n);
    // inv: a node with a treasure is REACHABLE from some
    //        unvisited node in queue
    while (!queue.isEmpty()) {
        Node n= queue.poll();
        if (n.hasTreasure) return n;
        if (!visited.contains(n)) {
            visited.add(n);
            for (Node neighbor : n.neighbors) {
                queue.add(neighbor);
            }
        }
    }
    return null; // Never executed, keeps Java happy
}
```

6. Concurrency (7 points)

This question deals with the bounded buffer problem, but pared down to the bare minimum. Consumers (each a thread) can get an OK to do something by calling function `consume` (see below). A system is introduced to be able to slow down the rate at which OKs are given. A bunch of producers (threads) call procedure `produce` when they want; each call on `produce` allows one more consumer to get an OK. At any time, at most 10 consumers can get an OK.

```
class OK {
    int n; // 0 <= n <= 10. The number of consumers who may get an OK.
           // n is increased by producers, decreased by consumers

    /** increase the number of consumers who can get an OK --but
      don't let it get over 10 */
    public synchronized void produce() {
        while (n == 10) wait();   // wait until n < 10
        n= n+1;
        notifyAll(); // Signal to all that an OK may be granted
    }

    /** return "OK"  --but wait until it is allowed */
    public synchronized String consume() {
        if (n == 0) wait(); // wait until n != 0
        n= n-1;
        notifyAll(); // Signal to all that an "OK" has been granted: n < 10
        return "OK";
    }
}
```

(a) 3 points Unfortunately, the code is not thread safe, and a bunch of calls involving just 2 consumers, c_1 and c_2, and one producer, p, can lead to n becoming negative. Describe the sequence of calls and show why n becomes negative.

c_1 calls `consume`, n is 0 so c_1 waits. c_2 calls `consume`, n is 0 so c_2 waits. p calls `produce` which sets n to 1 and wakes up c_1 and c_2. c_1 finishes `consume` which sets n to 0. c_2 finishes `consume` which sets n to -1.

(b) 4 points Fix class `OK` to eliminate the error you found in (a).

Solution

class `OK` is the same as above except `consume` is replaced with:

```
public synchronized String consume() {
    while (n == 0) wait(); // wait until n != 0
    n= n-1;
    notifyAll(); // Signal to all that an "OK" has been granted: n < 10
    return "OK";
}
```