1.00 Introduction to Computers and Engineering Problem Solving

Final Exam May 21, 2003

Name:	
Email Address:	
TA:	
Section:	

You have three hours to complete this exam. For coding questions, you do not need to include comments, and you should assume that all necessary files have already been imported. Final exam is open book, open notes. No laptops, calculators, cell phones or other electronics are allowed. Sharing of notes or books is not allowed.

Question	***Points***
Question 1	**/ 5**
Question 2	**/ 10**
Question 3	**/ 20**
Question 4	**/ 10**
Question 5	**/ 10**
Question 6	**/ 25**
Question 7	**/ 20**
Total	**/ 100**

Problem 1. Exceptions (5 points)

Which of the following statements is true given the program ExceptionTest? (Circle only one answer.)

a. The program will not compile.

b. The program will compile. If we run it, it will throw an exception before outputting any results.

[✓ c.] The program will compile. If we run it, it will output some results, but then throw an exception.

d. The program will compile and run without throwing any exceptions.

```
public class ExceptionTest {
    public static void main(String args[]) {
        String[] greek = {"Alpha", "Beta", "Gamma"};
        System.out.println(greek[1]);
        System.out.println(greek[3]);
    }
}
```

Problem 2. Abstract Classes & Inheritance (10 points)

Enraged by the high price of software, you decide to martyr yourself by writing a free Java version of Photoshop for the world, which you call “jimp”. You’d like for the user to be able to scale, and rotate shapes such as rectangles, triangles, circles, and squares. You decide to create an abstract Shape class to represent the commonalities these shapes will share in your program.

```
// Shape.java
1 public abstract class Shape
2 {
3  // x,y coordinates of upper left corner of Shape's bounding box
4  private int xCoord;
5  private int yCoord;
6  private final String type; // rectangle, circle, triangle, etc
7
8  public Shape(int x, int y, String t)
9  {
10    xCoord=x;
11    yCoord=y;
12    type=t;
13 }
14
15  // get methods omitted
16  public abstract void scale(double factor);
17  public final void move(int newX, int newY)
18  {
19    xCoord=newX;
20    yCoord=newY;
21  }
22
23  public String toString(){
24    return "SHAPE:"+type+",("+xCoord+","+yCoord+")";
25  }
26 }
```

You then define a `Rotatable` interface and a `Rectangle` class as follows. The `Rectangle` class contains a `main()` method for unit testing.

```
1 // Rotatable.java
2 public interface Rotatable{
3   public void rotate(double theta);
4 }
```

```
1 // Rectangle.java
2 public class Rectangle extends Shape implements Rotatable
3 {
4  private int width;
5  private int height;
6
7  // The type of a Rectangle is "rectangle".
8  public Rectangle(int x, int y, int w, int h)
9  {
10      // Part A.
11 }
12
13  public String toString(){
14    return super.toString() + "," + width + "," + height;
15  }
16  public void move(int newX, int newY){super.move(newX, newY);}
17  public void scale(double factor){...} // implementation omitted
18  public void rotate(double theta){...} // implementation omitted
19
20  public static void main(String[] args)
21  {
22    Rotatable r1=new Rectangle(25,25,5,5);
23    Shape r2=new Rectangle(50,50,12,12);
24    Rectangle r3=new Shape(75,75,"rectangle");
25    Shape r4=new Shape(100,100,"rectangle");
26
27    r2.rotate(45);
28
29    System.out.println(r1);
30    System.out.println(r2);
31
32    System.exit(0);
33  }
34 }
```

A. Complete the constructor for the Rectangle class.

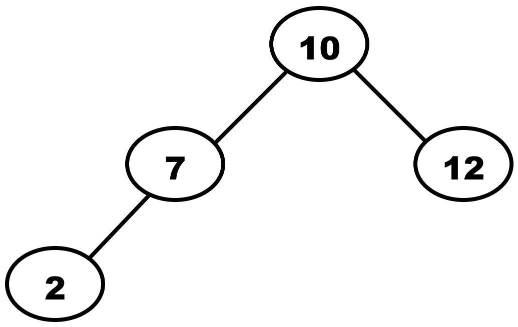
```
super(x, y, "rectangle");
width = w;
height = h;
```

B. Answer the following by circling "T" for true, or "F" fo false.

(i)	Line 16 of `Rectangle.java` will cause a compilation error.	[✓ T]	F
(ii)	Line 22 of `Rectangle.java` will cause a compilation error.	T	[✓ F]
(iii)	Line 23 of `Rectangle.java` will cause a compilation error.	T	[✓ F]
(iv)	Line 24 of `Rectangle.java` will cause a compilation error.	[✓ T]	F
(v)	Line 25 of `Rectangle.java` will cause a compilation error.	[✓ T]	F
(vi)	Line 27 of `Rectangle.java` will cause a compilation error.	[✓ T]	F
(vii)	Line 29 of `Rectangle.java` will cause a compilation error.	T	[✓ F]
(viii)	Line 30 of `Rectangle.java` will cause a compilation error.	T	[✓ F]

Problem 3. Binary Search Tree Insertion (5 points)

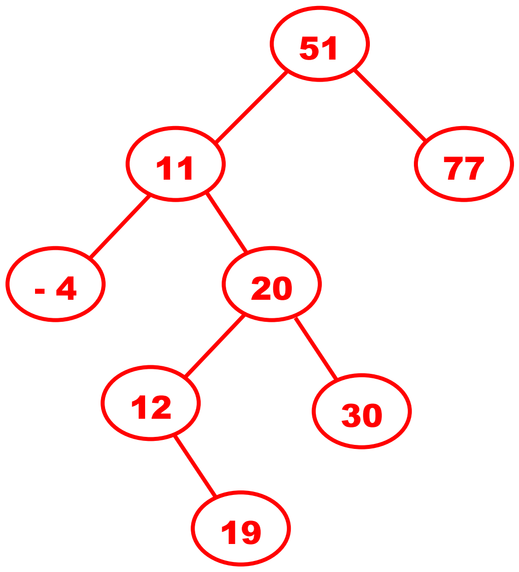
In a binary search tree (with no duplicate values), the data values of all descendants to the left of any node are less than the data value stored in that node, and all descendants to the right have greater data values. Here is a binary search tree that contains a single integer at each tree node.



Starting with an **empty tree** (don't use the one above), diagram the resulting binary search tree if we added nodes with the following keys in this order:

51, 11, 77, 20, -4, 30, 12, 19

Your diagram must follow the simple convention of the tree above.



Problem 4. Binary Search Tree Fun (25 points)

The class defined below, `DoubleTree`, is a binary search tree that can only contain `double`s.

```
public class DoubleTree {
     int size = 0;
     Node root = null;

     public DoubleTree() {
     }

     public void add(double d) {
          if ( root == null ) {
               root = new Node(d);
               size++;
          }
          else {
               if ( root.add(d) ) {
                    size++;
               }
          }
     }

     public double treeTotal() {
          if (root != null) return root.total();
          else return 0;
     }

     private static class Node {
          double value;
          Node left = null;
          Node right = null;
          Node(double d) {
               value = d;
          }
          public boolean add(double d) {
               if ( d > value ) {
                    if ( right == null ) {
                         right = new Node(d);
                         return true;
                    }
                    else {
                         return right.add(d);
                    }
               }
               else if ( d < value ) {
                    if ( left == null ) {
```

```
                              left = new Node(d);
                              return true;
                         }
                         else {
                              return left.add(d);
                         }
                    }
                    return false;
               }
               public double total() { /* part A */ }
          } // End Node
}  // End DoubleTree
```

You must implement several methods below of the Node and DoubleTree classes.

a) Implement an instance method in the `Node` class that will return the sum of all the `double` values contained in the sub-tree rooted at this node.

```
public double total() { /* this is a Node method */
  double tot = value;
  if (left != null) tot += left.total();
  if (right != null) tot += right.total();
  return tot;
}
```

b) Implement an instance method in the `DoubleTree` class that will return the average of all the `doubles` in the tree. This method must throw a `NoSuchElementException` if it is invoked on an empty `DoubleTree`.

```
public double average() { /* this is a DoubleTree mtd */
  if (root == null)
     throw new NoSuchElementException();
  return root.total() / size;
}
```

c) Implement an instance method in the `DoubleTree` class that returns the smallest `double` in the tree. This method must throw a `NoSuchElementException` if it is invoked on an empty `DoubleTree`. (Do not create any additional methods or data members in your solution.)

```
public double min() { /* this is a DoubleTree method */
```

```
if (root == null)
     throw new NoSuchElementException();
Node n = root;
while (n.left != null)  n = n.left;
   return n.value;
```

```
}
```

Problem 5. More Exceptions (10 points)

Consider the following code:

```
public class TryException {
   public static void main(String[] args) {
       int value;
       int [] anArray = {5, 6, 7};
       try {
           int index = Integer.parseInt(
            JOptionPane.showInputDialog("Index: "));
           value = anArray[index];
           System.out.println("Value is "+value);
       }
       catch (IndexOutOfBoundsException e) {
           System.out.println("Wrong index! Input 0, 1, or 2");
       }
       System.out.println("DONE.");
       System.exit(0);
   }
}
```

Answer the following questions by circling YES or NO and filling in the blank.

(a) If the user inputs "2", is an exception thrown? YES [✓ NO]

If the program outputs any result, what will it be? Write your answer in the box.

```
Value is 7
DONE
```

(b) If the user inputs "4", is an exception thrown? [✓ YES] NO

If the program outputs any result, what will it be? Write your answer in the box.

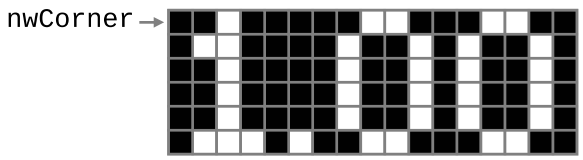
```
Wrong index! Input 0, 1, or 2
DONE
```

(c) If the user inputs "2.5", is an exception thrown? [✓ YES] NO

If the program outputs any result, what will it be? Write your answer in the box.

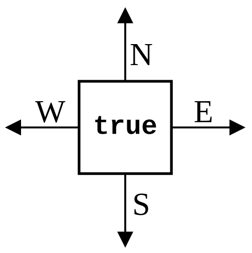
(no result, but an uncaught exception error will be sent to System.err)

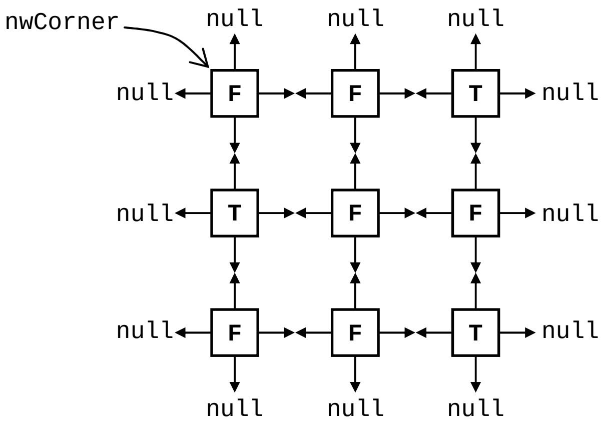
Problem 6. Inner Classes and Linked Lists (25 points)



In this problem we are working with a grid of lights, like those used as electronic signs. Each element of the light is a "Bulb"; its value is **`true`** if the element is on, and **`false`** if it is off. The picture above shows an example of the grid.

At the right is a picture of one `Bulb` whose value is `true`. It has references to the surrounding `Bulb` objects to the North, South, East and West. If element is at the edge of the grid, then one or more of these references may be null. You can assume that `Bulb`s are connected together correctly in this problem, and that the grid is rectangular. An example of a 3x3 grid is shown below.





The `LightGrid` and `Bulb` classes are given below.

```
public class LightGrid {

    class Bulb {
        boolean value;
        Bulb n, s, e, w;    /* all null by default */
        Bulb(boolean v) { value = v; }
    }

    Bulb nwCorner; /* reference to the upper left Bulb */

    public LightGrid(int rowCount, int colCount)
     /* constructor body omitted */...
}
```

To reduce clutter, we have omitted the body to the `LightGrid` constructor, but you should assume that this constructor exists and that it creates a `LightGrid` object with the number of rows and columns passed in.

The `LightGrid` class has only one member variable, `nwCorner`, which is a reference to the northwesternmost `Bulb` in the grid, as shown in the diagrams. You may not add any additional member variables to either the `LightGrid` or the `Bulb` classes.

(a) Complete the `getValue` method for the `LightGrid` class. `getValue` should return the value of the `Bulb` at row `row` and column `col`. The northwest `Bulb` is at row 0, column 0. You may not add, assume, or wish into existence any new member variables or methods. Your method does not need to do error checking; assume that the Bulb at row, col exists.

```
/* (a LightGrid method) */
public boolean getValue(int row, int col) {

         Bulb where = nwCorner;
         while (row > 0) { where = where.s; row--; }
         while (col > 0) { where = where.e; col--; }
         return where.value;

}
```

(b) Complete the `percentOn` method, which returns the percentage of `Bulb` objects whose value is `true`. If all `Bulbs` are on, it should return 1.0. If all `Bulbs` are off, it should return 0.0. You may not add, assume, or wish into existence any new member variables or methods.

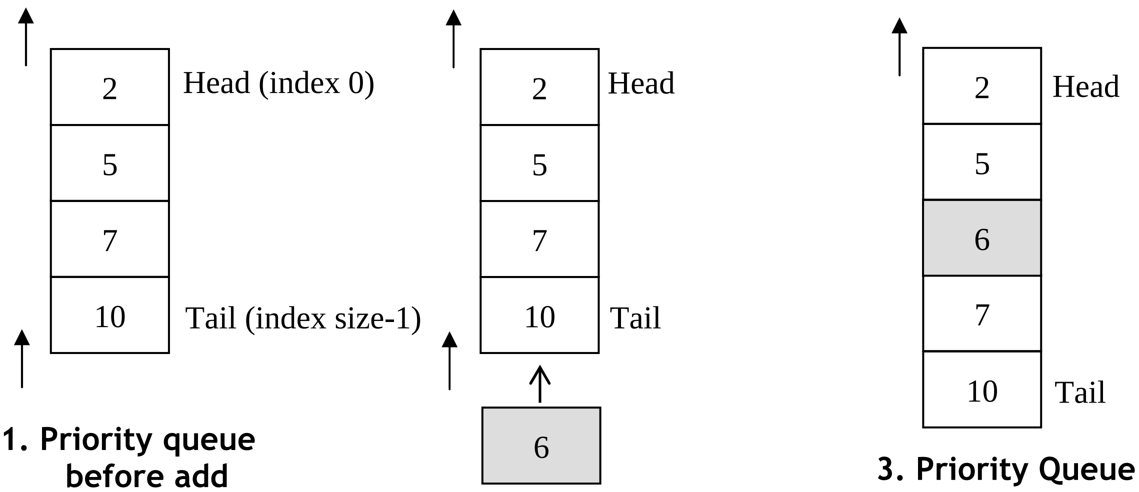
```
public double getPercentOn() { /*(a LightGrid method)*/
```

```
        int count = 0;
        int onCount = 0;
        Bulb row= nwCorner;
        while (row != null) {
            Bulb column = row;
            while (column != null) {
                if (column.value) onCount++;
                count++;
                column = column.e;
            }
            row = row.s;
        }
        return ((double)onCount)/count;
```

```
}
```

Problem 7. Priority Queues (20 points)

In this problem, you will use a sorted `Vector` to implement a "priority queue". This priority queue will only store objects of the `Integer` class in ascending order. In a priority queue, elements with the highest priority (lower integer values) are stored at the head of the queue. The priority queue always maintains the order of the elements according to their priority levels. Just like a regular queue, all the elements are initially added at the tail, and removed from the head. The `IntegerPQ` class assigns higher priorities to smaller numbers, which means that it will always store `Integer` objects in ascending order from the head to the tail. The diagram below shows how an `Integer` is added to the priority queue.



1. Priority queue before add

2. add(6)

3. Priority Queue after add

You may find the following methods of `Vector` useful in your solution.

Selected java.util.Vector methods
`public void addElement(Object obj)` Adds the specified element to the end of the Vector.
`public void add(int index, Object obj)` Adds the specified element at the specified position in the Vector.
`public void removeElementAt(int index)` Removes the element at the specified index from the Vector.
`public Object remove(int index)` Removes and returns the element at the specified position.
`public Object get(int index)` Returns the element at the specified position in the Vector.
`public void clear()` Removes all of the elements from this Vector.

(a) Complete the `add()` method below.

(b) Complete the `remove()` method below.

(c) Complete the `clear()` method below.

(d) Complete the `main()` method as directed below to output the contents of the priority queue.

(e) Write the output of the `main()`method in the space provided below.

```
public class IntegerPQ {
    // Vector used to store elements
    private Vector pq;

    public IntegerPQ() { pq = new Vector(); }

    // Part a. Add an Integer object to the priority queue.
    // You should assume that pq is sorted with highest
    // priority items (lowest values) at the beginning of
    // the pq Vector. pq must also be sorted after add is
    // called. Important: The head of the queue is at pq[0].
    public void add( Integer i ) {

     int position;
     for (position = 0; position < pq.size(); position++)
     {
        Integer j = (Integer)pq.get(position);
        if (j.compareTo(i) > 0) {
           pq.add(position, i);
           return;
        }
     }
     pq.add(i); /* adds to end */

    }
    // This method removes an element from Vector and
    // returns it
    public Integer remove() throws NoSuchElementException {

       // Part b
       // If queue is empty, throw a NoSuchElementException
       // Else, remove the element at the head and return it

         if (isEmpty())
           throw new NoSuchElementException();

         return (Integer)pq.remove(0);

    }
    public int size() { /* omitted */ }

    public boolean isEmpty() { /* omitted */ }

    public void clear() {

        // Part c
        // Complete the clear() method

         pq.clear();

    }

}
```

Now that **IntegerPQ** is complete, you will complete **IntegerPQTest** class, which has a **main()** method, to test your implementation.

```
public class IntegerPQTest {

    public static void main( String[] args ) {

        IntegerPQ priorityQ = new IntegerPQ();

        // Add integers
        priorityQ.add( new Integer( 23 ) );
        priorityQ.add( new Integer( 71 ) );
        priorityQ.add( new Integer( 10 ) );
        priorityQ.add( new Integer( 89 ) );
        priorityQ.add( new Integer( 63 ) );
        priorityQ.add( new Integer( 41 ) );
        priorityQ.add( new Integer( 55 ) );

        // remove elements
        priorityQ.remove();
        priorityQ.remove();

        // Add more integers
        priorityQ.add( new Integer( 99 ) );
        priorityQ.add( new Integer( 22 ) );
        priorityQ.add( new Integer( 15 ) );

        // Part d
        // Remove and print out all the Integer objects in
        // priorityQ. (Do not add any new methods.)
```

```
while (!priorityQ.isEmpty())
   System.out.println(priorityQ.remove());
```

```
        System.exit( 0 ) ;
    }
}
```

```
// Part e
// What is the output after executing the main method?
```

```
15
22
41
55
63
71
89
99
```

```
// End of test
// Thank you and enjoy the summer
```