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. (4 points) This code doesn't compile. Fix it. Be sure that with your fix, it does what its javadoc describes. (Note you may not need to add code in all the blank spaces – if you can get by with less, that's fine.)

```
/**
 * Peek at what's in a file by opening it and returning the first
 * line.  If anything goes wrong (no such file, error reading the
 * file,...) return null.
 * @param filename name of the file
 * @return the first line of the file, or null if anything went wrong
 */
public String peek ( String filename ) {


try {

  BufferedReader in
    = new BufferedReader( new FileReader( new File( filename ) ) );

  String line = in.readLine();

  in.close();

  return line;


} catch (IOException e) {
   return null;
}


}
```

The main issue is that we need a try/catch block to handle the exceptions that the file operations might throw. But a little care is needed to be sure there are no problems with scoping rules. In particular, if `line` is declared inside the try/catch, then it can't be referenced outside, which means there is a scoping problem if line is referred to in a return statement outside the try/catch.

Question 2. (2 points) The following code raises a number to a power by repeatedly multiplying by the number.

```
/** Raise x to the power n, for n >= 0. */
public double raise( double x, int n ) {
  double result = 1;
  for ( int i = 1; i <= n; i++ ) {
    result = result * x;
  }
  return result;
}
```

(a) (1 point) Exactly how many multiplications does this do when the power is some particular integer n? (Give an expression using n.) **n**

(b) (1 point) What is its complexity? **O(n)**

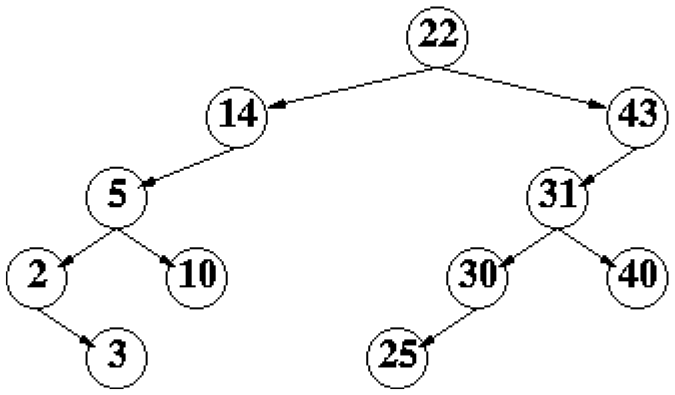
[During the grading session we decided that parts (c) and (d) were too much of a "trick" question, so we deleted those parts of the question from the grading and the total possible score.]

Question 3. (4 points) Put the following complexities in order of best (fastest) to worst (slowest):

O(n log n), $O(n^2)$, $O(2^n)$, O(log n)

O(log n) O(n log n) $O(n^2)$ $O(2^n)$

Question 4. (10 points) Answer the questions below about this binary tree.



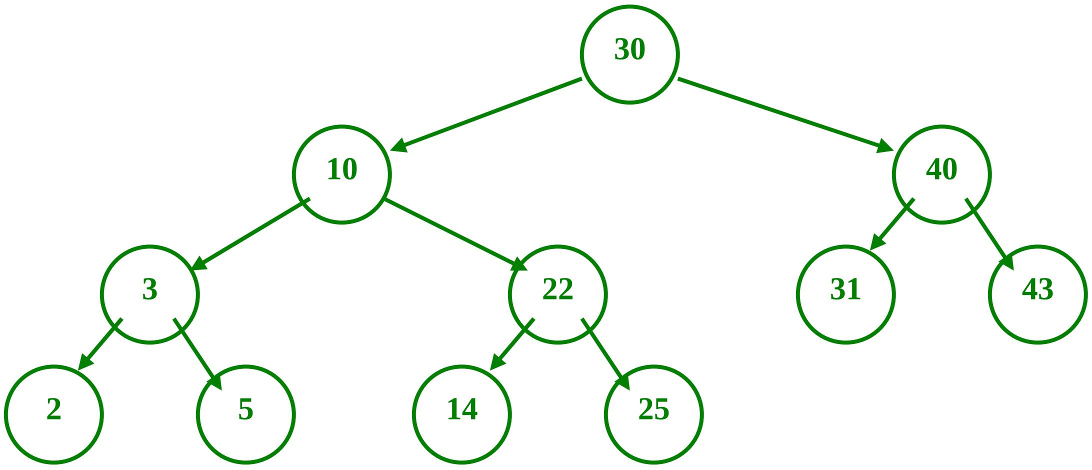
(a) (2 points) Which node(s) is(are) leaf nodes? **3 10 25 40**

(b) (1 points) What is the height of the node that contains 30? **4**

(c) (2 points) Is this a binary search tree? **yes**

(d) (3 points) Draw another tree below that contains the same numbers as the one above, is a binary search tree, and is as shallow (has the smallest height) as possible.

There are many possible binary trees of height 4, depending on what you pick as the root of the tree and each subtree. Here is one possibility.



(e) (2 points) Why is it an advantage for a binary search tree to be as shallow as possible?

If the tree is shallow (balanced), then the distance from the root to any leaf is O(log n), which means we can search the tree in O(log n) time. If the tree is not balanced, then the distance from the root to a leaf could be as long as the number of nodes in the tree, requiring O(n) time to search.

Question 5. (8 points) What is the **average** (expected) **time**, in *O*( ) notation, needed to determine whether a particular value appears in the following data structures, assuming that the data structure contains *n* values, and that an appropriate, fast algorithm is used.

i) Linked list **O(n)**

ii) Sorted array **O(log n)**

iii) Unsorted array **O(n)**

iv) Binary tree (not Binary Search Tree) **O(n)**

v) Binary Search Tree **O(log n)**

vi) Hash table (map) **O(1) [constant time]**

(b) Is the **worst-case** time needed to search for a value different from the average (expected) time for any of the data structures in part (a)? If so, which ones, and what is (are) the worst-case time(s) for each one?

Binary Search Tree O(n)

Hash table O(n)

Question 6. (8 points) In lecture we represented the nodes in a binary search tree containing integer values with the following data structure.

```
class BTNode {
   public int item;            // integer value in this node
   public BTNode left;         // left subtree, or null if none
   public BTNode right;        // right subtree, or null if none
}
```

Complete the definition of the recursive function nPos, below, so that it returns the *number of nodes* in a binary search tree whose item field is a positive integer (i.e., greater than 0). For full credit, you need to use recursion, and you also need to limit your search only to the parts of the tree that could contain positive values. (i.e., don't search parts of the tree that are known to only contain non-positive numbers.)

```
/** Return the number of nodes in the subtree with root r
 *  that contain positive integer values in their item field.
 */
public int nPos(BTNode r) {

   if (r == null) {
      return 0;
   } else if (r.item <= 0) {
      return nPos(r.right);
   } else {
      Return 1 + nPos(r.left) + nPos(r.right);
   }

}
```

Note: even if a node contains an item that is <= 0, there might be nodes in its right subtree that contain positive integers.

Question 7. (8 points) We spent a lot of time implementing classes that hold collections of objects. Here, you'll use one of the library collection classes – a LinkedList -- to implement another sort of class that holds objects – a stack. A stack is unusual because it only allows adding things and removing things at one end, just as though they were stacked one on top of the other – only the one on top can be removed, and you can only put another item on the top. That is, the last thing that got put into the stack is the first thing that comes out. A stack doesn't allow doing anything to other items on the stack besides the one on top.

Our Stack class will have two operations for adding and removing items: "push" adds an object to the top of the stack, and "pop" removes the top object from the stack and returns it. We'll give you the instance variable for the LinkedList -- you write push pop, and the constructor. Make sure they do what their javadoc says they will. Add anything you need to the constructor or instance variables.

```
public class Stack {

  private LinkedList list;


  /** Construct a new, empty stack */
  public Stack() {

      list = new LinkedList();

  }

  /**
   * Add an object to the front (or "top") of the stack.
   * @param object – the object to add
   */
  public void push( Object object ) {

      list.add(object);

  }

  /**
   * Remove and return the object at the front (or "top") of the stack.
   * @return the object at the top of the stack, if any
   * @throws NoSuchElementException if the stack is already empty
   */
  public Object pop() {

      if (list.size() == 0) {
         throw new NoSuchElementException();
      }
      return list.removeLast();

}
```

It is also possible to solve the problem using `LinkedList`'s `addFirst` and `removeFirst` methods. Because the underlying datastructure is a `LinkedList`, these operations take constant time, so this other implementation will be equally efficient as the one given above.

Question 8. (12 points) **Snark hunt**

You're hunting for the Snark, and want to find the shortest path to it. You have a map that you can use to plan your path. (We'll give you a class Map that represents the map – you just need to use it.) The map is a rectangular grid of squares, with numbered rows and columns starting at 0. You can only move to adjacent squares to the left, right, up, or down (if they exist), not diagonally. Some squares are blocked – the path can't go through a blocked square. One square contains the Snark. The map will tell you if a square is blocked or if the Snark is there. It's possible that you can't get to the Snark from your starting point.

The length of a path is the number of squares in the path, counting the starting square. For instance, if you start in row 2, column 3, and go right to 2, 4, down to 3, 4, right to 3, 5, that's a path of length 4.

While you're searching, it would be useless to go back into a square that's already in the path you're trying out – that would just add extra length. The map lets you mark squares so you can tell if you've already tried them. The map you'll be given initially has no marks in it.

Write a method that starts at some square in the grid, given its row and column, and finds a best move to make, that is, it returns the *first* step along a shortest path to the Snark. This return a String, "left", "right", "up", or "down" to indicate the best move, or "here" if it's already at the Snark. If it can't get to the Snark at all, it should just return an empty string.

You can add helper methods and classes if you need them, but you cannot define any instance or static variables, or change the Map class. For full credit, you must use recursion. It's ok if you have some redundant code.

A description of the Map methods is on the next page, and after that is the declaration and javadoc for the method you'll write, and space for helper methods or classes on a blank page after that.

Methods of class Map. (You can remove this page for reference while writing the solution)

```
/**
 * Returns the width of the grid.
 * @return the number of columns in the grid
 */
public int width()

/**
 * Returns the height of the grid.
 * @return the number of rows in the grid
 */
public int height()

/**
 * Make a copy of this Map.  Nothing you do to the copy will change the original.
 * @return a copy of this Map.
 */
public Map copy()

/**
 * Is this square blocked?
 * @param row – row of the square being asked about
 * @param col – column of the square being asked about
 * @return true if the square is blocked
 * @throws ArrayIndexOutOfBounds if row or col are outside the grid
 */
public boolean isBlocked( int row, int col )

/**
 * Is the Snark in this square?
 * @param row – row of the square being asked about
 * @param col – column of the square being asked about
 * @return true if the Snark is in this square
 * @throws ArrayIndexOutOfBounds if row or col are outside the grid
 */
public boolean snarkHere( int row, int col )

/**
 * Mark this square.
 * @param row – row of the square being asked about
 * @param col – column of the square being asked about
 * @throws ArrayIndexOutOfBounds if row or col are outside the grid
 */
public void mark( int row, int col )

/**
 * Is this square marked?
 * @param row – row of the square being asked about
 * @param col – column of the square being asked about
 * @return true if the square is marked
 * @throws ArrayIndexOutOfBounds if row or col are outside the grid
 */
public boolean isMarked( int row, int col )
```

```
/**
 * Using the given Map as a guide, choose the best move to make to get to the Snark
 * by a shortest path, starting from the given row and column.  The path cannot go
 * through any blocked squares.  If the Snark can be reached, return a String with
 * "here" if the Snark is in this square, or the best move to make, "left", "right",
 * "up", or "down".  If there is no path to the Snark, return an empty String (not
 * null).  (Note this only returns the first move along a shortest path.)
 *
 * @param row – the row of the square to start from
 * @param col – the column of the square to start from
 * @param map – a Map that tells what squares are blocked or occupied by the Snark
 * @return a String giving the first step along a shortest path to the Snark through
 * unblocked squares, or an empty String if there was no path to the Snark
 */
public String moveTowardSnark( int row, int col, Map map ) {

   if (map.snarkHere(row, col) {
      return “here”;
   }

   String bestDirection = “”;
   int bestPathLength = map.width()*map.height()+1; // longer than any possible path
   Map copy = map.copy();
   copy.mark(row, col);
   int trialPath = nMovesToSnark(row, col-1, copy);
   if (trialPath >= 0 &&  trialPath < bestPathLength) {
      bestDirection = “left”
      bestPathLength = trialPath;
   }
   trialPath = nMovesToSnark(row, col+1, copy);
   if (trialPath >= 0 &&  trialPath < bestPathLength) {
      bestDirection = “right”
      bestPathLength = trialPath;
   }
   trialPath = nMovesToSnark(row-1, col, copy);
   if (trialPath >= 0 &&  trialPath < bestPathLength) {
      bestDirection = “up”
      bestPathLength = trialPath;
   }
   trialPath = nMovesToSnark(row+1, col, copy);
   if (trialPath >= 0 &&  trialPath < bestPathLength) {
      bestDirection = “down”
      bestPathLength = trialPath;
   }
   return bestDirection;

}
```

```
// return the number of moves needed to find the snark along the best
// possible path starting at position row, col.  Return 0 if the snark
// is located at that position on the map; return the number of moves
// needed on the shortest path if it is possible to reach the snark
// starting at row, col; return -1 if no path is possible.

int nMovesToSnark(int row, int col, Map map) {

   // base cases
   if (row < 0 || col < 0 || row >= map.height() || col >= map.width()) {
      // off the grid, can’t continue
      return -1;
   if (map.isBlocked(row, col) || map.isMarked(row, col)) {
      return -1;
   else if (map.snarkHere(row, col)) {
      return 0;

   } else {  // recursive case
      // mark this square so we don’t visit it again and create a fresh copy
      Map copy = map.copy();
      copy.mark(row, col);

      // recursively search in each possible direction
      // (this implementation will check for running off the board in the
      //  recursive call and return -1 in that case)
      int minLength = copy.width() * copy.height() + 1;  // longer than any path
      for (int r = row-1; r <= row+1; r = r+2) {
         for (int c = col-1; c <= col+1; c = c+2) {
            int dist = nMovesToSnark(r,c,copy);
            if (dist != -1 && dist < minlength) {
               minLength = dist;
            }
         }
      }

      // if minLength was updated, return path length, else no path was found
      if (minLength < copy.width() * copy.height() + 1) {
         return minLength + 1;   // add 1 to count for map(row,col)
      } else {
         return -1;
      }
   }
}
```

Here's a second solution that uses a similar strategy to find the paths but differs in many ways.

```
// Offsets of the squares surrounding "this" one.  First offset is row,
// 2nd is column.
private int[][] neighbors = { {1,0}, {0,-1}, {0,1}, {-1,0} };

// Moves corresponding to the above.
private String[] moves = { "down", "left", "right", "up" };

public String moveTowardSnark( int row, int col, Map map ) {

  // Check for an end case.  We're at an end case if 1) we're at the
  // Snark, blocked, or marked.  If at the Snark, the length of this
  // one-square path is 1.  Blocked or marked squares mean this path
  // is no good -- return the maximal int.
  if ( map.snarkHere( row, col ) ) {
    return "here";
  }
  if ( map.isBlocked( row, col ) || map.isMarked( row, col ) ) {
    return "";
  }

  // Here, we have an unblocked move.  Copy our Map, mark our square...
  Map markedMap = map.copy();
  markedMap.mark( row, col );
  int bestLength = Integer.MAX_VALUE;  // Will be replaced by any real path.
  String bestMove = "";

  // ...and try out moves.
  for ( int i = 0; i < neighbors.length; i++ ) {

    // Get the move location.
    int nextRow = row + neighbors[i][0];
    int nextCol = col + neighbors[i][1];

    // Make sure it's in range of the map.
    if ( nextRow >= 0 && nextRow < map.height() &&
        nextCol >=0 && nextCol < map.width() ) {

      // It's safe -- do the recursive call.
      int length = bestPathLength( nextRow, nextCol, markedMap );

      // Is that better?
      if ( length < bestLength ) {
        bestLength = length;
        bestMove = moves[i];
      }
    }
  }

  // Return the best move.
  return bestMove;
}
```

```
// Helper for moveTowardSnark.  Returns the length of the best path from
// the given row and column.
private int bestPathLength( int row, int col, Map map ) {
  // Check for an end case.  We're at an end case if 1) we're at the
  // Snark, blocked, or marked.  If at the Snark, the length of this
  // one-square path is 1.  Blocked or marked squares mean this path
  // is no good -- return the maximal int.
  if ( map.snarkHere( row, col ) ) {
    return 1;
  }
  if ( map.isBlocked( row, col ) || map.isMarked( row, col ) ) {
    return Integer.MAX_VALUE;
  }

  // Here, we have an unblocked move.  Copy our Map, mark our square...
  Map markedMap = map.copy();
  markedMap.mark( row, col );
  int bestLength = Integer.MAX_VALUE;  // Will be replaced by any real path.

  // ...and try out moves.
  for ( int i = 0; i < neighbors.length; i++ ) {
    // Get the move location.
    int nextRow = row + neighbors[i][0];
    int nextCol = col + neighbors[i][1];

    // Make sure it's in range of the map.
    if ( nextRow >= 0 && nextRow < map.height() &&
        nextCol >=0 && nextCol < map.width() ) {

      // It's safe -- do the recursive call.
      int length = bestPathLength( nextRow, nextCol, markedMap );

      // Is that better?
      if ( length < bestLength ) {
        bestLength = length;
      }
    }
  }

  // Fall out of recursion.  Unless we're already maxed out, count
  // one for our square.
  if ( bestLength == Integer.MAX_VALUE ) {
    return bestLength;
  }
  return bestLength + 1;
}
```