General instructions for this assignment.

You will have up to **2 hours** to complete this timed assignment (a 15-minute buffer has been provided).

Please note: Timed exams will auto-submit when the assignment deadline is reached or the exam timer has expired. Here are some tips to help you manage your exam time:

- When you start your exam, make a note on your scratch paper of the time you started and roughly calculate the end time.
- Have a clock nearby so that you can easily check the time.
- At regular intervals, check the Coursera timer to see how much time is remaining.
- Keep in mind that, while ProctorU may try to assist you in determining the amount of time left, you must refer to the official Coursera timer.
- Once the exam time ends, Coursera will automatically submit your work. Anything you have entered into the exam will be submitted at that time.

You may use **as many pages** of scratch paper as you need to work out your solutions. You are allowed to access the scratch paper throughout the duration of the exam. The scratch paper must be blank at the time you begin your exam and destroyed at the end.

You will need to use **one of the following web browsers**: Chrome, Safari, Internet Explorer, Firefox

You **may** use the following resources:

- An online calculator or computer calculator.
- Website: https://docs.oracle.com/javase/8/docs/api/ (including anything linked from that site or having a URL that starts with that prefix)

Notes/Books: This is an open-note/book exam.

Question 1 (10 minutes)

For each of the scenarios below, select, from the given list, the data structure that best fits the description or best solves the data storage challenge. Note that not all of the given data structures will be used, and some may be used more than once.

Data structures:

A. List
B. Queue
C. Stack
D. Set
E. Map
F. Tree
G. Trie
H. Heap
I. Graph

Scenarios:

I. The data structure upon which apples grow: (F) tree 🍎
II. Storing a reference table of birthdays of people in the class, to be accessed by name: (E) map
III. Collecting just the days of the year that have birthdays to celebrate: (D) set. We would not want to allow duplicates which would preclude using a list for collecting the days (if we are processing a class list, duplicates are very likely for 200+ students).
IV. Encoding a street map for navigation (computing routes to give directions for how to get from one place to another): (I) graph
V. Efficiently store the names of millions of people (one copy of each name if there are duplicates): (G) Trie. Efficient storage makes tries a better answer than an arbitrary set in this case
VI. Representation of the power grid of a country: (I) graph

Question 2 (5 minutes)

Among the following options, which best justifies the use of java.util.Vector instead of java.util.ArrayList?

A. It grows automatically as you insert elements: No, both grow automatically, though Vector has a different default growth factor and controls, that is not what was asked.
B. Thread safety: Yes, Vector is synchronized, ArrayList is not.
C. Sortable: No, they do not differ in terms of sortability.
D. Cooler name: Not really. Hopefully you are not ignoring functionality just for style points.
E. It doesn't matter, they are completely equivalent: No. They do actually have differences.

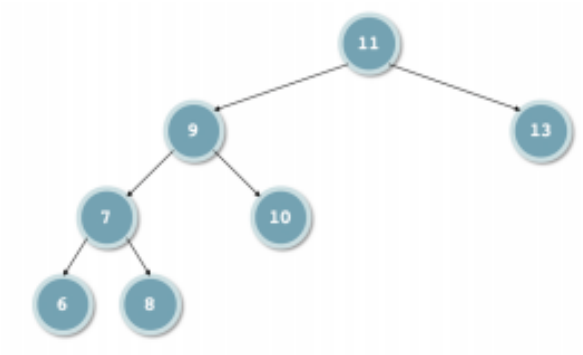
ArrayList and Vector in the java.util package are both implementations of the abstract concept of an “array list”. The organization and distinctions are a result of the early evolution of the Java language. If Vector had been added later as part of the collections framework (like ArrayList), it probably would be in java.util.concurrent and named ConcurrentArrayList.

Students, and really anyone who doesn’t use these classes frequently, would not be expected to remember the differences, but rather you are expected to refer to the documentation to be able to answer this question. It’s important to know when and how to look something up.

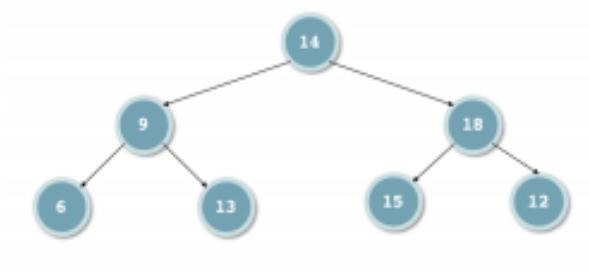
Question 3 (10 minutes)

Which of the following represents a valid AVL tree? Select all that apply.

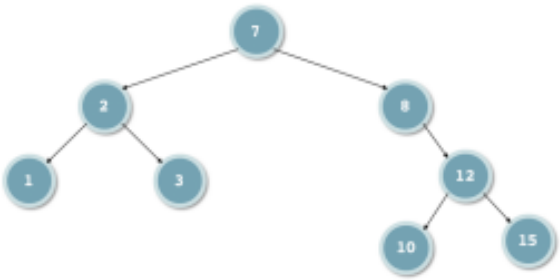
A.



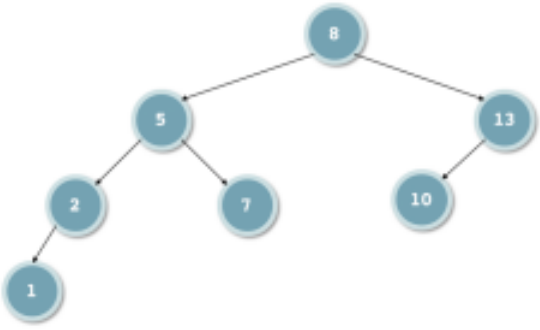
B.



C.



D.



Tree A is invalid because the imbalance at node 11. Node 12 is out of place in tree B. Tree C is imbalanced at node 8. So Tree D is the only valid AVL tree.

Question 4 (10 minutes)

Consider a HashSet using separate chaining with four buckets containing the following integer values:

- Bucket #0: 20 -> 4 -> null
- Bucket #1: null
- Bucket #2: 6 -> 16 -> null
- Bucket #3: 7 -> null

Given the values in the HashSet above, which of the following hash functions could be the one used in this implementation? Select the best answer.

A. hash(n) = n; If this were true, then element 16 would be in bucket #0.
B. hash(n) = n % 10; This works for all values in the HashSet.
C. hash(n) = n * n; If this were true, then element 6 would be in bucket #0.
D. hash(n) = n * 10; If this were true, then element 6 would be in bucket #0.
E. None of the above

When this question was presented to past groups of students, many were confused and selected E, since none of the four options directly produces the corresponding bucket number.

However, keep in mind that the hash function does not produce the bucket number when using separate chaining. The hash function produces the hashcode, and then the bucket number is calculated using the formula “hashcode % number_of_buckets” as we saw in the videos and as is common in the separate chaining implementation. Thus Option B is the best answer.

Question 5 (15 minutes)

```
1   public class MyClass {
2
3       private int g = 0;
4       private int k = 0;
5       private Object lock = new Object();
6
7       public void fun1() {
8           synchronized(lock) {
9               g += 3;
10          }
11          k++;
12      }
13
14      public void fun2(int a, int b) {
15          g += a;
16          a += b;
17          k = a;
18      }
19
20      public synchronized void fun3() {
21          g += k + 2;
22      }
23
24      public synchronized void fun4() {
25          k++;
26          g += 11;
27      }
28
29  }
```

For each of the 4 parts on the next page, assume that Threads T1 and T2 share the same MyClass object. Indicate whether each situation may or may not lead to a race condition, or whether it could never occur.

Question 5 (continued)

Part 1. T1 executes line 15 while T2 executes line 16.

A. This could occur, but would not lead to a race condition.
B. This could occur, and might lead to a race condition.
C. This could not occur.

This would not lead to a race condition because the threads modify two different variables. Additionally, the variable "a" is a local variable and each thread has its own copy. It should be noted that both threads executing line 15 simultaneously would lead to a race condition and therefore both threads executing this function as written is a hazard, it's just that this particular state is not a potential race condition.

Part 2. T1 executes line 26 while T2 executes line 21.

A. This could occur, but would not lead to a race condition.
B. This could occur, and might lead to a race condition.
C. This could not occur.

Because the two methods are synchronized, it is not possible for two threads to simultaneously run these methods on the same object, thus these lines could never execute at the same time.

Part 3. T1 executes line 21 while T2 executes line 9.

A. This could occur, but would not lead to a race condition.
B. This could occur, and might lead to a race condition.
C. This could not occur.

Both threads attempt to modify the same variable, and it is possible to run both simultaneously, since T1 is using "this" object's lock but T2 is using the "lock" object's lock.

Part 4. T1 executes line 11 while T2 executes line 25.

A. This could occur, but would not lead to a race condition.
B. This could occur, and might lead to a race condition.
C. This could not occur.

Both threads attempt to modify the same variable, and it is possible to run both simultaneously, since T2 is using "this" object's lock but T1 is not using any lock on line 11.

Question 6 (5 minutes)

There are bugs in this function. Fix them.

```
public static double mean(Collection<? extends Number> values) {
        double total = 0;

        if (values.isEmpty() || values == null)
                return Double.NaN;

        for (Number n : values) {
                total += n.doubleValue();
                if (n == null)
                        return Double.NaN;
        }

        return total / (double) values.size();
}
```

Enter the complete corrected function here:

```
public static double mean(Collection<? extends Number> values) {
        double total = 0;

        if (values == null || values.isEmpty())
                return Double.NaN;

        for (Number n : values) {
                if (n == null)
                        return Double.NaN;
                total += n.doubleValue();
        }

        return total / (double) values.size();
}
```

Both null checks in the original code occur after the variables are dereferenced and would be unreachable in the cases where they would be useful. This is almost always wrong (one might deliberately write bad code to test tools intended to detect and warn about mistakes like these) and clearly bugs. You are given no further information, so it is not safe to assume that returning NaN is a mistake or that you don't need to worry about nulls. Thus, removing the null checks because they are useless as written is not a valid solution. In past usage of this question, many students caught one of the two "use before null check" bugs but failed to fix the other.

Note: in C-like languages the binary boolean operators like “||” and “&&” are not strictly transitive — order does matter in terms of what gets evaluated. For “||” (“or”), if the expression to the left is true, the expression on the right is not evaluated.

So:

```
X = A || B;
```

is equivalent to:

```
if(A) X = true;
else
  if(B) X = true;
  else X = false;
```

Question 7 (25 minutes)

Consider the following function:

```
public static int count(int[] A, int[] C) {
        int M = A.length;
        int N = C.length;
        int count = 0;
        for (int i = 0; i < M; i++) {
                for (int j = 0; j < N; j++) {
                        if (A[i] > C[j])
                                count++;
                }
        }
        return count;
}
```

A. What is the asymptotic complexity as written? (4 minutes)
   a. O(1)
   b. O(N)
   c. O(M+N)
   d. O(M*N); this is a simple question of nested loops — the runtime is the product of the number of iterations of the two loops.
   e. O(M^N)
   f. May not terminate

Write a faster version (20 minutes):

```
public static int count(int[] A, int[] C) {
      int M = A.length;
      int N = C.length;

      PriorityQueue<Integer> aPQ = new PriorityQueue<>();
      for(int a: A) aPQ.add(a); // M log(M)

      PriorityQueue<Integer> cPQ = new PriorityQueue<>();
      for(int c: C) cPQ.add(c); // N log(N)

      int count = 0;
      int traversedC = 0;
      while(!aPQ.isEmpty()) { // M iterations
                  int a = aPQ.poll();
                  while((!cPQ.isEmpty()) && a > cPQ.peek())
                  {
                        cPQ.poll(); // log(N)
                        traversedC++;
                  }
                  count += traversedC;
      }
      return count;
}
```

We are not given details about the size of M or N, if they are sorted, and whether they are dense or sparse. Therefore we must consider how runtime will scale for arbitrary inputs.

First, consider the correctness of this approach. Does this function produce the same output as the original implementation? The original function computes the sum, over all values A[i] in A, of the number of values in C that are less than A[i]:

$$\sum_{i=0}^{M} \sum_{j=0}^{N} (A[i] > C[j])? 1 : 0$$

In the improved version, the priority queues provide a monotonically increasing stream of the values from the original lists. (Alternately, you can think about treating them as sorted arrays, which would only change a few lines of code). Because A[i] is strictly increasing as we progress through those values, we know that the number of elements of C that are less than A[i] will include all the elements that were less than A[i - 1], and also may include a few more that are greater than or equal to A[i-1] but less than A[i]. Instead of restarting the count from the

beginning of C, we can just look for new items that need to be added to the running tally (traversedC). With cPQ being a sorted form of C, we can just poll repeatedly as long as the element at the head of the queue is smaller than the current A[i] (in the code "a > cPQ.peek()"). With each iteration of the for loop, traverseC grows to the number that we need to add to count, yielding the same overall count in the end. The improvement in runtime comes from only going through the elements in C twice, once to enqueue and once to compare. (The extra peek and compares contribute a lower order term that drops out of the big-O runtime.) But to get to this point, we had to sort the lists, so what does that do to the runtime?

Filling a priority queue (java uses a heap for its PriorityQueue) with a list takes N*log(N) time. So the first half of the function is O(M * log(M) + N * log(N)) to create the two queues.

In the second half of the function, the for loop over aPQ is M iterations and we should assume that getting each value from aPQ is log(M). At first glance it might look like the inner while loop is O(N * log(N)), and in the worst case (the first value from A is larger than all the values in C) it actually is. But it should be noted that the queue is drained — the elements are only removed from cPQ once per invocation of the body of that while loop. Even though that loop is inside the for loop, the body of the while loop will not run more than N times in total. The while predicate (isEmpty and peek) is composed of constant time operations and does not add to the asymptotic complexity. Therefore total runtime of the while loop across all iterations of the for loop will be O(N * log(N)). Adding that to the operations to aPQ and we get O(M * log(M) + N * log(N)).

Given the complexities of the two halves of the function are the same, that means the overall runtime is O(M * log(M) + N * log(N)).

B. What is the asymptotic complexity of an optimal implementation? (1 minute)
    a. O(N)
    b. O(M + N)
    c. O(M * N)
    d. O(M * log(M) + N * log(N)) ; see the above text for a detailed explanation
    e. O(M * log(M) * N * log(N)) ; Note: product, not sum for this answer

Question 8 (20 minutes)

Write a function to reverse a linked list. The function should take a list node that is the head of the list and return a node that is the new head of the reversed list. This function should be destructive (it will necessarily alter the nodes in the list instead of copying the data stored therein). **You may not make any function or method calls at all from your function.** Anything slower than O(N) will be penalized.

```
public class LinkedList {
	public static class Node {
		String value;
		Node next = null;
	}

	public static Node reverse(Node head) {
		/* Insert code here */
		Node oldHead = head;
		Node temp;
		head = null;
		while(oldHead != null) {
			temp = oldHead.next;
			oldHead.next = head;
			head = oldHead;
			oldHead = temp;
		}
		return head;
	}
}
```

Question 9 (20 minutes)

In the programming assignments you were asked to implement a function to check if a graph walk was a Hamiltonian Cycle for a given graph:

isHamiltonianCycle: Given a Graph and a List<String> of node values, this method indicates whether the List represents a Hamiltonian Cycle through the Graph.

A Hamiltonian Cycle is a valid path through a graph in which every node in the graph is visited exactly once, except for the start node, which must be visited twice and must be identical to the end node, so that the path forms a complete cycle.

If the values in the input List represent a Hamiltonian Cycle given the order in which they appear in the List, the method should return true, but the method should return false otherwise, e.g., if the path is not a cycle, if some nodes are not visited, if some nodes are visited more than once, if some values do not have corresponding nodes in the graph, if the input is not a valid path (i.e., there is a sequence of nodes in the List that are not connected by an edge), etc.

The method should also return false if the input Graph or List is null.

A. Do each of the following tests give you useful information (even if it's only slightly useful or might not be a test you would actually choose to use) towards determining whether the walk is a Hamiltonian cycle or not? Answer true or false for each test. Consider each test independently from the others.

   a. Are there any duplicates in the list other than the first and last nodes? True: Useful to reject
   b. Is there at least one node in the graph with at least two incoming edges? False: even the starting node just needs one incoming edge for the return step
   c. Are the first and last nodes in the list the same? True: part of the definition of cycle
   d. Is the graph a directed graph? False: Irrelevant
   e. Is the input list a null pointer? True: mentioned explicitly in the problem description
   f. Are all of the nodes in the graph in the list? True: part of the definition of a Hamiltonian Cycle.
   g. Is the graph a tree? True: a tricky one. If you know it's a tree, you can reject it because the graph can not have any cycles. But this is not really a practical test that you would actually use, as it only addresses certain special cases that do not reduce the work you need to do when the graph is not a tree.
   h. Is the input graph actually a null pointer? True: mentioned explicitly in the problem description
   i. Is there a repeated edge in the list? True: you can reject a walk with repeated edges.

j. Is the list the shortest walk (with the same start and end node) that traverses all nodes in the list? False: a graph may have multiple different Hamiltonian cycles with different distances.
k. Does the list contain null values? False: Null and empty strings are valid keys in a HashMap. There is no information provided to suggest those would be invalid node labels.
l. Is the length of the list the same as the number of nodes in the graph? True: But this is only slightly useful. The length of the list should be V + 1 (the number of nodes + 1, because of the duplication of the first and last items). Checking if it is V is just a check for one of many (potentially infinite) wrong values.
m. Are all edges in the list valid edges in the graph? True: this is part of the definition of being a path or walk through the graph. All edges in the list must be edges in the graph.
n. Are all edges in the graph in the list? False: There are cases where this is true and it's a Hamiltonian Cycle (the graph is just a ring), cases where it's false (add one more edge to that ring), and likewise cases where the list is not a valid Hamiltonian Cycle where this property is true (acyclic graphs), and where it is false (again, a ring with an extra edges).
o. Is the graph fully connected? True: But only slightly useful. A fully connected graph of at least 3 nodes will definitely have Hamiltonian Cycles. Even though this test doesn't tell us whether or not the given list is a Hamiltonian Cycle, it does confirm that it's not impossible, just as the tree test confirms that it *is* impossible.
p. Are all items in the list nodes in the graph? True: this is a critical property

B. What is the optimal runtime for this method (V = the number of nodes/vertices, E = the number of edges)? You may assume the graph uses HashMap<String, ? extends Collection<Edge>> to store nodes and edges ,and that all HashMap operations are constant time.
    a. O(-V) ; You've created a working time machine and should apply for a Nobel a year ago
    b. O(1) ; return false;
    c. O(V) ; See the next explanation. Basically there's not enough information to expect we can avoid stepping through all edges.
    d. O(V + E); We have to check each node, and from each node see if there is an edge to the next item in the list. We have no information about optimized storage that would allow for faster adjacency checks so must iterate through each edge from the current node to see if the next item in the list is reachable (i.e., if the pair {current, next} represents a valid edge). In the starter code given in the original assignment, the provided methods do step through each edge, to build a neighbor set. It was possible to reduce this to O(V) in that case by constructing a new edge with the pair and probing the edge set directly, but this optimization is specific to that given implementation and is less likely to work in more interesting implementations of graphs.
    e. O(V * E) ; It's only necessary to step through the edges coming from the node being examined at each step, not all edges
    f. Exponential ; Only if something is really wrong with an implementation
    g. Not guaranteed to terminate ; It's easy to forget to increment a loop iterator!