COM S / ENGRD 211 Computers and Programming
Summer Session 2004

Prelim I

July 12, 2004

Name:_____

CUID:_____ NetID: _____

You have one hour and fifteen minutes to do this exam.

All programs must be written in Java.

Where applicable, show your work so we can award partial credit for wrong answers.

The last page of this exam has a few questions about the course. You can tear this off and fill it out after the exam (which will also preserve anonymity). If you do so, hand it to one of the staff and make sure your bonus points are noted on your exam cover.

Problem	Score	Grader
1. References (10 points)		
2. Induction (10 points)		
3. Recursion (15 points)		
4. DL-Lists (20 points)		
5. SL-Lists (20 points)		
6. Trees (25 points)		
TOTAL (100 points)		
Feedback Form (+ 4 points)		

1. References (10 points)

Consider the code at the right. Now consider it again carefully. What are the final values for variables w, x, y, and z? In other words, what gets printed by main()? (Partial credit can be given for wrong answers if you give short explanations.)

```
public class A {

   static void reset(String s) {
         s = “nil”;
   }

   static String change(String s) {
         s.replace(‘r’, ‘b’);
         return s;
   }

   static String caps(String s) {
         s = s.toUpperCase();
         return s;
   }

   public static void main(String args[]) {
         String w = "ack";
         reset(w);
         String x = "car";
         String y = change(x);
         String z = caps(x);
         System.out.println(w);
         System.out.println(x);
         System.out.println(y);
         System.out.println(z);
   }

}
```

2. Induction (10 points)

Use induction to prove the following result. Your answer must state clearly (i) the base case or cases; (ii) the inductive hypothesis; (iii) the inductive step; and (iv) a conclusion. **For full credit, you must show your work, including algebra.** So show lots of work.

$$1 \cdot 1! + 2 \cdot 2! + 3 \cdot 3! + \ldots + n \cdot n! = (n+1)! - 1 \quad \text{for all } n \geq 1$$

3. Recursion (15 points)

Consider the recursive function *hy(a, n, b)* defined by

$$hy(a,n,b) = \begin{cases} 1 + b & \text{if } n = 0 \\ a & \text{if } n = 1,\ b = 0 \\ a & \text{if } n > 1,\ b = 1 \\ hy(a,\ n\text{-}1,\ hy(a,\ n,\ b\text{-}1)) & \text{otherwise} \end{cases}$$

(a) (10 points) Write a public static method that takes three integers *a*, *n*, and *b* as parameters and computes the value of *hy*(*a*,*n*,*b*), and prints diagnostic output as follows. At the beginning of the method, before any computation is performed, the values of the three parameters are printed. At the end of the method, after all computation is performed and just before it returns, the values of the parameters are printed again along with the return value. For example, calling hy(4, 1, 2) should print

```
invoke hy(4, 1, 2)
     ... other lines deleted ...
return 6 from hy(4, 1, 2)
```

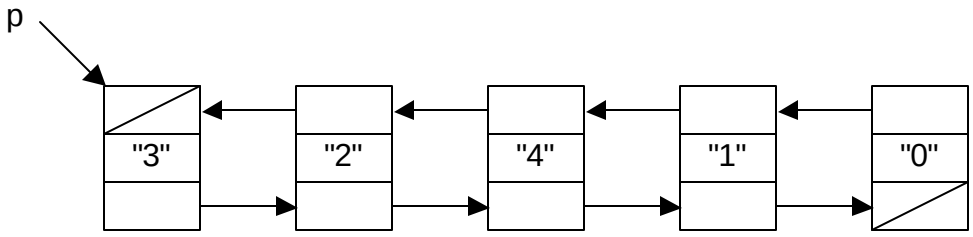
You may assume that all three parameters are non-negative.

(b) (5 points) Show the full output for your method when invoked as hy(4, 1, 2).

This page left empty.

4. Doubly Linked Lists (20 points)

Recall the notion of a *doubly-linked list without header*. One such list is shown below. Each cell is a DListCell instance containing an Object elt, and two DListCell references prev and next.



List containing sequence "3", "2", "4", "1", "0"

Write the DListCell class. All fields in DListCell should be private. You must provide enough functionality so that the main() method below works as indicated, and produces the list in the figure above with variable p referring to the cell on the left. At minimum this involves writing constructors, "getter" routines, and the insert method. Your parameters should match those shown in the code, with the data being an Object. The insert method inserts a single new item after a given item in the list, and has no return value. All your methods may assume that all parameters are non-null, but otherwise they must work for any valid input. E.g., insert must be able to insert a single new cell after the first cell in a list, after the last cell in a list, or after any other cell.

```
public static void main(String args[]) {
      DListCell p = new DListCell("0"); // list p contains only "0"
      for (int i = 1; i <= 3; i++) {
            p = new DListCell(""+i, p); // create new first cell
      }
      // list p now contains  "3",  "2",  "1",  "0"
      DListCell.insert(p.getNext(), new DListCell("4")); // insert "4" after  "2"
      // list p now contains "3",  "2",  "4",  "1",  "0"
}
```

This page left empty.

5. Singly-Linked Lists (20 points)

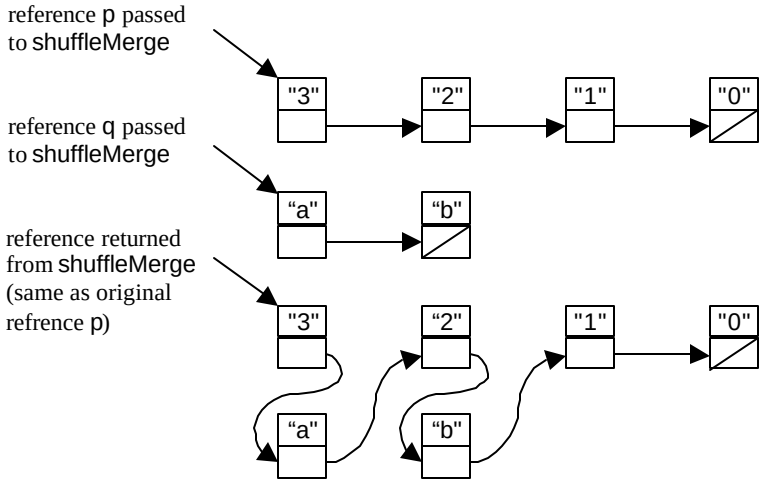
Write a static method shuffleMerge which given two singly-linked lists p and q, merges them together by taking elements alternately from the two lists. First, an element is taken from p, then one from q, then one from p, and so on. If either list is empty (null) or runs out of elements early, the rest of the elements are taken from the other list. An example is shown below, as well as the signatures to the public methods in the ListCell class from lecture. Your method have the following signature:

```
public static ListCell shuffleMerge(ListCell p, ListCell q);
```

and meet these requirements:

- The lists are merged ***in place***. That is, you must modify p and q, rather than allocate new list cells.
- Return a reference to the (possibly-empty) merged list, as well. You will notice that the original reference p is normally returned, or q if p was null.
- Do not use a "header" (the List class discussed in lecture) on the lists.
- **No credit will be given if you allocate new cells** or a new list – that is, **do not allocate any new ListCell objects** (not even temporary ones). You can use as many method-local variables (e.g, temporary references) as you need.
- Do not use any static "class" variables or other helper methods. All of your code must be contained within shuffleMerge.

Hint: Partial credit will be given if you can at least give an outline of an algorithm for solving this problem. It can be solved recursively in as few as 3 or 4 lines of Java code. There are also straightforward iterative solutions.



Shuffle-merging two lists in-place.

```
public class ListCell { // signatures of all public methods and constructors
    public ListCell(Object elt, ListCell next); // new cell with given elt and next
    public Object getElt( ); // get the “data” from this cell
    public ListCell getNext(); // get the next cell after this one
    public void setElt(Object elt); // set the “data” for this cell
    public void setNext(ListCell next); // set the next pointer to a different cell
}
```

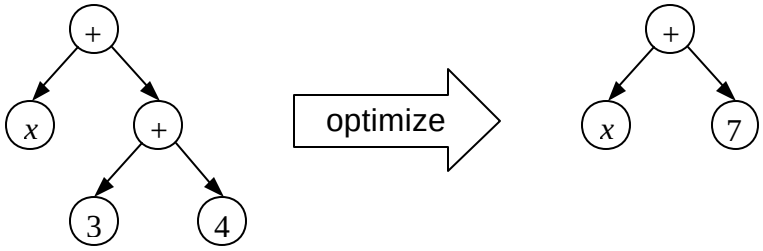
This page left empty.

6. Trees (25 points)

Here is a modified version of the grammar for simple arithmetic expressions given on the homework:

Expression : (Expression + Expression)
Expression : integer
Expression : variable // a character in the range 'a' – 'z' or 'A' – 'Z'

We can represent any expression in this grammar by an *abstract syntax tree (AST)*. For example, the expression *(x + (3 + 4))* is represented by the binary tree below (at left). In this example, the sub-expression *(3 + 4)* is constant, and so we might like to optimize the AST by replacing the right sub-tree by a single node.



Write a static method optimize which, when given an AST, produces an optimized AST with no constant sub-expressions. The optimized and original AST should not share any storage in common, with the possible exception of (immutable) String objects. That is, you should construct an entirely new tree in your method, and should not modify the original tree in any way. You can also make new copies of the String objects, if you like.

The *AST* is implemented using the TreeCell class, but modified to hold Strings instead of Objects. Each variable, integer, and operation will be stored in the TreeCell as a String instance. You may assume that the original AST is not null, and that each of its TreeCells contain a valid string: either "+", a single letter, or an integer. The public methods signatures for the TreeCell class are given below, with other helpful methods from Character and Integer. You are only responsible for writing the optimize method.

Hints: An expression with no sub-expressions, or one that is empty, by definition has no constant sub-expressions. So in these cases there is no real work to be done. One way to optimize an expression that does have sub-expressions is as follows: first optimize left and sub-expressions; then, examine the root node and its left child and right child to see if the whole tree can be optimized away.

```
public class TreeCell { // signatures of all public methods and constructors
    // construct TreeCell with no children
    public TreeCell(String elt);
    // construct TreeCell with given children
    public TreeCell(String elt, TreeCell left, TreeCell right);
    public String getElt(); // get element at this node
    public TreeCell getLeft(); // get left child
    public TreeCell getRight(); // get right child
}
```

```
public class Character { // signatures of some helpful methods
    static boolean isLetter(char c); // determine if c is a letter
    static boolean isDigit(char c); // determine if c is a digit
}
```

```
public class Integer { // signatures of some helpful methods
    static int parseInt(String s); // parse s as an signed decimal integer
}
```

This page left empty.

Course Feedback
(1 point per question answered)

You may tear off this sheet and give it to the course staff to maintain anonymity. You can also take as much extra time as you need to fill this out.

1. How many hours per week are you spending on the assignments for this course?

2. Do you attend lectures regularly? What can we do to improve the lectures?

3. Do you see the consultants or TAs for help? What can we do to improve consulting?

4. Any other comments? (Assignments, prelim, etc.)