CS2110 Final Exam

11 May 2019

Question	1 Name	2 Short Answer	3 Collections	4 Sorting	5 Data Structures	6 Graphs	7 Object Oriented	Total
Max	1	23	13	12	16	15	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 2-12! There are 7 questions on 12 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 (0 points)

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 2-12 (so each page has identification).

2. Short Answer (23 points)

(a) True / False (10 points)   **Circle** T or F in the table below.

(a)	T	F	Using loop invariants guarantees the algorithm's runtime is at worst $O(n^2)$.
(b)	T	F	Race conditions are not a problem when the only operations are reads.
(c)	T	F	`LinkedList<Double>` is not a subtype of `LinkedList<Object>` even though `Double` extends `Object`.
(d)	T	F	Any graph with exactly 4 nodes is planar.
(e)	T	F	If $f(n)$ is in complexity class $O(n)$, it cannot also be in $O(\log n)$.
(f)	T	F	Objects of class `String` can be changed by method `substring`.
(g)	T	F	`this == null` always evaluates to false.
(h)	T	F	The nth number of the Fibonacci Sequence can be calculated in $O(\log n)$ time.
(i)	T	F	All methods in an abstract class must be abstract.
(j)	T	F	`ArrayList`'s method `contains` has a lower asymptotic complexity than `LinkedList`'s `contains`.

(b) Hashing (6 points)   Suppose we have a hash table of fixed size 10 and would like to insert the numbers 1, 3, 7, and 9. Give a hash function that would result in the maximum number of collisions. *You may not use hash function $x \to c$ where c is some constant.*

Consider the following hash table, using open addressing with linear probing. The hash table doubles in size when the load factor is ≥ 0.75. The hash function is the identity function $x \to x$.

	9	6	

What does the hash table look like after adding 2 and then 1?

(c) Concurrency (4 points)   Do you need to worry about concurrency when using a set like a HashSet in a program with many threads? If so, why, and if not, why not?

Do you need to worry about concurrency when using a List like an ArrayList in a program with many threads? If so, why, and if not, why not?

(d) Functions equals and hashCode (3 points) Suppose you are writing a class `C`, and you want it to have its own functions `equals` and `hashCode`. In order for these to be used in hashing, what property must the two functions have?

3. Collections (13 points)

(a) Pascal's Triangle (10 points)

Pascal's Triangle is shown to the right. We refer to each row of the triangle as a level. The row containing [1] is level 0, the row containing [1, 1] is level 1, the row containing [1, 2, 1] is level 2, etc. Each level is computed using values in the previous level. Values 1 are at the beginning and end of the row. For the values in-between, two adjacent elements from the previous row are added.

```
Level
0               1
1              1 1
2             1 2 1
3            1 3 3 1
4           1 4 6 4 1
5         1 5 10 10 5 1
6       1 6 15 20 15 6 1
```

Implement function `pascalLevel`, below. Recall that `L.add(e)` adds item `e` to the end of list `L`. *Use recursion.*

```
/** Return level n of Pascal's triangle.
  * Precondition: n >= 0. */
public ArrayList<Integer> pascalLevel(int n) {




}
```

(b) Using Collections (3 points)

Consider method `getSum` below.

```
public int getSum(_??_ ints) {
  int sum= 0;
  for (int v : ints) {
    sum= sum + v;
  }
  return sum;
}
```

For each of the options to the right, would `getSum` compile if _??_ were that type?

I. `int[]`

II. `float[]`

III. `LinkedList<int>`

IV. `ArrayList<Integer>`

V. `HashSet<Integer>`

VI. `C<Integer>` where class `C` implements interface `Iterator<Integer>`

4. Sorting (12 points)

(a) Heapsort (5 points)

Below are the pre- and post-condition of the second phase of algorithm heapsort. As you know, the second phase swaps the values of array `b` around so that `b` is sorted. We also give you the invariant of the loop of the second phase, and to the right are two methods you should use.

Write the second phase of heapsort, using the precondition, postcondition and invariant —if it helps, draw them as array diagrams on scratch paper.

```
/** Precondition: b[0..h] is a maxheap,
    except b[0] may be in wrong place.
    Bubble b[0] down in b[0..h] so that
    b[0..h] is a max-heap. */
public static void bubbleDown(b, h)

/** Swap b[i] and b[j]. */
public static void swap(int[] b, int i, int j))
```

Precondition: `b[0..]` is a max-heap
Postcondition: `b[0..]` is sorted

Invariant: `b[0..k]` is a max heap
`b[k+1..]` is sorted
`b[0..k] <= b[k+1..]`

```
// Perform the second phase of heapsort.
```

(b) Quicksort Partition Algorithm (7 points)

We want to implement the partition algorithm of quicksort differently. Below is a precondition, postcondition, and invariant for a loop. Assume $h < k$. Implement method partition below, using

(1) a loop developed from this precondition, postcondition, and invariant and

(2) the simplest code that is needed to complete the method; that is, the simplest code to get from Postcondition R below to the postcondition of partition. *Read the postcondition of method partition carefully. Write it as an array digram if that helps you.*

Use swap(b, i, j) to swap b[i] and b[j].

Precondition Q: b

h		k
x	?	

Postcondition R: b

h		j		k
x	<=x		>=x	

Invariant P: b

h		t		j		k
x	<=x		?		>=x	
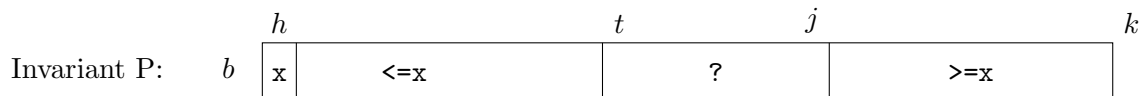
```
/** Let x be the value in b[h].
 * Swap values of b[h..k-1] and store a value in a local variable j to truthify:
 *    b[h..j-1] <= x = b[j] <= b[j+1..k-1]
 * Then return j. */
public int partition(int[] b, int h, int k) {




}
```

5. Data Structures (16 points)

(a) Stacks and Queues (4 points)

In the code to the right, we add and remove elements from a data structure **d**.

   I. What is printed if **d** is implemented as a stack?

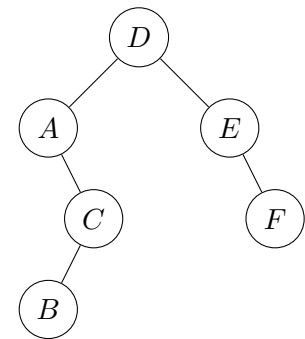   II. What is printed if **d** is implemented as a queue?

```
d.add(1);
d.add(2);
System.out.println(d.remove());
d.add(3);
d.add(4);
System.out.println(d.remove());
System.out.println(d.remove());
d.add(5);
System.out.println(d.remove());
System.out.println(d.remove());
```
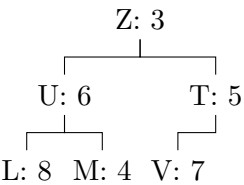
(b) Trees (4 points)
Write the preorder and level order traversals of this tree:
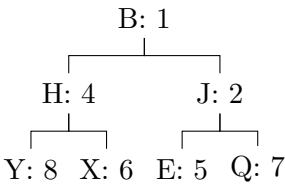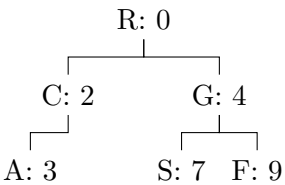
Preorder:

Level Order:

(c) Heaps (4 points)
State below each of the trees whether it is a valid min-heap. If it is not, state which invariant is unsatisfied. The letters are the values and the numbers are the priorities.
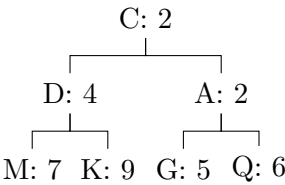

  1:           2:   3:   4:

(d) Linked Lists (4 points)
Your friend from CS 2110, Dave, comes to you with an idea for a better implementation of a singly linked list. SuperLinkedList is like a normal linked list, but it also has an array `b` with pointers to the nodes at each index (so `b[0]` is a pointer to the first node). Dave claims that, thanks to this improvement, the expected time of operation `get()` is now constant thanks to the array, while `prepend()` and `append()` remain constant time due to the linked list. Is Dave correct? Why or why not?

Another friend of yours, Mike, really hated assignment 3 and decides to get rid of nodes entirely. He implements his linked list using a HashMap, with the index in the "LinkedList" as the key and the value of that node as the value. He claims that for any set of operations this implementation runs at least as fast as an implementation of LinkedList using Nodes. Is Mike correct? Why or why not?

6. Graphs (15 points)

(a) DFS & BFS (3 points)
For each of the following scenarios, would it be better to use breadth-first search or depth-first search?

I. You have a graph with every city in the United States as nodes and connections between cities as edges. You want to find a route from Ithaca to the city of Cortland, which you know is somewhere nearby.

II. You are trying to navigate a character through a maze to find a glowing ring.

III. You want to route packets containing a video from one computer to another across a network using the fewest number of connections.

(b) Topological Sort (2 points)
Give one topologically sorted order of the nodes in the graph to the right.

N Q S
P R
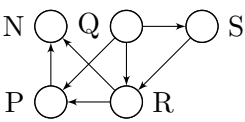
(c) Dijkstra's Shortest Path Algorithm (5 points)

Consider the graph to the right. Below, we have executed two iterations of Dijkstra's shortest path algorithm starting from node G. We show the settled set, the frontier set, the path lengths calculated so far, and the backpointers calculated so far.

Execute the next iteration of the algorithm, and write the results in the section "State after three iterations". Write the complete settled set, frontier set, all path lengths in d and backpointers in bk, as we did for the part after two iterations. Make it legible!
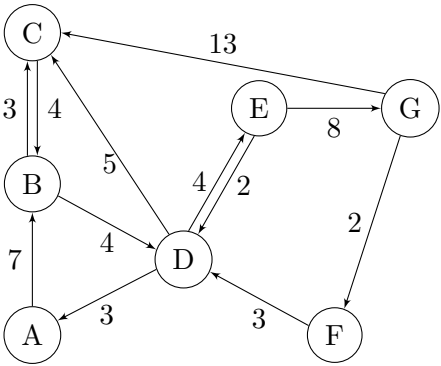
State after two iterations:

Settled	Frontier	Path length	Backpointer
F, G	C, D	d[G] = 0	bk[G] none
		d[C] = 13	bk[C] = G
		d[F] = 02	bk[F] = G
		d[D] = 05	bk[D] = F

State after three iterations:

Settled	Frontier	Path length	Backpointer

(d) In-/Out-Degree (5 points)

Recall that in a directed graph, the *in-degree* of a node n is the number of edges pointing from other nodes to n and the *out-degree* is the number of edges pointing from n to other nodes. For the following questions, assume that $|V|$ and $|E|$ are the numbers of nodes and edges of the graph.

i. What is the worst-case time complexity of calculating the *in-degree* of node n for a graph using an adjacency **matrix**?

ii. What is the worst-case time complexity of calculating the *out-degree* of node n for a graph using an adjacency **matrix**?

iii. What is the worst-case time complexity of calculating the *in-degree* of node n for a graph using an adjacency **list**?

iv. Implement function `outDegree` in class `Graph` below. It should be as fast as possible.

```
/** A directed graph */
public class Graph {
  /** Adjacency list: adj.get(n) returns a list of nodes reachable from Node n  */
  private HashMap<LinkedList<Node>> adj;

  /** Return the out-degree of Node n */
  public int outDegree(Node n) {




  }

  We omit unnecessary parts of this class.
}
```

7. Object-Oriented Programming (20 points)

This problem has five parts (a-e) across three pages. Unnecessary parts of classes are omitted. There is no need for assert statements for preconditions.

(a) (2 points) Implement SuperMarket's constructor.

(b) (6 points) Implement SuperMarket's method sell.

```java
/** An instance maintains info about a supermarket: items in stock and its sales. */
public class SuperMarket implements Iterable<Item> {
  /** The items currently in stock,
    * A key is an item, the value is the quantity of that item.
    * The quantity is always > 0. */
  private HashMap<Item, Integer> stock;
  /** The items sold; the value denotes the quantity sold */
  private HashMap<Item, Integer> sold;

  /** Constructor: an empty supermarket with no sales. */
  public SuperMarket() {




  }


  /** Return false if item is not in stock.
    * Otherwise, sell one item and return true.*/
  public boolean sell(Item item) {











  }
```

```
  /** Return an Iterator over the items in the supermarket */
  public Iterator<Item> iterator() {
    return new SuperMarketIterator();
  }
}


/** An item in a store, with a name and a price. */
public class Item {  We omit the implementation of this class.  }
```

(c) Iterator (5 points)
In order to make SuperMarket iterable, we need to write a private inner class, SuperMarketIterator, to implement the Iterator. Complete functions hasNext and next below.

```
/** Inner class that implements an iterator over the items in the SuperMarket. */
private class SuperMarketIterator implements Iterator<Item> {
  // A list of the items in this supermarket
  private List<Item> stockItems;
  // items stockItems[index..] still have to be enumerated.
  private int index;

  public SuperMarketIterator() {
    index= 0; // this can be omitted since 0 is the default for type int
    // Store the keys in HashMap stock into ArrayList stockItems.
    stockItems= new ArrayList<>(stock.keySet());  }

  /** Return true if there is another item to enumerate. */
  public boolean hasNext() {




  }

  /** Return the next item to enumerate, but
    * Throw a NoSuchElementException if there are no more. */
  public Item next() {






  }
}
```

(d) White Box Testing (4 points)

Imagine we added function `restock`, written below, to class `SuperMarket`. Below the method, write (in English) test cases to cover all possible branching paths through `restock`.

```
/** Add q items to the stock. Return true if, after restocking,
  * there are fewer than 10 of this item in stock after restocking. */
public boolean restock(Item item, int q) {
  int oldStock= 0;
  if (stock.containsKey(item) != false) {
    oldStock= stock.get(item);
  }
  int totalStock= oldStock + q;
  stock.put(item, totalStock);

  if (totalStock < 10) return true;
  return false;
}
```

Test Cases:

(e) new-expression (3 points)

Below, explain how the new-expression `new SuperMarket()` is evaluated.