The test has six problems. You’ll be graded on your best four answers.

1. Suppose that we wanted to modify the Java class `NodeOutDeg1` so that objects of that class represented graph nodes having at most (rather than exactly) one outgoing edge. Naturally, the `next` instance variable in a node having no outgoing edge would have value `null`. Of course, in any graph all of whose nodes have outdegree at most one, each node is such that the path from it either eventually reaches a cycle or else a “dead end”.

Below is a short excerpt of the class showing instance variable declarations and one constructor. (If you need to see the whole class, use the link on the course web page or ask the instructor to display it on the screen.)

```java
public class NodeOutDeg1 {

    // instance variables
    // ------------------
    private final int label; // For debugging purposes
    private NodeOutDeg1 next; // Node to which outgoing edge is directed
    private boolean marked; // For bookkeeping during path traversals

    // constructors
    // -----------

    /* Establishes this node as having the given label and an
    ** outgoing edge to itself.
    */
    public NodeOutDeg1(int label) {
        this.label = label; marked = false; this.next = this;
    }
}
```

(a) Indicate how you would modify the constructor shown above so that the new node had no outgoing edge.

(b) Develop a method called `reachesCycle()` that, when applied to a node, reports whether or not the path from that node reaches a cycle. A call to such a method from a client could look like this: `v.reachesCycle()`.

2. Within the same scenario as the previous problem, develop a method called `makeSelfLoops()` that, when applied to a node, changes the outgoing edge of every node reachable from it so that every such node has a self-loop (meaning an edge to itself).
3. The following is an excerpt of the class `QueueViaArray<T>`. Assume that it employs the wraparound/circular array-based representation of queues, as discussed in class.

```java
class QueueViaArray<T> implements Queue<T> {
    /* instance variables used in representing a queue */
    private T[] contents; // array to hold items occupying the queue
    private int frontLoc; // points to array location holding front item
    private int numItems; // holds # items currently in the queue

    /* instance methods */
    public QueueViaArray() { ... }
    public void enqueue(T item)
    { contents[(frontLoc+numItems) % contents.length] = item; numItems++; }
    public void dequeue()
    { frontLoc = (frontLoc + 1) % contents.length; numItems--; }
    ...
    }
```

Suppose that an instance of `QueueViaArray<Character>` is such that its instance variables have values as depicted in the picture below. (For brevity, elements of `contents[]` are shown as characters rather than as references to instances of class `Character`.)

```
0 1 2 3 4 5 6 7 8 9   frontLoc numItems
+-----------------+-------------------+-----+-----
contents |k'|$'|r'|o'|t'|s'|g'|j'|u'|n'|   | 7 |  4 |
+-----------------+-------------------+-----+-----
```

(a) Logically speaking, what is the sequence of characters that resides on the queue having this concrete representation? Print them in order from front to rear.

(b) Suppose that we call `enqueue()` to place the characters `e` and `t` into the queue, in that order. Show the updated concrete representation of the queue (i.e., the updated values of `contents`, `frontLoc`, and `numItems`).

```
0 1 2 3 4 5 6 7 8 9   frontLoc numItems
+-----------------+-------------------+-----+-----
contents | | | | | | | | | | |   | 7 |  4 |
+-----------------+-------------------+-----+-----
```

(c) Suppose that the `dequeue()` operation were performed twice (on the original queue shown above, not upon the result of (b)). Show the updated concrete representation of the queue.
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>frontLoc</th>
<th>numItems</th>
</tr>
</thead>
<tbody>
<tr>
<td>+--+--+--+--+--+--+--+--+--+--+--</td>
<td>----</td>
<td>----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>contents</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+--+--+--+--+--+--+--+--+--+--+--</td>
<td>----</td>
<td>----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

3
4. Suppose that graph refers to an instance of the class Graph having ten vertices and having edges as indicated in the adjacency matrix shown below on the left. Specifically, the matrix should be interpreted to mean that graph.hasEdge(i,j) is true (i.e., there is an edge from vertex i to vertex j) if and only if the entry in row i and column j of the matrix is 1. (Rows are horizontal, columns are vertical.)

Further suppose that we make the call distancesFrom(graph, 3) to the method shown below on the right, which will return an array containing the lengths of the shortest paths from vertex 3 to all vertices reachable from it. Show the contents of this array, as well as the history of the queue used in the method.

```java
public int[] distancesFrom(Graph g, int source) {
    final int N = g.numVertices();
    int[] dist = new int[N];
    dist[source] = 0;
    Queue<Integer> q = new QueueViaX<Integer>();
    q.enqueue(source);
    while (!q.isEmpty()) {
        int x = q.frontOf(); q.dequeue();
        for (int y = 0; y != N; y++) {
            if (dist[y] == -1 && g.hasEdge(x,y)) {
                dist[y] = dist[x] + 1;
                q.enqueue(y);
            }
        }
    }
    return dist;
}
```

History of the queue:

```
+----------------+----------------+----------------+----------------+----------------+
|                |                |                |                |                |
|                |                |                |                |                |
+----------------+----------------+----------------+----------------+----------------+
When Enqueued

When Dequeued

Final contents of array returned:

```
0 1 2 3 4 5 6 7 8 9
+----------------+----------------+----------------+----------------+----------------+
|                |                |                |                |                |
|                |                |                |                |                |
+----------------+----------------+----------------+----------------+----------------+
```

**Bonus:** Show how to modify the Java method so that, rather than returning an array indicating the distances from the source vertex, it returns an array indicating, for each vertex, the vertex that precedes it on the path to it from the source vertex.
5. Suppose that HeapSort was being applied to an array of type int[] and Phase 1 (Heapification) ended with the array containing (34, 21, 17, 19, 6, 8, 13). Fill in the empty nodes (i.e., array elements) below to show the contents of the heap as it exists after each of the six iterations of Phase 2 (Deconstructing the Heap). The contents of the heap before the first iteration is provided.
6. When a heap is being used as the underlying representation for a priority queue, it needs, in addition to the `siftDown` operation used by the `HeapSort` algorithm, a `siftUp` operation. Specifically, when a new value is to be placed into the heap (because a value is being inserted into the priority queue that the heap represents), we “create” a new leaf node, place the value into it, and then perform a `siftUp` operation. This operation follows an “upward” path in the heap (in contrast to `siftDown`, which goes “downward”) starting at the new node, swapping the values in the current node and its parent until either the current node is the root or its value is in proper relation to that in its parent.

Supply the body of the method `siftUp()`. You should assume that the heap in question is a max-heap containing values of type `int` and that its underlying representation is an array, just as in the HeapSort programming assignment. In such an array, the parent of node $k$ is node $(k - 1)/2$. (More precisely, if somewhat pedantically, location $(k - 1)/2$ contains the value of the node that is the parent of the node whose value is contained in location $k$.)

```java
/* Performs a siftUp operation starting at node k within the ** max-heap represented by A[]. */
public static void siftUp(int[] A, int k) {
```