Unless you qualify for extended time on exams, you have 2 hours to complete this exam once you start it. When you have finished the exam please write and sign the Honor Pledge on this page. Also give your starting and stopping times for the exam.

If you forget the name of something (oh, what does Java call the length of a String??) just write a note saying “I’m going to call this X” and then use that.

The 8 numbered questions are equally weighted.
1. Here is a list of data: 4 2 12 1 3 9 0 6. For each of the following structures I will walk through the list in order, add each item to the structure and then go into a loop in which I remove elements one at a time from the structure and print them as I remove them. **In what order do I print the items for**

A) **A stack.** Adding to the structure: push; removing from the structure: pop  
   reverse the order: 6 0 9 3 1 12 2 4

B) **A queue.** Adding to the structure: offer; removing from the structure: poll  
   keep in original order: 4 2 12 1 3 9 0 6

C) **A priority queue.** Adding to the structure: offer; removing from the structure: poll  
   increasing order: 0 1 2 3 4 6 9 12

D) In the add stage, I insert the values into a **BinarySearchTree** that starts off empty. So 4 becomes the root and we insert the other values around it.  
   Skip the remove stage and just give a level-by-level listing of the final tree, as we did on Exam 2. If you don’t remember Exam 2, list the root on the first line, the root’s two children on the third line, and so forth.

![BinarySearchTree Diagram](attachment:image.png)

E) **In the add stage I form a hash table of size 8** (the data fits; you don’t need to resize the table) with linear open addressing, using each data value as its own hash code (so the hash value is the remainder when we divide the value by 8). In the remove stage I remove and print the data at index 0, then the data at index 1, then index 2, etc.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>12</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>
2. In each part give a Big-Oh estimate of the worst-case time it takes to find an element in the given structure

   a) An AVL tree with n nodes

      \[ O( \log(n) ) \]

   b) A Binary Search tree with n nodes

      \[ O(n) \]

   c) A sorted ArrayList with n nodes

      \[ O( \log(n) ) \]

   d) A sorted LinkedList with n nodes

      \[ O(n) \]

   e) A directed graph with n nodes and m edges, using the graph structure from Lab 9, where the data is stored in the vertices so you need to search through the vertices.

       A lot of people missed this one. We need to visit every node in a graph. In Lab 9 our graph was built with a hashmap of vertices. This hashmap will have roughly \[ O(n) \] entries. The average lookup in a hashmap is constant time, but the worst case is \[ O(n) \]. So we need to do \[ n \] lookups, each of which takes time \[ O(n) \]. The total time this takes is \[ O(n^2) \].
3. Here is a wild recursive function.

```c
int collatz( int n ) {
    if (n == 1)
        return 1;
    else if (n%2 == 0)
        return 1+collatz(n/2);
    else
        return 1+collatz(3*n+1);
}
```

**Explain in English how you would create a Dynamic Programming version of this function.** Note that the arguments to collatz can become very large – larger than any table size. You can handle that any way you like (e.g., expanding the table or only using Dynamic Programming for some values of n), but you should handle it some way. What would you initialize your table entries to?

One way to do this is to make our table be an array of ints of size N. N could be anything you want, but just to have something to talk about let’s say N is 1000. Initialize all of the entries of the array to -1 (it is easy to see that the Collatz function can’t return a negative value), except for the index 1 entry which we initialize to 1. The body of collatz(n) then becomes

```c
if ( n < N && Table[n] > 0)
    return Table[n];
else if (n%2 == 0) {
    int t = 1+collatz(n/2);
    if (n < N)
        Table[n] = t;
    return t;
}
else {
    int t = 1+collatz(3*n+1);
    if (n < N)
        Table[n] = t;
    return t;
}
```

An alternative is to make the table be a hashmap, which maps n to the value of collatz(n). If you do this you sacrifice a little bit in execution speed, but you can forget all about the n<N comparisons. If Table.get(n) == null then compute the value of collatz(n) as t, do Table.put(n, t), and return t.
4. Here is an AVL tree:

![AVL Tree Image]

In case you have forgotten what level-by-level listings look like, here is one for this tree:

```
100
50  200
30  70  150  300
20  130  250  400
220
```

Give either the AVL tree or a level-by-level listing of the AVL tree that results from inserting value 230 into this tree.

The process is

a) Use BST properties to insert the node.
b) Let Z be the lowest node above the insertion point that fails the AVL property
c) Let Y be Z’s tallest child and let X be Y’s tallest child.
d) Let a, b, and c be X, Y, and Z in increasing order.
e) Let t1, t2, t3, and t4 be the four subtrees of X, Y, and Z that are not X, Y, or Z
f) Replace node Z with b; b’s two children are a and c; the four children of a and b are t1, t2, t3, and t4.

Some of you choose the wrong node for Z. If you choose a higher node (such as 300 instead of 250) the algorithm isn’t guaranteed to produce an AVL tree. If you stick with the algorithm it always reorganizes into an AVL tree.

In the following picture the tree on the left is the original tree doctored to do steps (a) through (e). Note that the subtrees t1, t2, t23, and t4 are all null. The tree on the right is the answer created from step (f).
5. Here is a picture of a binary Heap represented as a tree:

![Binary Heap Diagram]

If you prefer this could be represented as an array:

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>30</th>
<th>10</th>
<th>40</th>
<th>35</th>
<th>15</th>
<th>20</th>
<th>50</th>
<th>60</th>
</tr>
</thead>
</table>

**Give an algorithm (in English, not in code) for deleting a node from this heap.** For example, we might want to delete the value 30. You can assume that we are starting at the node that needs to be deleted; you don’t have to find it. Note: we didn’t talk about this in class; I’m asking you to invent an algorithm.

Heaps have to be complete binary trees; they can’t have holes. For example we couldn’t just remove value 15 from the tree above. So the algorithm is:

a) Replace the value to be deleted by the value of the last leaf (60 in this example). If we are storing the values in indexes 1 through n of an array, the last leaf is the value at index n.

b) Remove the last leaf and reduce the size of the heap by 1.

c) If the value of the last leaf is less than the value being deleted, percolate up starting at the point of deletion. If the value of the last leaf is greater than the value being deleted, percolate down.
Here is most of a class that implements queues with integer values:

```java
class MyQueue {
    class Node {
        int data;
        Node next;
        Node(int d) { // NODE CONSTRUCTOR
            data = d;
            next = null;
        }
    }

    Node head, tail;

    public MyQueue() { // QUEUE CONSTRUCTOR
        head = null;
        tail = null;
    }

    public void offer(int x) { // INSERT INTO QUEUE
        if (head == null) {
            tail = new Node(x);
            head = tail;
        } else {
            tail.next = new Node(x);
            tail = tail.next;
        }
    }
}
```

Give code for the following method which removes and returns the head of the queue:

```java
int poll() throws NoSuchElementException
```

```java
int poll() throws NoSuchElementException {
    if (head == null)
        throw new NoSuchElementException("The queue is empty");
    else {
        Node p = head;
        head = head.next;
        if (head == null) tail = null; // THIS COULD BE OMITTED
        return p.data;
    }
}
```
7. We can represent binary trees with the following Node class:

```java
class Node {
    String name;
    Node left, right;
}
```

In this representation an empty tree is null; a leaf is a Node where left and right are both null.

a) Write a function `int height(Node t)` that returns the length of the longest path from node `t` to a leaf. The height of an empty tree is -1; the height of a leaf is 0.

```java
int height(Node t) {
    if (t == null)
        return -1;
    else if (t.left == null && t.right == null)
        return 0;
    else return 1 + Math.max(height(t.left), height(t.right));
}
```

b) The diameter of a tree is the length of the longest path from one leaf to another. This longest path may or may not go through the root. In the following pictures the tree on the left has diameter 3, and the tree on the right has diameter 6; the dark edges show longest paths.

![Diagram of binary trees](image.png)

The diameter of a leaf is 0, the diameter of an empty tree is -1.

Write `int diameter(Node t)` that returns the diameter of the tree whose root is `t`.

```java
int diameter(Node t) {
    if (t == null)
        return -1;
    else if (t.left == null && t.right == null)
        return 0;
    else return Math.max(height(t.left) + height(t.right) + 2,
                          Math.max(diameter(t.left), diameter(t.right)));
}
```

The picture above on the left shows a longest path going through node `t` (the root). Its length is 2 (from the edges from `t` to its children) plus the sum of the child heights. The picture on the right shows the diameter being inherited from `t`'s right child.
8. Finding shortest paths in a rectangular grid is easy -- the length of the shortest path from one node to another is the difference in their columns plus the difference in their rows. But this goes out the window if the grid is missing some of its nodes. Consider the following grid where the black circles are existing nodes and the white circles are missing nodes:

Node A is at [3,2] (row 3, column 2). Node B is at [1,2]. The shortest path from A to B has 6 steps: From A to [3,1] to [3,0] to [2,0] to [1,0] to [1,1] to B. If the node at [2,4] was live there would be a path of length 4 from A to B.

Suppose we have a grid with R rows and C columns, and a boolean function Live(r, c) that returns true if there is a live node at row r, column c and false if there isn’t. You can also assume that Live(r, c) returns false if r<0 or r >=R or c<0 or c>=C. Finally, you can assume that, like the Square class in Lab 3, each node knows its row and column location.

I am going to designate one existing node on the grid as BOB. **Give an algorithm that finds the length of the shortest path, if there is one, from BOB to every other existing node.** Note that I am only asking for the lengths, not the actual paths. If it helps you can modify the nodes to store any data you want.

Give every node a distance from BOB; initialize this to -1. Make a queue of nodes. Start by putting BOB in the queue with distance 0.

Repeat the following until the queue is empty:

a) Poll the queue; let’s call the node we get X.
b) If X is at row r, column c, think of the four locations (r-1, c), (r, c+1), (r+1, c), (r, c-1).
   If Live at one of these locations is true, then let Y be the node at that location. If Y’s distance is still -1 then change Y’s distance to X’s distance + 1 and offer Y to the queue. Repeat the step with all 4 directions.

Eventually all the nodes that can be reached from BOB will have a positive distance and so the queue will become empty. Any node whose distance is -1 is not reachable from BOB. The distance for all other nodes is the length of the shortest path from BOB to the node.