Name: NetID:

# CS2110 Final Exam SOLUTION

**14 December 2019**

| | 1 | 2 | 3 | 4 | 5 | 6 | Total |
|---|---|---|---|---|---|---|---|
| Question | Name | Short Answer | Sorting and such | Data Structures | Graphs | Object Oriented | |
| Max | 1 | 24 | 20 | 16 | 19 | 20 | 100 |
| Score | | | | | | | |

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

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

Write your name and Cornell **NetID**, **legibly**, at the top of the first page, and your Cornell ID Number (7 digits) at the top of pages 2-8! There are 7 questions on 8 numbered pages, front and back. Check that you have all the pages. When you hand in your exam, make sure your pages are 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 it's good style.

**Academic Integrity Statement:** I pledge that I have neither given nor received any unauthorized aid on this exam. I will not talk about the exam with anyone in this course who has not yet taken the final.

_____

(signature)

## 1. Name (1 point)

Write your name and NetID, **legibly**, at the top of page 1. Write your Student ID Number (the 7 digits on your student ID) at the top of pages 2-8 (so each page has identification).

## 2. Short Answer (24 points)

**(a) True / False (13 points)** **Circle** T or F in the table below.

| | | | |
|---|---|---|---|
| (a) | T | F | Topological sort will work on every DAG (directed acyclic graph). True |
| (b) | T | F | Every $n$-node tree has the same number of edges. True: $n-1$ edges. |
| (c) | T | F | Let $A, B, C$ be nodes in an undirected graph. Suppose the shortest path from $A$ to $B$ is 10 and the shortest path from $A$ to $C$ is 5. A path exists in which the shortest path from $B$ to $C$ is 20. False. The $B$ to $C$ shortest path length is at most 15 because of the path $B, A, C$. |
| (d) | T | F | Since a `String` is stored as a `char` array, it's possible to modify a character in a `String` in constant time. False. Strings are immutable. |
| (e) | T | F | It's possible to have $f(x)$ in $O(g(x))$ but $f(x) > g(x)$ for all $x$. True. e.g. 2 in $O(1)$. |
| (f) | T | F | A `NullPointerException` is thrown when a variable is assigned the value `null`. False. The statement `Object x= null;` is legal and assigns null to `x`. |
| (g) | T | F | A class can extend only one class and implement only one interface. False. A class can implement several interfaces. |
| (h) | T | F | When implementing a set using hashing with open addressing and linear probing, to remove $b[k]$ from the set, store null in $b[k]$. False. Linear probing may not work properly after that, since it stops when a null value is found. |
| (i) | T | F | Upcasting happens automatically, but downcasting must be done manually. True |
| (j) | T | F | Since merge sort has runtime $O(n \log n)$ and insertion sort has runtime $O(n^2)$, merge sort is always faster. False. For small arrays, the overhead of recursion and merging can make merge sort slower. |
| (k) | T | F | When running a multi-threaded program with $n$ threads, you must have $n$ cores. False |
| (l) | T | F | If you run a multi-threaded program 1000 times and it works as you expected, you can be pretty sure that your code does not have a race condition. False |
| (m) | T | F | If you preface a Java method with the word `synchronized`, Java will not let multiple threads execute the method at the same time. True |

**(b) Concurrency (4 points)** Assume that the initial value of `y` is 2 and consider the code below in two threads. What are the possible values of `y` after execution of the two threads?

**Thread 1:**
```
y= y * 3;
```

**Thread 2:**
```
y= y + 5;
```

The possible values are (1) 11: Do y= y * 3; y= y + 5;.
(2) 21: Do y= y + 5; y= y * 3;.
(3) 7: Threads store y, which is 2, in different registers; y= 2*3 is done; then y= 2 + 5; is done.
(4) 6: Threads store y, which is 2, in different registers; y= 2 + 5 is done; then y= 2*3 is done.

**(c) Generics (4 points)** Consider the code segment given to the right. Below, to the right of each assignment statement, circle "Yes" or "No", depending on whether it is legal —it will compile:

1. `oarray[3]= "hello";` Yes No
2. `oarray= iarray;` Yes No
3. `s= ts;` Yes No
4. `ts= s;` Yes No

```
Object[] oarray= new Object[5];
Integer[] iarray= new Integer[5];

Set<String> s= new HashSet<String>();
TreeSet<String> ts= new TreeSet<String>();
```

1. Yes, 2. Yes, 3. Yes, 4. No.

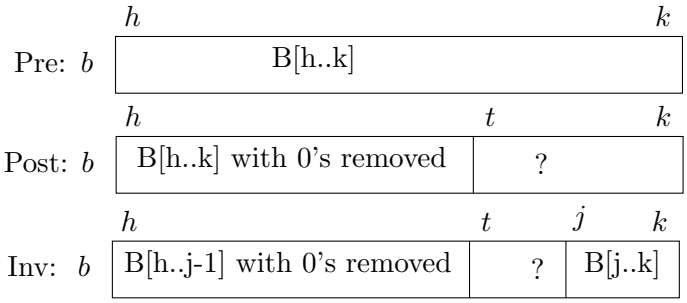
**(d) Big-O (3 points)** Prove that $6n^2 + 10n$ is in $O(n^2)$.

$6n^2 + 10n$
$\leq$ <Assume $1 \leq n$, so $n \leq n^2$ >
$6n^2 + 10n^2$
$=$ <Arithmetic>
$16n^2$ So choose $c = 16$ and $N = 1$

## 3. Sorting and such (20 points)

**(a) 8 points** We want a loop (with initialization) that moves all the non-zero values of `b[h..k]` to its beginning. Example: change (0, 5, 4, 0, 6, 0, 7) to (5, 4, 6, 7, ?, ?, ?), where ? indicates we do not care what is in that position.

To the right are the pre- and post-conditions and the loop invariant to be used in writing the code.




Here, we use `B` for the initial value of array `b`. For example, in the postcondition, the first segment `b[h..t-1]` contains the initial value of `b[h..k]` with zeros removed.

**(a1) 2 points** Write the loop initialization here: t= h; j= h;

**(a2) 2 points** Write the loop condition here (do not write "while"): $j \leq k$

**(a3) 4 points** Write the repetend. if (b[j] != 0) {b[t]= b[j]; t= t + 1;}
j= j + 1;

**(b) (4 points)** Complete the body of the loop in method `insertionSort()` below. State first in a comment what the body does. Then implement that comment, using a loop. Note that `Comparable` requires method `compareTo`, and that is what should be used to compare array elements. You may use a statement `swap(p, q)`.

```
public void insertionSort(Comparable[] b) {
    // inv: b[0..i-1] is sorted
    for (int i= 1; i < b.length; i= i+1) {
        // TODO 1:
        // Push b[i] down to its sorted position in b[0..i]
        int j= i;
        // inv: b[0..i] is sorted except that b[j-1] could be > b[j]
        while (j > 0 && b[j-1].compareTo(b[j]) > 0) {
            swap(j, j-1);
            j= j - 1;
        }
    }
}
```

**(c) (4 points)** Consider the following class `Catalog`. A catalog processing system needs to receive catalogs in order of decreasing inventory size. If two catalogs have the same inventory size, the catalog with the alphabetically earlier name should come first. Complete method `compareTo()` accordingly.

```
/** An instance represents a comparable Catalog object */
public class Catalog implements Comparable<Catalog> {
    public String name;
    public ArrayList<CatalogItem> inventory;
    ...
    /** = negative integer, zero, or positive integer depending on whether
      * this Catalog comes before, is equal to, or comes after ob. */
    @Override public int compareTo(Catalog ob) {
        int size= inventory.size();
        int ob_size= ob.inventory.size();
        if (size == ob_size) return name.compareTo(ob.name);
        return ob_size - size;
    }
}
```

**(d) (2 points)** Suppose the catalog processing system doesn't care what order to process catalogs if they have the same inventory size. Write an anonymous function that given catalogs `cat1` and `cat2` returns the appropriate `compareTo` value.

(cat1, cat2) -> cat2.inventory.size() - cat1.inventory.size()

**(e) (2 points)** State the tightest worst-case additional space complexity of the sorting algorithms below. For quicksort, assume that we are using the implementation using the least amount of space possible.

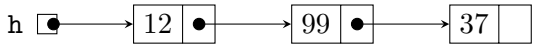
insertion sort: O(1) quicksort: O(log n) merge sort: O(n) selection sort: O(1)

# 4. Data Structures (16 points)

### (a) Linked lists (8 points)

Objects of class `ListNode`, defined partially to the right, form nodes of a singly linked list. An example of such a linked list is shown below the class definition, with `h` pointing to the head.

```
public class ListNode {
    private int val;
    public ListNode next;
    ...
}
```




Complete method `reverse`, given below. It must be recursive; it must not contain a loop. The `val` fields of the nodes must not be changed; only the `next` fields.

Be sure to consider the base case(s) first. In thinking about the recursive case, we suggest that you draw an example of the linked list before the recursive call and, under it, draw the list after the recursive call is executed, according to what the spec says it does. Do not be drawn into thinking how the recursion is executed.

```
/** Reverse the linked list whose head is (pointed to by) h and
  * return (a pointer to) the head of the reversed list. */
public ListNode reverse(ListNode h) {
    if (h == null || h.next == null) return h;  // A video in JavaHyperText shows
    ListNode result= reverse(h.next);           // how to develop this function.
    h.next.next= h;                             // Look in the videos on recursion.
    h.next= null;
    return result;
}
```

### (b) Stacks and Queues (4 points)

Sequentially push values A, B, C, D, E into a stack $S$, but note that an element may be popped from $S$ at any time. An element popped from $S$ is immediately pushed onto queue $Q$. One sequence below **cannot** be the content of $Q$ at the end. Circle the "NO" for that one. (C)

A. (E, D, C, B, A) NO

B. (D, E, C, B, A) NO

C. (D, C, E, A, B) NO

D. (A, B, C, D, E) NO

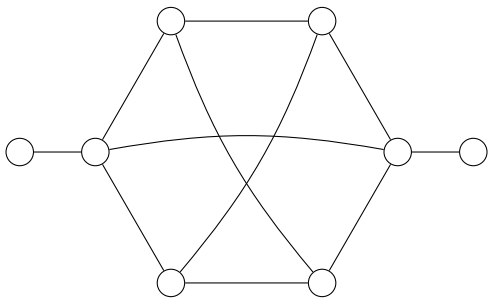
### (c) Trees (4 points)

Write the post-order of a binary tree with given pre-order and in-order.

- Pre-order: 27, 10, 5, 51, 41, 73, 99, 90
- In-order: 5, 10, 27, 41, 51, 73, 90, 99
- Post-order: 5, 10, 41, 90, 99, 73, 51, 27

## 5. Graphs (19 points)

**(a) Graph properties (8 points)** Consider the undirected graph to the right.



1. (2 points) Is it bipartite? Yes
2. (2 points) Is it planar? No
3. (2 points) At least how many colors do we need for a proper coloring? 2
4. (2 points) How many edges are in a spanning tree of this graph? 7

**(b) (2 points)** For a graph with all edge weights 1, Dijkstra's shortest path algorithm to find the distance from one node to all others is equivalent to one of algorithms given below. Circle it. BFS

BFS DFS Kruskal Prim

**(c) (2 points)** Consider a directed graph of all flights in the U.S. You want to determine whether there are flights from Ithaca to a particular city with one hop (e.g. Ithaca → Chicago → Phoenix). Which of the algorithms shown below is best suited for this purpose? Circle it. BFS

BFS DFS Kruskal Prim

**(d) (2 points)** Given a directed graph $G$, what is the tightest bound on worst-case runtime to check whether an edge $(x, y)$ exists in $G$:

1. If the graph is represented in an adjacency list? $O(outdegree\ of\ x))$. $O(|E|)$ is a possible answer, e.g. if the adjacency list is kept as a linked list, so the entry for x has to be searched for.
2. If the graph is represented in an adjacency matrix? $O(1)$

**(d) (5 points)** Below, write method `numreachable`. Make it recursive, not iterative. Keep the notion of "visited" abstract: write "visit n" to visit node n and "n is visited" or "n is unvisited" to check whether node n has been visited. You can use an English phrase to get all the neighbors of a given node.

```
/** Return the number of nodes reachable along a completely unvisited
  * path from s (including node s if it is unvisited). */
public int numReachable(Node s) {
    if (s is visited) return 0;
    visit s;
    int ret= 1;
    for each neighbor w of s { ret += numReachable(w); }
    return ret;
}
```

## 6. Object-Oriented Programming (20 points)

This question deals with characters in certain locations in a game. The necessary part of class `Location` appears to the right; it deals only with whether a location is occupied or not. Note the precondition on method `setOccupied`; at most one character can occupy a location at any time.

Also to the right is interface `Movable`, with one method.

In this question, be sure not to reimplement code that can be inherited.

Below is abstract class `Character`. Note that its method `equals` is specified, but we hide the body because you don't need to see it.

**(a) (4 points)** Complete the body of the constructor of class `Character`.

```
public class Location {
  /** True iff this location is occupied.*/
  private boolean occupied;

  /** Constructor: an unoccupied location*/
  public Location() {occupied= false;}

  public boolean isOccupied()
      {return occupied;}

  /** Pre: if this is occupied, s is false.*/
  public void setOccupied(boolean s)
      {occupied= s;}
}

public interface Movable {
  /** Change the location to loc.
    * Throw an Exception if already occupied.*/
  void moveTo(Location loc) throws Exception;
}
```

```
public abstract class Character {
    /** location is not null. */
    protected Location location;

    /** Constructor: An instance occupying loc.
    * Throw an Exception if loc is already occupied. */
    public Character(Location loc) {
        if (loc == null || loc.isOccupied()) throw new Exception();
        location= loc;
        location.setOccupied(true);
     }

    /** Return true iff this and ob are of the same class and have the same field f. */
    @Override public boolean equals(Object ob) { ... }
}
```

**(b) (10 points)** Class `Player`, given below, extends `Character` and implements `Movable`. Complete its constructor and methods `moveTo()` and `equals` —`equals` overrides `equals` in class `Character`.

```
public class Player extends Character implements Movable {

    private int health;

    /** Constructor for a player at location loc with initial health hStart.
      * Throw an Exception if illegal argument, such as loc is already occupied */
    public Player(Location loc, int hStart) {
        super(loc);
        health= hStart;
    }

    /** Move player to location loc.
      * Throw an Exception if loc is already occupied.*/
    public @Override void moveTo(Location loc) {
        if (loc == null || loc.isOccupied()) throw new Exception();
        location.setOccupied(false);
        loc.setOccupied(true);
        location= loc;
    }

    /** Return true iff this and ob are of the same class, have the same field f,
         and have the same health. */
    @Override public boolean equals(Object ob) {
        if (!super.equals(ob)) return false;
        return health == ((Player)ob).health;
     }
}
```

**(c) (6 points)** Below, write "Correct" after each line of code that is legal Java and "Error" if the line is not legal (won't compile).

```
1. Character c1= new Character(new Location()); Error, Character is abstract
2. Movable m1= new Movable();  Error, cannot create new Movable
3. Player p1= new Player(new Location(), 10); Correct
4. Character c2= new Player(new Location(), 10); Correct
5. Character c3= new Player(new Location()); Error, need two arguments
6. Movable m1= new Player(new Location(), 10); Correct
```