Name: NetID:

Prelim 2 Solutions

CS 2110, November 20, 2014, 5:30 PM

	1	**2**	**3**	**4**	**5**	**Extra**	**Total**
Question	True/False	Short Answer	Complexity Induction	Trees	Graphs	Extra Credit	
Max	20	10	15	25	30	5	100
Score							
Grader							

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

You have **90 minutes**. Good luck!

Write your name and Cornell **netid** at the top of EACH page! There are 5 questions on 12 numbered pages, front and back. Check that you have all the pages. When you hand in your exam, make sure your booklet is still stapled together. If not, please use our stapler to reattach all your pages!

We have scrap paper available. If you do a lot of crossing out and rewriting, you might want to write code on scrap paper first and then copy it to the exam, so that we can make sense of what you handed in.

Write your answers in the space provided. Ambiguous answers will be considered incorrect. You should be able to fit your answers easily into the space provided.

In some places, we have abbreviated or condensed code to reduce the number of pages that must be printed for the exam. In others, code has been obfuscated to make the problem more difficult. This does not mean that its good style.

Name: ______________________________ NetID: ____________

1. True / False (20 points)

a)	[✓ T]	F	On a connected unweighted graph, Breadth First Search is guaranteed to find the shortest path to any given node from a defined start node.
b)	T	[✓ F]	It will always take $O(\log n)$ time to find an arbitrary node in a binary search tree of size n. [It depends on the shape of the binary search tree –could have all left links empty (null) and then it is $O(n)$]
c)	[✓ T]	F	Even though `String` is a subclass of `Object`, `LinkedList<String>` is not a subtype of `LinkedList<Object>`. [Slide 6 of lecture 13 explains why.]
d)	[✓ T]	F	A `HashMap` has expected lookup cost $O(1)$ and worst case lookup cost $O(n)$.
e)	[✓ T]	F	No comparison-based sorting algorithm can achieve worst-case complexity better than $O(n \log n)$.
f)	T	[✓ F]	The tightest bound on worst-case complexity of removing an item from a heap containing n items is $O(1)$. [It's $O(\log n)$ because something has to be bubbled down.]
g)	T	[✓ F]	The tightest bound on the worst-case complexity of inserting an item into a heap containing n items is $O(n)$. [It's $O(\log n)$.]
h)	[✓ T]	F	A "`for (T elem:  container)`" statement can be used to iterate over the members of container if container in an instance of `Iterable<T>`.
i)	T	[✓ F]	A method that takes $O(n \log n)$ will always compute a result faster than an $O(n^2)$ method, assuming n has the same value for each method. [An example is mergesort vs quicksort. quicksort is $O(n^2)$ in worst case but it is used because it is faster than mergesort.]
j)	T	[✓ F]	A `hashCode()` method must return a prime integer larger than 7.
k)	T	[✓ F]	If a method calls a procedure that requires $O(n \log n)$ operations once and calls another procedure that requires n operations $\frac{n}{2}$ times, the complexity of the method is $O(n \log n)$. [It is $O(n^2)$].
l)	[✓ T]	F	If you don't provide a different hash function, inserting a `String` of length l into a `HashSet` of size n takes $O(l)$ time.[It takes $O(l)$ time to compute the hashCode for the String]
m)	[✓ T]	F	If an undirected connected graph has n nodes, the minimum spanning tree will have $n-1$ edges and no cycles.
n)	[✓ T]	F	On an unweighted graph Breadth First Search will visit all vertices at distance d from the start node before any vertex at distance $d+1$.
o)	T	[✓ F]	On the same graph, Prim's and Kruskal's algorithm will always return the exact same minimum spanning tree. [Suppose all edge weights are 1. Then there may be many minimum spanning trees, and they could construct different ones.]
p)	T	[✓ F]	Breadth-first search can be easily implemented recursively because it maintains a stack of nodes to be visited. [BFS uses a queues, not a stack.]
q)	[✓ T]	F	In a valid coloring of a graph, only non-adjacent vertices can have the same color.
r)	T	[✓ F]	An inner class cannot access private instance members of the surrounding class.
s)	T	[✓ F]	When a graph is stored as an adjacency-list, checking if there is an edge between two vertices takes $O(1)$ time in the worst case. [It is $O(n)$ for a complete graph of n nodes because the adjacency list for a node will have length $O(n)$].
t)	[✓ T]	F	If a graph has no cycles of odd length, then it is always 2 colorable.

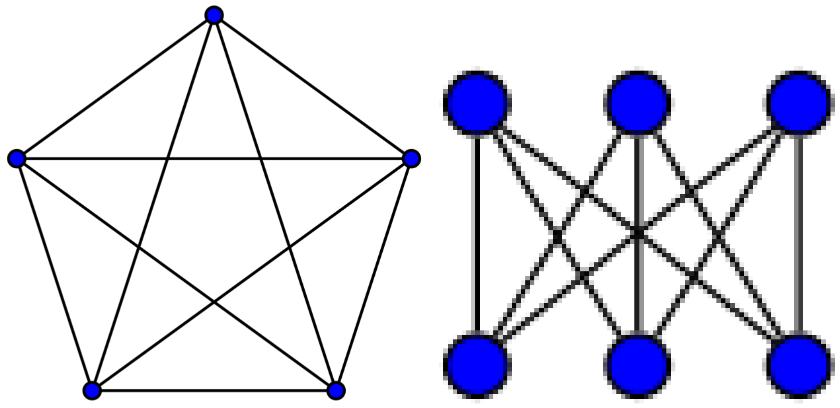
2. Short Answer (10 points)

(a) 2 points If a heap is implemented by storing the elements in a Java array, the left child of the node at index k will be at what index?
$2k + 1$

(b) 2 points If you wanted to write your own exception class, what Java class could you extend?
`java.lang.Exception`

(c) 3 points In a binary tree of depth d, what is the maximum and minimum number of nodes it could have? Minimum: $d + 1$, Maximum $2^{d+1} - 1$

(d) 3 points Draw a non-planar graph of minimal size.
Either of the two graphs below are correct.



3. Complexity and Induction (15 points)

(a) 3 points What is the tightest bound on the time complexity of the following function? **Be careful! This requires you to know the complexity of all operations and function calls below.**

```
/** Return the sum of the values in list. */
public static int sumLinkedList(LinkedList<Integer> list) {
    int sum = 0;
    for (int i = 0; i < list.size(); i= i+1){
        sum = sum + list.get(i);
    }
    return sum;
}
```

$O(n^2)$, note that `list.get(i)` is an $O(n)$ operation

(b) 4 points Can the function body in part (a) be rewritten to improve the time complexity —perhaps using a different loop? If so, rewrite it below and state what the best possible time complexity is. If not, explain why no improvement over your answer from part (a) is possible.

You can improve the time complexity to $O(n)$ by replacing the loop with an enhanced for loop. `for (Integer i :  list) sum= sum + i;`

(c) 8 points Let $P(n)$ be: $9^n - 4^n$ is divisible by 5. Prove by induction that $P(n)$ holds for all integers $n \geq 0$.

(i) **2 points** Write the base case(s) for your proof.
$P(0)$ is that $9^0 - 4^0 = 0$ is divisible by 5, which is clearly true

(ii) **3 points** Write the inductive hypothesis for your proof.
We assume that $P(k)$ is true. That is, that $9^k - 4^k$ is divisible by 5

(iii) **3 points** Complete the inductive step for your proof.
We will show that $P(k) \Rightarrow P(k+1)$

$$9^{k+1} - 4^{k+1} =$$

$$9 * 9^k - 4 * 4^k =$$

$$9 * 9^k - 9 * 4^k + 9 * 4^k - 4 * 4^k =$$

$$9(9^k - 4^k) + 5 * 4^k$$

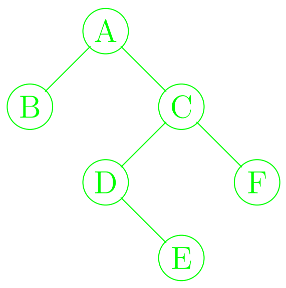
Clearly, $5 * 4^k$ is divisible by 5. By $P(k)$ $9^k - 4^k$ is divisible by 5, therefore $9 * (9^k - 4^k)$ must also be divisible by 5. Since the sum of two numbers divisible by 5 is also divisible by 5, we have shown that $9 * (9^k - 4^k) + 5 * 4^k = 9^{k+1} - 4^{k+1}$ is divisible by 5. Therefore, we have shown that $P(k) \Rightarrow P(k+1)$

4. Trees (25 points)

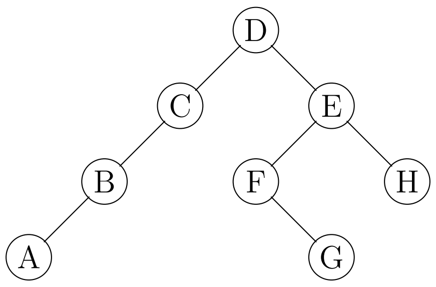
(a) 3 points Given the following pre-order and in-order tree traversals, draw the tree.

Pre-order	A, B, C, D, E, F
In-order	B, A, D, E, C, F

The correct tree is shown below



(b) 3 points Given the following tree, state the post-order tree traversal.



The post order traversal is: A, B, C, G, F, H, E, D

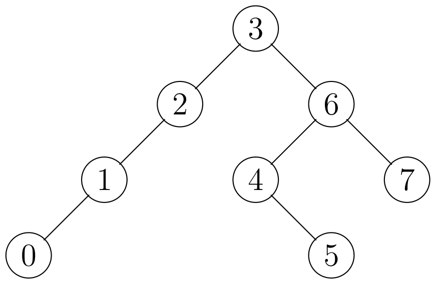
(c) 19 points The following questions pertain to class `TreeNode` defined below:

```
public class TreeNode {
    public int value;
    public TreeNode left;
    public TreeNode right;
    public int size; // The number of nodes in this tree, including this node
}
```

(i) **7 points** Complete method `initSize` according to its specification:
Solution

```
/** Initialize field size in t and in all of its descendants.
  * Return the size of t. */
public static int initSize(TreeNode t) {
    if (t == null) {
        return 0;
    }
    t.size = initSize(t.left) + initSize(t.right) + 1;
    return t.size;
}
```

(ii) **12 points** The inorder traversal of a tree visits the nodes in a certain order. An example is given in the tree below, where the number in each node is the node's inorder-index. Note that if tree `t` has 3 nodes in its left subtree, its root is numbered 3. *Note: The values of each node may be different from its label. The tree is not necessarily a Binary Search Tree.*



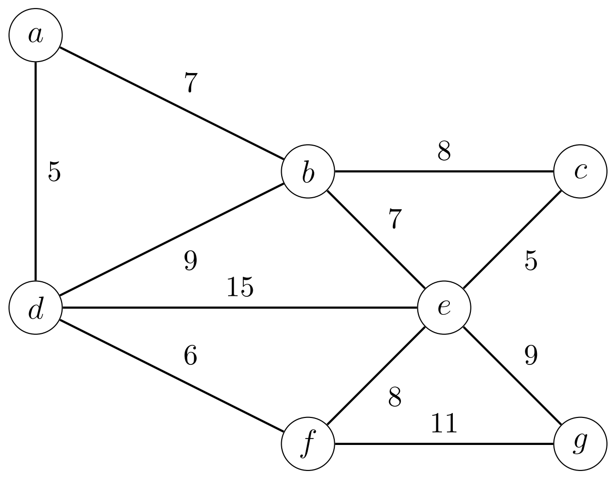
Complete method `findN` according to its specification. There is no need to check the preconditions; assume they are true. You may and should use field `size` from the previous question. **Your solution must run in time $O(h)$ for full credit, where h is the height of the tree.**

Solution

```
/** Return node number n of the inorder traversal of tree t.
  * Precondition: t != null && n >= 0 && n < t.size
  * and field size is initialized in t and all its descendants. */
public static TreeNode findN(TreeNode t, int n) {
    int leftSize = t.left != null ? t.left.size : 0;
    if (leftSize == n) {
        return t;
    }
    if (n < leftSize) {
        // t.left != null if n >= 0
        return findN(t.left, n);
    }
    // t.right != null if n < t.size
    return findN(t.right, n - (leftSize + 1));
}
```

5. Graphs (30 points)

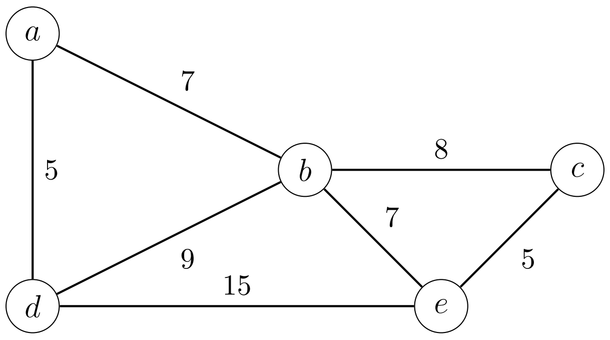
(a) 5 points Consider the following graph:



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 (d, e)

The following edges would be added to a minimum spanning tree in the order listed by Prim's algorithm: $(a, d), (d, f), (a, b), (b, e), (e, c), (e, g)$

(b) 10 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.

The correct answer is given below. Note that all L values are filled in. On the exam you were only required to write the values that changed.

Node label	Distance						
Selected Node	N/A	a	d	b	e	c	N/A
Frontier Set	$\{a\}$	$\{b, d\}$	$\{b, e\}$	$\{c, e\}$	$\{c\}$	$\emptyset$	$\emptyset$
$L[a]$	0	0	0	0	0	0	0
$L[b]$	∞	7	7	7	7	7	7
$L[c]$	∞	∞	∞	15	15	15	15
$L[d]$	∞	5	5	5	5	5	5
$L[e]$	∞	∞	20	14	14	14	14
Iteration	Init	0	1	2	3	4	Final

(c) 15 points Consider the following class representing a node in an undirected graph.

```
public class Node {
    public HashSet<Node> neighbors;
}
```

Let n be a node of a graph g. The connected component of g that contains n is the subgraph contaning all nodes (and their edges) that are connected to n by some path. The graph could consist of a bunch of connected components, none of which are connected to each other.

Complete functions `numComponents` and `dfs` below according to their specifications.

Solution

```
/** Return the number of connected components in graph g. */
public static int numComponents(HashSet<Node> g) {
    HashSet<Node> visited= new HashSet<Node>();
    // Put declarations of local variables and initialization here
    int components= 0;
    for ( Node n : g) {
        if (!visited.contains(n)){
            dfs(n, visited);
            components= components + 1;
        }
    }
    // Put your return statement here
    return components;
}

/** Precondition: n is not in visited. Add every Node reachable from n
     along paths of unvisited nodes to visited. */
public static void dfs(Node n, HashSet<Node> visited) {
    visited.add(n);
    for (Node neighbor : n.neighbors) {
        if (!visited.contains(neighbor)) {
            dfs(neighbor, visited);
        }
    }
}
```

Extra Credit (5 points, all or nothing)

Consider a situation where your input int array of size n is almost sorted.

[1, 4, 3, 2, 5, 7, 6, 8]

In this array, each number is at most k positions away from its correctly sorted position. In the above example, $k = 2$. Complete procedure `sort` below according to its specification. You may use whatever data structure you would like and may use its methods without implementation as long as your intentions are clear

Your solution must run in $O(n \log k)$ time and be clearly organized for credit.

Solution

```
/** Sort the array.
  * Precondition: Each element is no more than k
  * positions away from where it should be in sorted order */
public void sort(int[] b, int k) {
    PriorityQueue<Integer> heap = new PriorityQueue();
    for (int i = 0; i < k; i++) {
        heap.add(b[i]);
    }
    for (int i = 0; i < b.length - k; i++) {
        b[i] = heap.poll();
        heap.add(b[i + k]);
    }
    for (int i = b.length - k; i < b.length; i++) {
        b[i] = heap.poll();
    }
}
```