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 ) {




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




  String line = in.readLine();




  in.close();




  return line;




}
```

Question 2. (6 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;
}
```

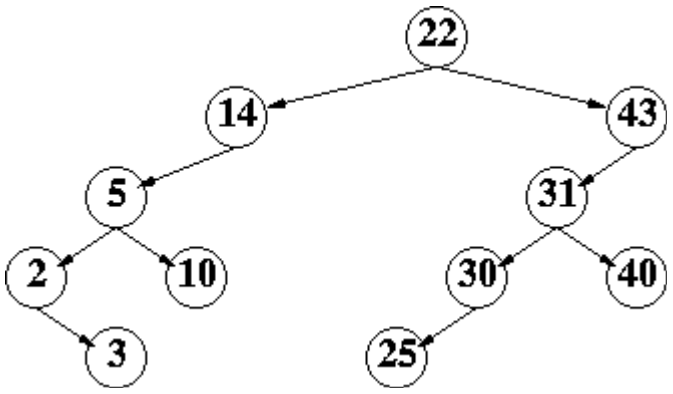
(a) (1 point) Exactly how many multiplications does this do when the power is some particular integer n? (Give an expression using n.)

(b) (1 point) What is its complexity?

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)

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



(a) (2 points) Which node(s) is(are) leaf nodes?

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

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

(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.

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

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

ii) Sorted array

iii) Unsorted array

iv) Binary tree (not Binary Search Tree)

v) Binary Search Tree

vi) Hash table (map)

(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?

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) {









}
```

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() {









  }

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









}
```

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 ) {









}
```

Java Reference Information

Feel free to detach these pages and use them for reference as you work on the exam. This information is identical to the reference information on the 2nd midterm. You may not need most (or even all) of it to answer the questions on this exam.

`class` ***BufferedReader***

`String readline()`	Return next line from input stream, or `null` if no more input. Can throw `IOException`.

`class` ***PrintWriter***

`void print(arg)`	Print arg to the `PrintWriter` stream. The parameter can be any type
`void println()`	Terminate the current output line and move to the beginning of the next. line
`void println(arg)`	Print `arg`, then advance to the beginning of the next line

`class` ***String***

All of the search methods in class `String` return -1 if the item is not found

`int length()`	length of this string
`int indexOf(char ch)`	first position of `ch`
`int indexOf(char ch, int start)`	first position of `ch` starting from `start`
`int indexOf(String str)`	first position of `str`
`int indexOf(String str, int start)`	first position of `str` starting from `start`
`int lastIndexOf(char ch)`	last position of `ch`
`int lastIndexOf(char ch, int start)`	last position of `ch` searching backward from `start`
`int lastIndexOf(String str)`	last position of `str`
`int lastIndexOf(String str, int start)`	last position of `str` searching backward from `start`
`String substring(int start)`	substring of this string from position `start` to the end
`String substring(int start, end)`	substring of this string from position `start` to `end-1`
`String trim()`	copy of this string with leading and trailing whitespace deleted

All *Collection* interfaces (*List*, *Set*) and classes (*ArrayList*, *LinkedList*, *HashSet*, *TreeSet*)

```
boolean add(Object obj)
boolean addAll(Collection other)
void clear()
boolean contains(Object obj)
Iterator iterator()
boolean remove(Object obj)
int size()
Object[] toArray()   // return an array containing all the
                     // elements in this collection
```

In addition, all `Collection` classes provide a constructor that takes another `Collection` as a parameter and creates a new collection whose initial contents are copied from that parameter. (i.e., `public ArrayList(Collection c)`, and similarly for other classes.)

Additional methods in *List*, *ArrayList*, *LinkedList*

```
add(int position, Object obj)
remove(int position)
```

`Additional methods in` *LinkedList*

```
getFirst(), getLast(), addFirst(), addLast(), removeFirst(),
     removeLast()
// these retrieve, add, and remove items at either end of the list.
// remove returns the item removed from the list
```

Map, *HashMap*, *TreeMap*

```
Object put(Object key, Object value)
Object get(Object key)
Object remove(Object key)
Set keySet()
Collection values()
int size()
```

arrays

If `a` is a Java array, `a.length` is the number of elements in that array. If `m` is a 2-dimensional Java array, `m[k]` refers to row `k` of the array, and `m[k].length` is the length of that row (which is the same for all rows in a normal, rectangular array).

Exceptions

Some standard exceptions that might be useful: `IllegalArgumentException`, `IndexOutOfBoundsException`, `NoSuchElementException`, `NullPointerException`.