1.00 Introduction to Computers and Engineering Problem Solving

Final Examination - May 19, 2004

Name:	
E-mail Address:	
TA:	
Section:	

You have 3 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.

Good luck!

Question	***Points***
Question 1	**/ 10**
Question 2	**/ 10**
Question 3	**/ 20**
Question 4	**/ 20**
Question 5	**/15**
Question 6	**/ 25**
Total	**/ 100**

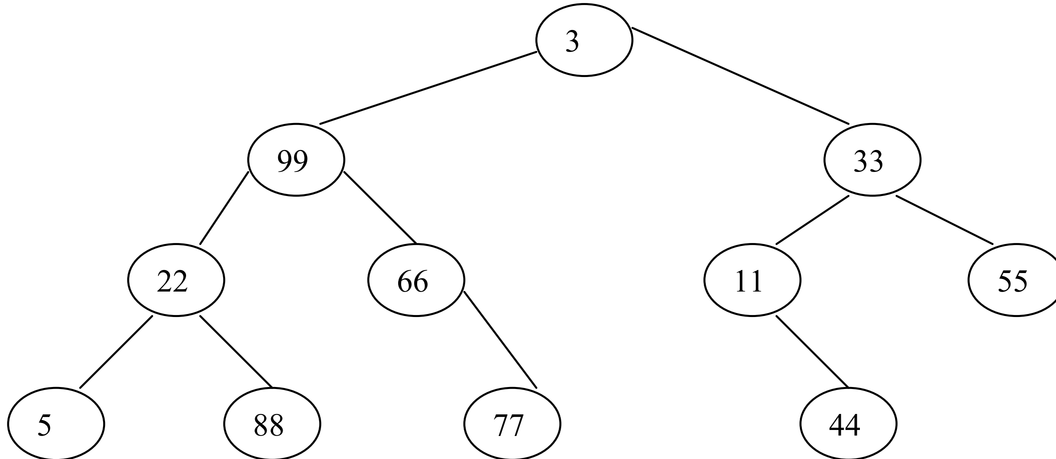
THIS PAGE INTENTIONALLY LEFT BLANK

Question 1. Tree traversal (10 points)

After several terms of 1.00, the TAs decided they were tired of the inorder and postorder traversals taught in the course. After much thought, they developed new, *modified* inorder and postorder traversals. These new traversals are:

A *modified* inorder traversal performs the following, recursively:
- Traverse the right subtree
- Visit the node
- Traverse the left subtree

A *modified* postorder traversal performs the following, recursively:
- Traverse the right subtree
- Traverse the left subtree
- Visit the node



a) If the tree shown is traversed in ***modified*** inorder, starting at the root of the tree, what is the sequence of nodes visited?

Answer:

b) If the tree shown is traversed in ***modified*** postorder, starting at the root of the tree, what is the sequence of nodes visited?

Answer:

Question 2. Circular linked list (10 points)

A circular linked list is a linked list in which the last node refers to the first node. By convention, `first` is a reference to the first node in the linked list. The program below builds a circular list and manipulates it. Your task is to predict the output at three stages in the program.
***Hint*:** Trace the program by drawing the linked list at intermediate steps (draw the list with the `data` and `next` fields for every node, and with the `first` and `last` references)

```
public class CircularList {

    public static class Node {
        int data;
        Node next;

        Node (int d, Node n) {
            data = d;
            next = n;
        }
    }

    public static void main(String[] args)
    {
         int i;
         int N = 5, M = 3;
```

```
         // Step (a)
         Node first = new Node(1, null);
         first.next = first;
         Node last = first;
         // End of Step (a)
```

```
         // Step (b)
         for (i = 2; i<= N ; i++) {
             last.next = new Node (i,first);
             last= last.next;
         }
         // End of Step (b)
```

```
         // Step (c): What's happening here?
         while (last!= last.next) {
             for (i=1; i<M; i++)
                 last= last.next;

             last.next = last.next.next;
         }
         // End of Step (c)
```

```
        // Output the result
        System.out.println (last.data);
    }
}
```

```
// Draw the list after step (a) finishes
```

```
// Draw the list after step (b) finishes
```

```
// What is the final output from step (c)?
```

Question 3. Computing grade ranges (20 points)

In order to assign overall letter grades for 1.00, the teaching staff needs to organize students' numerical scores.

First, the staff places all student scores into an `int[]` array called `grades`. Next, they generate an `int[]` array, `cumulative`, that contains the cumulative number of student scores at or below that score. All scores are within the range 0 to 100. The TAs then use this array to determine how many students scored between grades *a* and *b*.

Example: There are *N=10* students in a class, with scores in the `grades` array as follows:
67, 68, 65, 70, 74, 70, 69, 74, 68, 70

Running the `cumulativeGrades` method returns an array of cumulative grades. An example output is shown in row 3 of the table below.

grades	0	…	64	65	66	67	68	69	70	71	72	73	74	75	….	100
frequency	0	….	0	1	0	1	2	1	3	0	0	0	2	0	….	*0*
cumulative grades	0	…	0	1	1	2	4	5	8	8	8	8	10	10	…	*N= 10*

Using the cumulative array it is possible to count the number of students with grades in a range between values a and b. In the example above, the number of grades between a=67 and b=73, inclusive, is 7, the difference between the cumulative number at 73 and 66 (one less than 67).

Part 1 : cumulativeGrades

Please complete the `cumulativeGrades` method. The method takes an `int[]` array containing all the grades for the class as its argument and returns an `int[]` array containing the cumulative grade distribution.

```
public static int[] cumulativeGrades (int[] grades){

}
```

Part 2: numGradeRange

Complete the numGradeRange method that calculates the number of scores in a given range. The method takes as arguments the `int[]` array containing the cumulative grade distribution, and the lower and upper bound grades `a` and `b`. The method returns an `int` that is the total number of student scores within the given range, including both end points. You may assume $b \geq a$. Be careful at the limits of the range.

```
public static int numGradeRange (int[] cumulativeGrades, int a, int b){





}
```

Question 4. Streams (20 points)

Your business needs a way to back up its customer records. Given a Vector of `Customer` objects (as defined below) you must write the `Customer` objects to a text file. The `Customer` class is:

```
public class Customer{
     private String customerName;
     private String customerAddress;
     private String customerPhoneNumber;

     public Customer(String custName, String custAddress,
                           String custPhone ){
     customerName = custName;
     customerAddress = custAddress;
     customerPhoneNumber = custPhone;
     }

     public String getCustomerName(){
        return customerName;
     }

     public String getCustomerAddress (){
        return customerAddress;
     }

     public String getCustomerPhoneNumber (){
        return customerPhoneNumber;
     }


} //End of Customer class
```

Part 1: writeToFile. Please complete the `writeToFile` method. The `writeToFile` method creates an appropriate stream and writes the customer data into a file. The output file must have the following format, with each customer's data on a separate line:

customerName1, customerAddress1, customerPhone1
(...)
customerNamen, customerAddressn, customerPhonen

```
public class TestCustomer {
```

```
public static void writeToFile (String fname, Vector v)
      throws IOException {

}
```

Part 2: main. Please complete the `main` method. It must create a Vector of three `Customer` objects (pick their names, addresses and phones yourself) and call the `writeToFile` method. It must handle the `IOException` that might be thrown by `writeToFile` by catching it and outputting some relevant message. Assuming that you are writing to a file called "output.txt".

```
public static void main(String[] args) {

}
```

```
} //end of TestCustomer class
```

Question 5. Threads (15 points)

In this question, you will complete Java classes that can run multiple `Threads`.

Part 1. SimpleThread1

Complete the `SimpleThread1` class, which extends the `Thread` class and overrides its `run()` method. Write the `run()` method so that it prints out `message` on the console **once every second**, a total of five times. The static `sleep()` method of the `Thread` class takes an `int` as an argument and puts the current `Thread` to sleep for that number of milliseconds.

SimpleThread:

```
public class SimpleThread1 extends Thread {

    private String message;

    public SimpleThread1(String message) {
        this.message = message;
    }

    public void run() {
        try {



        }
        catch (InterruptedException e) {
        }
    }
}
```

Part 2. SimpleThreadTest1

Complete the main() method of the `SimpleThreadTest1` class below.
In `main()`, create two instances of the `SimpleThread1` class, the first taking "This is thread 1" as an argument, and the second taking "This is thread 2". Then, invoke the `start()` method on both instances, so that each `SimpleThread1` will print out its own message. Here is the sample output:

This is thread 1
This is thread 2
This is thread 1
This is thread 2
This is thread 1
This is thread 2
This is thread 1
This is thread 2
This is thread 1
This is thread 2

SimpleThreadTest:

```
public class SimpleThreadTest1 {

    public static void main(String[] args) {




    }
}
```

Part 3. SimpleThreadTest1 runtime:

In how many seconds from the start of the program will SimpleThreadTest1 terminate?

____

Part 4. SimpleThread2 and SimpleThreadTest2

In this part, you will complete `SimpleThreadTest2` to test `SimpleThread2`. Unlike `SimpleThread1`, which extends `Thread`, `SimpleThread2` implements the `Runnable` interface. The class `SimpleThread2` is:

SimpleThread2:

```
public class SimpleThread2 implements Runnable {
    private String message;

    public SimpleThread2(String message) {
        this.message = message;
    }

    public void run() {

            // Assume the run() method has the same code
            // as your answer to Part 1

    }
}
```

Your job is to complete the `SimpleThreadTest2` class, which creates and uses two instances of the `SimpleThread2` class. The output from running the main method of `SimpleThreadTest2` must be exactly the same as the output generated by `SimpleThreadTest1` above.

SimpleThreadTest2:

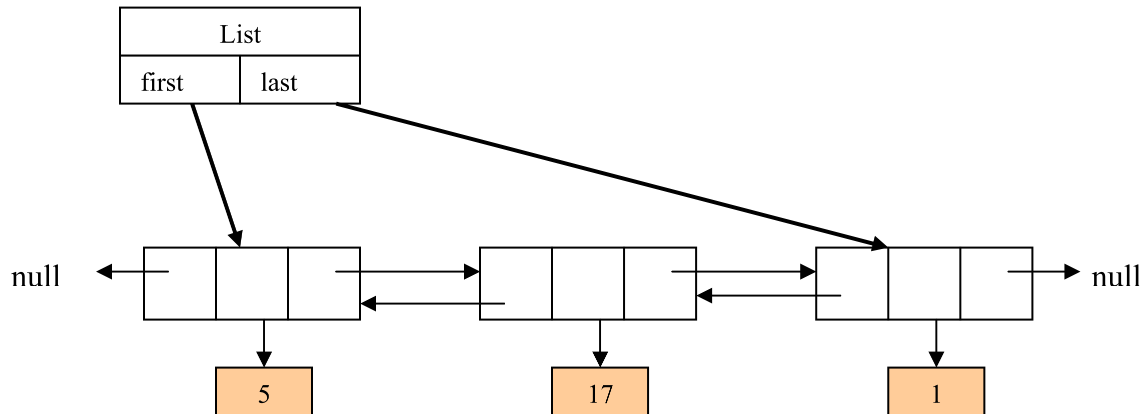
```
public class SimpleThreadTest2 {

    public static void main(String[] args) {
```

```
}}
```

Question 6. Doubly linked list operations (25 points)

In a doubly linked list, every node has a reference (`next`) to the node after it, and another reference (`prev`) to the node before it in the list. As usual, the list itself has `first` and `last` references pointing to the first and last node, respectively. The first node has its `prev` field equal to `null`, and the last node has its `next` field equal to `null`.

Here is what a doubly linked list of 3 Integers looks like:



You are given the following `DoublyLinkedList` class (`Node` is a static inner class).

```
public class DoublyLinkedList {

      public static class Node {

            private Object data;
            private Node prev;
            private Node next;

            public Node(Object o, Node p, Node n) {
                  data = o;
                  prev = p;
                  next = n;
            }

            public Node(Object o) {
                  this(o, null, null);
            }
      }

      private Node first;
      private Node last;

      public DoublyLinkedList() {
            first = last = null;
      }

      private boolean isEmpty() {
            return first == null;
      }
```

```
private boolean hasSingleton() {
      return !isEmpty() && first == last;
}

private boolean inList(Node n) {
      if (n==null || isEmpty())
            return false;

      boolean exists = false;
      for (Node temp = first; temp != null; temp = temp.next)
            if (n.data.equals(temp.data)) {
                  exists = true;
                  break;
            }
      return exists;
}

private boolean isFirst(Node n) {
      return (n!=null && n==first);
}

private boolean isLast(Node n) {
      return (n!=null && n==last);
}

private void removeFirst() {
      if(!isEmpty())
        if (first == last)
          first = last = null;
         else {
           first.next.prev = null;
           first = first.next;
         }
}

private void removeLast() {
      if(!isEmpty())
        if (first == last)
          first = last = null;
         else {
              last.prev.next = null;
              last = last.prev;
         }
}

public void insertAfterLast(Node y) {
      if (y == null) return;
      if (isEmpty())
            first = last = y;
      else {
            last.next = y;
            y.prev = last;
            y.next = null;
            last = y;
      }
}
```

You must write three additional methods for this class as detailed below:

Part 1: insertBeforeFirst

Complete the method `insertBeforeFirst(),` which inserts a new `Node y` at the beginning of the list. ***Hints***: you may want to look at `insertAfterLast()` for inspiration. Assume that `Node y's prev` and `next` may hold arbitrary values when passed to your method; you should explicitly set `prev` and `next` in all cases that you handle in your code. Remember to check that `Node y` is not null.

```
public void insertBeforeFirst(Node y){

}
```

Part 2: insertYafterX

Complete the method `insertYafterX()`, which inserts a new `Node y` after `Node x` in the list. You must handle the following special cases:

- The list is empty.
- `Node x` not in the list.
- `Node x` is the only element in the list.
- `Node x` is the first or last element in the list.

Remember to check that Nodes x and y are not null; you don't need to throw an exception if they are.

```
public void insertYafterX(Node y, Node x) throws NoSuchElementException {

}
```

Part 3: removeY

Complete the method `removeY()` which removes a specific `Node y` from the list. You must handle the following special cases:

- The list is empty.
- `Node y` not in the list
- `Node y` is the only element in the list.
- `Node y` is the first or the last element in the list.

You may find the methods `removeFirst` and `removeLast` useful. Remember to check that `Node y` is not null. You don't need to throw an exception if y is null.

```
public void removeY(Node y) throws NoSuchElementException{









}
```