Name: NetID:

CS2110 Final Exam SOLUTION

11 May 2019

	1	**2**	**3**	**4**	**5**	**6**	**7**	**Total**
Question	Name	Short Answer	Collections	Sorting	Data Structures	Graphs	Object Oriented	
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 $O(n^2)$. False. Loop invariants can be used for algorithms with any runtime complexity.
(b)	T	F	Race conditions are not a problem when the only operations are reads. True
(c)	T	F	`LinkedList<Double>` is not a subtype of `LinkedList<Object>` even though `Double` extends `Object`. True.
(d)	T	F	Any graph with exactly 4 nodes is planar. True. The only non-planar graphs include K_5 or $K_{3,3}$, both of which have more than 4 nodes.
(e)	T	F	If $f(n)$ is in complexity class $O(n)$, it cannot also be in $O(\log n)$. False. $f(n) = 1$ is in both.
(f)	T	F	Objects of class `String` can be changed by method `substring`. False. Strings are immutable; substring creates a new string.
(g)	T	F	`this == null` always evaluates to false. True.
(h)	T	F	The nth number of the Fibonacci Sequence can be calculated in $O(\log n)$ time. True. We saw two ways to do this in lecture.
(i)	T	F	All methods in an abstract class must be abstract. False.
(j)	T	F	`ArrayList`'s method `contains` has a lower asymptotic complexity than `LinkedList`'s `contains`. False. Both are expected and worst-case time $O(n)$.

(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 $x \to c$ where c is some constant.*
$x \to 10x$ would work, as does $10 * g(x)$ for any function $g(x)$. x^4 also works.

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?

	9	2	1			6	

(c) Concurrency (4 points) Do you need to worry about concurrency when using a set like a HashSet? If so, why, and if not, why not?
Yes. Example. Suppose two threads add the same element as roughly the same time. The same kind of race condition could happen as when two threads execute x= x+1; simultaneously.

Do you need to worry about concurrency when using a List like an ArrayList? If so, why, and if not, why not?
Yes, a race condition could happen. Suppose threads A and B want to remove the first element of the list if its size is > 0. A could see that the size is 1, then thread B removes the only element, then thread A gets an exception.

(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 function have?

If two objects `c1` and `c2` satisfy `c1.equals(c2)`, then this much be true:
`c1.hashCode() == c2.hashCode()`.

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`. *Hint: use recursion to get the previous level and work from there. Think about the base case!*

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

  ArrayList<Integer> p= new ArrayList<Integer>();
  p.add(1);
  if (n == 0) return p;

  ArrayList<Integer> last= pascalLevel(n - 1);
  for (int i= 0; i < last.size() - 1; i++) {
    p.add(last.get(i) + last.get(i + 1));
  }
  p.add(1);
  return p;
}
```

(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[]` yes

II. `float[]` no

III. `LinkedList<int>` no It would have to be `LinkedList<Integer>`

IV. `ArrayList<Integer>` yes

V. `HashSet<Integer>` yes

VI. `C<Integer>` where class `C` implements interface `Iterator<Integer>` no This would work only if it implemented `Iterable`, not `Iterator`

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.
k= b.length-1;
while (k > 0) {
  swap(b, 0, k);
  k= k-1;
  bubbleDown(b, k);
}
```

(b) Quicksort Partition Algorithm (7 points)

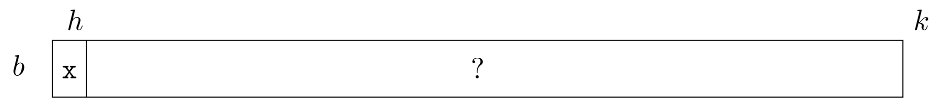
We want to implement the partition algorithm of quicksort differently. Below is a precondition, postcondition, and invariant for a loop. Assume $h < k$. Implement method `partition` below, using

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

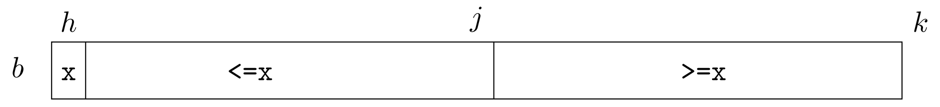
(2) the simplest code that is needed to complete the method; that is, the simplest code to get from Postcondition R below to the postcondition of `partition`. *Read the postcondition of method* ***`partition`*** *carefully. Write is an an array diagram if that helps you.*

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

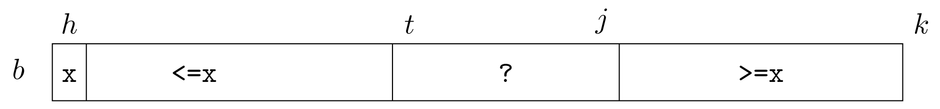
Precondition Q:



Postcondition R:



Invariant P:



```
/** 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 t= h+1;   int j= k-1;
  // invariant: P, given above
  while (t <= j) {
    if (b[t] <= b[h]) t= t + 1;
    else {
       swap(b, t, j);
       j= j - 1;
    }
  }
  // Postcondition R, above
  swap(b, h, j);
  return j;
}
```

Cornell ID Number (7 digits):

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?

2 4 3 5 1

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

1 2 3 4 5

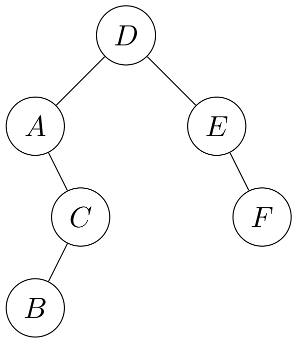
```
d.add(1);
d.add(2);
System.out.println(d.remove());
d.add(3);
d.add(4);
System.out.println(d.remove());
System.out.println(d.remove());
d.add(5);
System.out.println(d.remove());
System.out.println(d.remove());
```

(b) Trees (4 points)

Write the preorder and level order traversals of this tree:

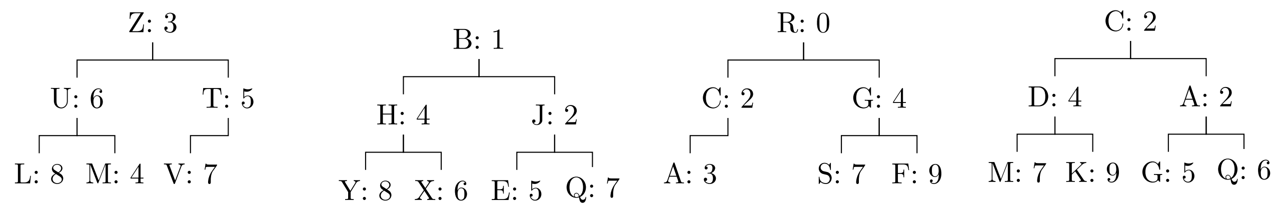
Preorder: D A C B E F

Level Order: D A E C F B



(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: Violates heap invariant

2: Valid

3: Violates completeness invariant

4: Valid. Equal priorities are allowed

(d) Linked Lists (4 points)

Your friend from CS 2110, Dave, comes to you win 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[2]` has a pointer to the second node). Dave claims that, thanks to this improvement, the expected time of the `get()` operation 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?

Dave is not correct. Because you have to keep track of the pointers to each node, `prepend()` takes time $O(n)$. This implementation has the same time guarantees as an ArrayList.

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?

Mike is not correct. A HashMap has **worst-case** time $O(n)$ to add a node, while a LinkedList has worst case constant time. Removing a node from the middle of the linked list would also take linear time when implemented using a HashMap.

6. Graphs (15 points)

(a) DFS & BFS (3 points)

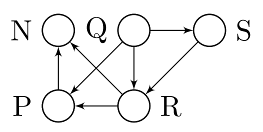
For each of the following scenarios, would it be better to use breadth-first search or depth-first search?

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. BFS DFS will travel along the first path it finds, which could go all across the U.S., while BFS will look at closer locations first and will therefore find Cortland quickly.

II. You are trying to navigate a character through a maze to find a glowing ring. DFS You used DFS for this exact scenario in A7. Part of the reason is that BFS requires moving between non-adjacent nodes, whereas DFS can be used to implement a DFS walk that only moves between adjacent nodes.

III. You want to route packets containing a video from one computer to another across a network using the fewest number of connections. BFS BFS will explore computers that are 1 connection away from the start, then 2 connections, etc, so it will find the path that requires the fewest hops.

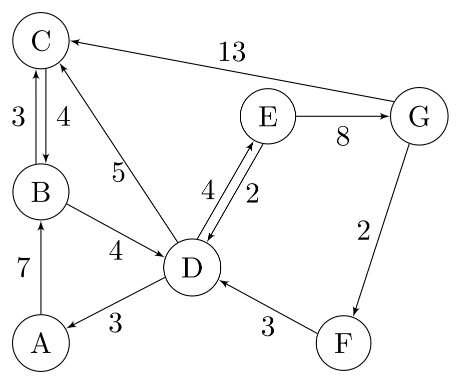
(b) Topological Sort (2 points)
Give one topologically sorted order of the nodes in the graph to the right.



Q S R P N

(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
F, G, D	C, E, A	d[G] = 0	bk[G] none
		d[C] = 10	bk[C] = D
		d[F] = 02	bk[F] = G
		d[D] = 05	bk[D] = F
		d[E] = 09	bk[E] = D
		d[A] = 08	bk[A] = D

(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**?
$O(|V|)$. The matrix column for n has to be traversed.

ii. What is the worst-case time complexity of calculating the *out-degree* of node n for a graph using an adjacency **matrix**?
$O(|V|)$. The matrix row for n has to be traversed.

iii. What is the worst-case time complexity of calculating the *in-degree* of node n for a graph using an adjacency **list**?
$O(|V| + |E|)$ Every node has to be checked for edges leaving it, and in the worst case node n is at the end of every one of the lists, so all $|E|$ edges have to be checked.

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) {
    return adj.get(n).size();
  }

  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`.

```
/** 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() {
    stock= new HashMap<>();
    sold= new HashMap<>();
  }

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

    // Remove item from stock
    int numLeft= stock.get(item) - 1;
    if (numLeft == 0)  stock.remove(item);
    else  stock.put(item, numLeft);

    // Add item to sold
    if (!sold.containsKey(item))  sold.put(item, 1);
    else {
      int temp= sold.get(item);
      sold.put(item, temp + 1);
    }

    return true;
  }
  /** 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;
  // Index of the next item to enumerate (stockItems.size() if none)
  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 index < stockItems.size();
  }

  /** Return the next item to enumerate, but
    * Throw a NoSuchElementException if there are no more. */
  public Item next() {
    if (!hasNext()) throw new NoSuchElementException();
    Item nextItem= stockItems.get(index);   OR    index++;
    index++;                                      return stockItems.get(index-1);
    return nextItem;
  }
}
```

(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:
We expect the following:

1. `item` is **not** in `stock` and q $<$ 10.
2. `item` is **not** in `stock` and q $\geq$ 10.
3. `item` is in `stock` and the previous stock plus q $<$ 10.
4. `item` is in `stock` and the previous stock plus q $\geq$ 10.

(e) new-expression (3 points)
Below, explain how the new-expression `new SuperMarket()` is evaluated.
1. Create (draw) an object of class `SuperMarket`.
2. Execute the constructor call `SuperMarket()`.
3. Give as value of the new-expression a pointer to the new object.