Reference information about many standard Java classes appears at the end of the test. You might want to tear off those pages to make them easier to refer to while solving the programming problems.

Question 1. (2 points) The Java Collection Classes define several interfaces like `List`, `Set`, and `Map`, and then provide several implementations of these interfaces including `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`, and `TreeMap`. Why provide so many different ones? Why not just pick a "best" implementation for each kind of container and provide that?

There are several possible implementations for the different container class and no one is "best". Different implementations of an interface have different performance characteristics and which one is "best" depends on which operations need to be fast for a particular application.

Question 2. (3 points) Show that $3n^2 + 100\,n \log n + 30$ is $O(n^2)$.

We need to find some constant c such that

$$3n^2 + 100\,n \log n + 30 \leq c\,n^2$$

for all large values of n. If we pick $c = 4$ that will be sufficient.

Question 3. (4 points) (a) What is the *load factor* of a hash table? For good performance, should this number be large or small? Why?

The load factor of a hash table is defined to be the ratio *n*/*b*, where *n* is the number of items in the hash table (its size) and *b* is the number of buckets.

For good performance this ratio should be small. That implies that there are only a handful of items in each hash bucket, which means searches in the table will be fast.

(b) Java's class `Object` contains a pretty poor `hashCode()` method, which is overridden in many classes to provide a better implementation for those particular classes. Why did anyone bother to define `hashCode()` in class `Object` at all? Why not just include it in those classes that need to define it?

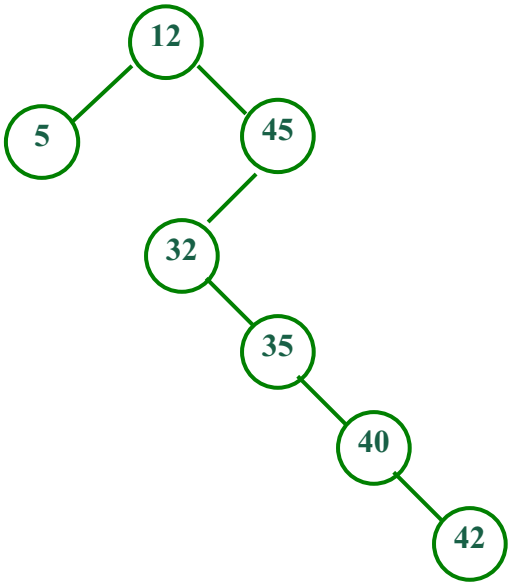
The main purpose for including `hashCode()` in class `Object` is so that it is part of the interface of every Java class. That allows any kind of object to be inserted in a container that is implemented using a hash table.

Question 4. (3 points) Write down the output that is produced when the following queue operations are executed. Assume that the queue contains `String`s only, so no casting is necessary when an object is removed from the queue.

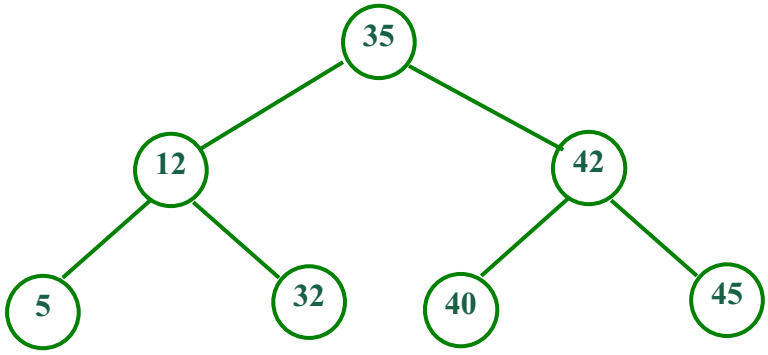
```
Queue q = new Queue();
q.insert("how");
q.insert("now");
q.insert("brown");
String s1 = q.remove();
String s2 = q.front();
q.insert("cow");
String s3 = q.remove();
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
```

how
now
now

Question 5. (6 points) (a) Draw the ***binary search tree*** that results when we add these numbers to an initially empty tree in this order: 12 45 32 35 40 5 42.



(b) Draw a ***balanced*** binary search tree containing the same numbers as in part (a).



Question 6. (6 points) During the quarter we've looked at several different ways of storing a collection of objects, in particular:

- Lists based on arrays
- Single- and double-linked lists
- Trees, particularly binary search trees
- Hash tables

For each of the following applications, indicate which data structure would be most appropriate and give a brief explanation justifying your choice in a technical way using, among other possibilities, the *O*() times needed to perform various operations.

(a) Inventory list for an on-line toy store. We need to be able to frequently look up information about a toy given its part number. Information about individual toys (number in stock, etc.) is often updated every time a toy is ordered. Toys are added and deleted from the list much less frequently.

Lookups for a part number should be efficient as possible. A hash table is best for this since lookups require constant time (assuming a decent implementation of the hash table).

(b) Audit trail for a credit card company. The audit trail records every transaction (charge, payment) that the company processes and there are hundreds of transactions per second. The audit trail itself is read sequentially and compared to the company's data about individual accounts to check that the account data is accurate.

A list, probably a linked list, is best for this since the operations that need to be efficient are appending a new item at the end and (less important) accessing the items in the list sequentially.

(c) List of high scores for a hand-held computer game. Whenever a game is over, the score is recorded. The ten highest scores can be displayed in decreasing order if the user clicks on the game button that requests this.

A simple sorted list is fine. There are only a small number of items in the list and inserting a new value is done very infrequently (whenever a game is over), so we don't need a fancy data structure to maintain a large sorted collection efficiently.

Question 7. (12 points) Consider the following class definitions:

```
class Matrix {
  public Matrix()    { System.out.println("Matrix constructor"); }
  public void jump() { System.out.println("Matrix jump"); }
  public void punch(){ System.out.println("Matrix punch"); }
}

public class Agent extends Matrix {
  public Agent()      { System.out.println("Agent constructor"); }
  public void gesture() { System.out.println("glare"); }
  public void punch() {
    gesture();
    System.out.println("pow!");
  }
}

public class AnnoyingAgent extends Agent {
  public AnnoyingAgent() { System.out.println(
                                "AnnoyingAgent constructor"); }
  public void jump()     { System.out.println("Aiieeee!"); }
  public void gesture()  { System.out.println("sneer"); }
}

class TheOne extends Matrix {
  public TheOne()    { System.out.println("TheOne constructor"); }
  public void punch(){ System.out.println("whap!"); }
  public void dodge(){ System.out.println("Bullet time!"); }
}
```

In each of the question parts on the next page, indicate the output produced when the group of statements in that part is typed into a DrJava interactions window. Assume that each group of statements starts with an empty, newly reset interactions window. If a statement results in either a compile-time or runtime error, show all of the output that would be produced prior to that statement, then describe the error.

Hint: Be sure you notice that there are `println` statements in the constructors.

You can tear out this page if you want so you can refer to it without having to flip back and forth while answering the questions.

(a)

```
TheOne neo = new TheOne();
```
Matrix constructor
TheOne constructor

```
neo.jump();
```
Matrix jump

```
neo.dodge();
```
Bullet time!

(b)

```
Matrix one = new Agent();
```
Matrix constructor
Agent constructor

```
one.punch();
```
glare
pow!

```
one.gesture();
```
Error: no gesture method in class Matrix

(c)

```
AnnoyingAgent three = new Agent();
```
Error: incompatible types

```
three.gesture();
```

```
three.jump();
```

(d)

```
Agent smith = new AnnoyingAgent();
```
Matrix constructor
Agent constructor
AnnoyingAgent constructor

```
smith.jump();
```
Aiieee!

```
smith.punch();
```
sneer
pow!

Question 8. (6 points) Consider the following method definitions. Assume that all of the two-dimensional arrays have *n* rows and *n* columns.

```
// zero out the given n x n matrix.
public void zap(double[][] matrix) {
   for (int r = 0; r < matrix.length; r++) {
      for (int c = 0; c < matrix[r].length; c++) {
         matrix[r][c] = 0.0;
      }
   }
}

// set the diagonal elements of matrix to 1.0
public void setDiagonal(double[][] matrix) {
   for (int k = 0; k < matrix.length; k++) {
      matrix[k][k] = 1.0;
   }
}

// return a new matrix of size n x n with 1's on the diagonal
// and 0's elsewhere.
public double[][] getDiagonal(int n) {
   double[][] ans = new double[n][n];
   zap(ans);
   setDiagonal(ans);
   return ans;
}

// return the product of two n x n matrices p and q
public double[][] multiply(double[][] p, double[][] q) {
   int n = p.length;
   double[][] ans = new double[n][n];
   for (int r = 0; r < n; r++) {
      for (int c = 0; c < n; c++) {
         ans[r][c] = 0.0;
         for (int k = 0; k < n; k++) {
            ans[r][c] = ans[r][c] + p[r][k]*q[k][c];
         }
      }
   }
   return ans;
}
```

For each of the following statements, give the running time as a function of *n* using *O*( )-notation. Assume that p and q are already-initialized arrays with *n* rows and *n* columns.

(a) `zap(p);` ***O*(n^2)**

(b) `double[][] newArray = getDiagonal(n);` ***O*(n^2)**

(c) `double[][] result = multiply(p,q);` ***O*(n^3)**

The next two questions concern Binary Search Trees containing integer values in their nodes. The nodes are defined by the following Java class.

```
public class IntBSTNode {     // one node an integer BST
   public int        value;   // data associated with this node
   public IntBSTNode left;    // left subtree; null if empty
   public IntBSTNode right;   // right subtree; null if empty
   // no constructor, since we don't need it
}
```

Question 11. (7 points) Complete the definition of method nLeaves below so it returns the number of leaf nodes in the tree whose root is given as its argument. (Hint: recursion ***is*** your friend.)

```
// return the number of leaves in the binary tree with root r
public int nLeaves(IntBSTNode r) {

   if (r == null) {
      return 0;
   } else if (r.left == null && r.right == null) {
      return 1;
   } else {
      return nLeaves(r.left) + nLeaves(r.right);
   }

}
```

(`IntBSTNode` definition repeated here to reduce the need for page flipping.)

```
public class IntBSTNode {     // one node an integer BST
   public int        value;   // data associated with this node
   public IntBSTNode left;    // left subtree; null if empty
   public IntBSTNode right;   // right subtree; null if empty
   // no constructor
}
```

Question 12. (7 points) Complete the definition of method `nLess(val)` below so it returns the number of nodes in the binary search tree containing a value strictly less than `val`. **For full credit,** your method **must not visit** any more nodes in the tree than are actually necessary (Hint: recursion ***is*** your friend.) (Hint: for this problem, it ***really*** helps to draw some diagrams and think about the possible cases before writing your code.)

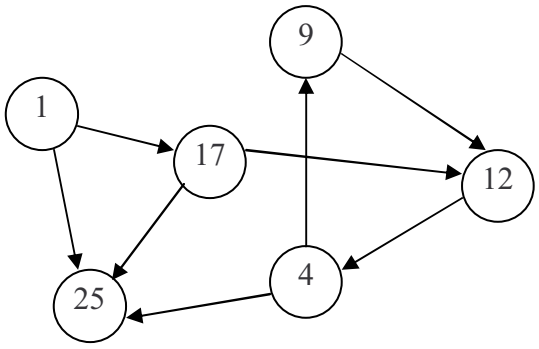
```
// return the number of nodes in the binary search tree
// with root r whose value field is less than val.  If the
// tree is empty, return 0.
public int nLess(IntBSTNode r, int val) {

   if (r == null) {
      return 0;
   } else if (r.value >= val) {
      return nLess(r.left, val);
   } else {
      return 1 + nLess(r.left, val) + nLess(r.right, val);
   }

}
```

Note: answers that used > instead of >= in the test received full credit. The important thing was whether the answer avoided exploring subtrees that couldn't possibly contain a value greater than the one being checked.

Question 13. (12 points) Trees are a special case of a more general data structure known as a *graph*. A graph consists of nodes and edges, just like in a tree, but the edges can connect any node to any other. For example, here is a small graph where each node has at most two edges connecting it to other nodes.



For this problem we are interested in finding out if, given a particular node, we can reach a node containing a specific value. In the above example graph, there is a path from the node containing 1 to the node that contains 12 (1-17-12). There is also a path from the node containing 17 to the node containing 9 (17-12-4-9). However there is no path from the node containing 12 to the nodes containing 1 or 17. Finally, there is no path from any node to a node containing 3, since 3 does not appear anywhere in the graph.

We will represent a node as follows.

```
/** A node in a graph with at most two adjacent nodes */
public class GNode {
   // instance variables
   private int value;    // the value held in this node
   private GNode next1, next2;
                         // The next nodes in the graph that can
                         // be reached directly from this node.
                         // Either or both can be null if there
                         // are fewer than 2 neighboring nodes

   private int status;              // status of this node,
                                    // defined as follows:
   private static int CLEAR = 0;    // initial status of node
                                    // before search starts

   // add any additional instance variables or constants that
   // you need below

   private static final int VISITED = 1;
                                    // status of a GNode that
                                    // has already been visited
                                    // during the search
```

(continued)

Question 13. (cont.) For this problem, complete the definition of method `canReach(val)` in class `GNode` below so it returns true if either the current `GNode` contains the specified value, or if there is some path from the current `GNode` to another `GNode` that contains the value.

You may assume that the instance variable `status` in every node in the graph has the value `CLEAR` when the search starts. You may change this variable during the search if you wish.

Hints: Be sure you don't get stuck in a cycle, where you keep visiting a sequence of nodes over and over. Don't panic if the solution turns out to be fairly short.

```
   /** (in class GNode)
    * Search for a given value in this graph.
 * @param val  The value we are looking for.
 * @return true if either this GNode's value field equals
 *         val, or if there is some path from this GNode to
 *         a GNode containing val. Return false otherwise.
 */
public boolean canReach(int val) {

   if (this.value == val) {
      // found it
      return true;

   } else if (this.status == VISITED) {
      // already been here - further search is futile
      return false;

   } else {
      // mark this GNode so we won't search from here again
      this.status = VISITED;

      // search from neighboring GNodes, if any
      return ((this.next1 != null && next1.canReach(val)) ||
              (this.next2 != null && next2.canReach(val)));

}
```

Note: a real search algorithm on a graph needs to figure out how to clear out the "VISITED" status before the next search. For this problem, it was enough to have a single search work properly.