UNIVERSITY OF TORONTO

Faculty of Arts and Science

APRIL-MAY 2006 EXAMINATIONS

CSC 148H1 S and A48H3 S

Duration — 3 hours

No aids allowed

ANSWERS

Answer all questions in the space provided in this paper.

Check that this test paper has 17 pages, including this cover page.

Comments are not required except where indicated, though they may help us to mark your answers.

You need not throw or catch exceptions, unless explicitly asked to.

Efficiency and style are not of the highest importance, but may count for marks in the case of major difficulties.

Helper methods are allowed unless specifically prohibited.

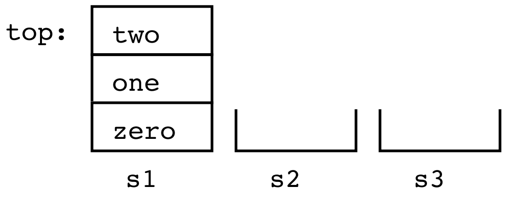
1.	__________	/24
2.	__________	/10
3.	__________	/15
4.	__________	/5
5.	__________	/10
6.	__________	/10
7.	__________	/16
8.	__________	/10
Total	__________	**/100**

continued …

1. [24 marks = 8 × 3]

Each of these short questions is worth 3 marks, and they are separate, independent questions unless otherwise stated.

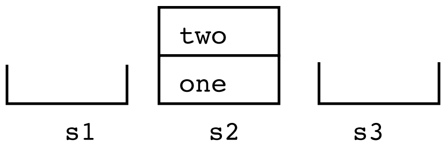
(a) We have three stacks, s1, s2 and s3, that can contain data of type `String`. Here are their initial contents:



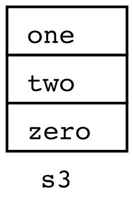
As you can see, initially s2 and s3 are empty. Here is a sequence of operations on the three stacks:

```
s2.push(s1.pop());
s3.push(s1.pop());
s1.pop();
s1.push(s2.pop());
s2.push(s3.pop());
s2.push(s1.pop());
```

In the space below, draw the contents of the three stacks after the operations are complete.

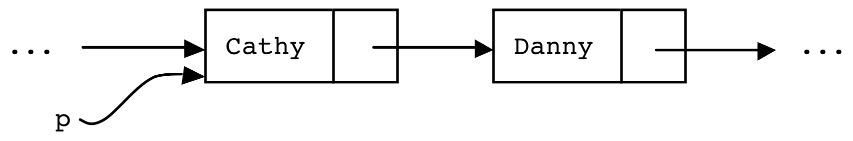


(b) This question is about the same three stacks, `s1`, `s2` and `s3`, as in part (b). We add the rule that when you pop an element from a stack, it can only be pushed onto a higher-level stack. For example, "`s1.push(s3.pop());`" would not be allowed. Suppose we start from the same initial state as in part (b) and eventually move all the strings to `s3`. Give an example of a permutation (ordering) of the elements "`zero`", "`one`" and "`two`" that we *cannot* attain on `s3`.



1. (continued)

(c) Consider this situation in the middle of a linked list:



Suppose that in a Java program you have a variable `p` that points to the list node containing "`Cathy`". Suppose also that nodes belong to this class:

```
public class Node {
  private String data;
  private Node next;
  public Node(String d, Node n) { data = d; next = n; }
}
```

Write one Java command, without any new variable names, that will store "Jim" in the list, after "Cathy" and before "Danny".

```
p.next = new Node("Jim", p.next); // We assume Node is an inner class.
```

(d) Suppose that you were given the specification of a Java method last week, and though you have given it some thought, you have not begun implementing it. Then yesterday you were given a precondition to add to the specification.

Does the precondition make your job easier or harder? Explain why or why not.

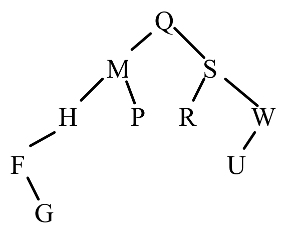
(Answer in one to three sentences.)

```
It makes it easier, because the precondition limits the range of parameters
that we have to handle correctly.
```

continued …

1. (continued)

(e) Here is a binary tree:



Give the results of performing traversals of this tree, printing each node as you visit it:

i. in-order traversal:

```
F G H M P Q R S U W
```

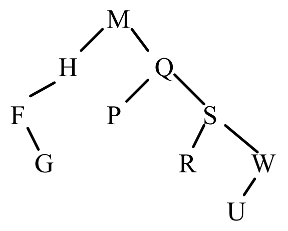
ii. pre-order traversal:

```
Q M H F G P S R W U
```

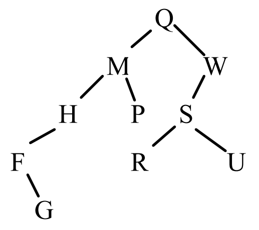
iii. post-order traversal:

```
G F H P M R U W S Q
```

(f) Working with the tree in par t(e), apply a right rotation at node Q, so that M becomes the new root. Draw the resulting tree.



(g) Again working with the tree in part(e) — without the change made in part (f) — apply a left rotation at node S. Draw the resulting tree.



continued …

1. (continued)

(h) Recall the `LowerTriangular` class in Assignment 2, where a triangular two-dimensional array was used to store matrix values. Suppose we instead store all the data in a one-dimensional array, with the rows laid out "end to end". For example, this matrix:

$$\begin{pmatrix} 11 & 0 & 0 \\ 21 & 22 & 0 \\ 31 & 32 & 33 \end{pmatrix}$$

would be stored as this array: `{ 11, 21, 22, 31, 32, 33 }`. Notice that the matrix elements that must be 0 are omitted.

Here is the original `get0()` method that returns the matrix elements at row *i* and column *j*:

```
public int get0(int i, int j) {
  if (j <= i) {
    return data[i][j];    // **
  }
  else {
    return 0;
  }
}
```

Fix the line marked "`**`" so that it works with a one-dimensional storage array as outlined above. The array is still named "`data`", but it now has only one index.

Hint: The sum of the first *n* positive integers is 1 + 2 + 3 + … + *n* = *n*(*n* + 1) / 2.

```
return data[i*(i + 1)/2 + j];  // (i – 1)*i: -1 mark
```

continued …

2. [10 marks]

In high school, you probably learned how to solve systems of linear equations. Here is an example:

$$\begin{aligned} 2x_1 + 5x_2 + 3x_3 &= 31 \\ 6x_2 - x_3 &= 16 \\ x_3 &= 2 \end{aligned}$$

The example is so easy you can probably do it in your head: $x_1 = 5$, $x_2 = 3$, and $x_3 = 2$. Surprising as it may seem, that's the expected form for a linear system after some standard preprocessing: the last equation has only one variable in it, the second last has two, and so on until all n variables are present in the first equation. The array A giving the coefficients of the left-hand side is therefore *upper triangular*: it is $n \times n$, but A_{ij} is 0 whenever $j < i$. Then the solution works backwards from the last equation. In our example, we find x_3 from the last equation, plug that into the second equation to get x_2, and finally put both those values into the first equation to get x_1.

In "real life", you have to worry about special circumstances; for example, what do you do when an important coefficient is 0? Here, we will assume that the problem is nicely behaved, so you can ignore worries about dividing by 0.

Complete the method begun below:

```
/** Return the solutions to the system of linear equations represented
 * by coeffs (the coefficients on the left-hand side) and rhs (the constants
 * on the right-hand side).
 * Precondition: If n is the number of rows in coeffs, then:
 * 1) the rows of coeffs are of length n, n-1, n-2, ..., 2, 1.
 * 2) the length of rhs is n.
 * @return The solution to the system, an array of length n.
 */
public static double[] solve(double[][] coeffs, double[] rhs) {



  int n = coeffs.length;
  double[] soln = new double[n];

  for (int row = n – 1; row >= 0; row--) {
    soln[row] = rhs[row];
    int rowlen = coeffs[row].length; // rowlen = n – row
    for (int vbl = row + 1; vbl < n; vbl++) {
      int vblpos = vbl – row; // where the coefficient of the vbl is
      soln[row] -= coeffs[row][vblpos]*soln[vbl];
    }
    soln[row] = soln[row] / data[row][0];
    // It’s OK to use rhs[] itself to store the answer, though that seems
    // a little unpleasant.
  }
  return soln;
}
```

continued …

2. (continued)

(You may use this space, though you probably will not need it.)

continued …

3. [15 marks = 7 + 8]

Consider a binary tree consisting of nodes belonging to this Java class:

```
public class Node {
  public Object data;
  public Node left;
  public Node right;
}
```

In this question, the tree is a binary tree. It does not have to be a binary *search* tree.

(a) Write a method `nodeCount()` that returns an `int[]` — an array of `ints` — of length three, containing the number of nodes, leaves and internal nodes of the tree rooted at `root`.

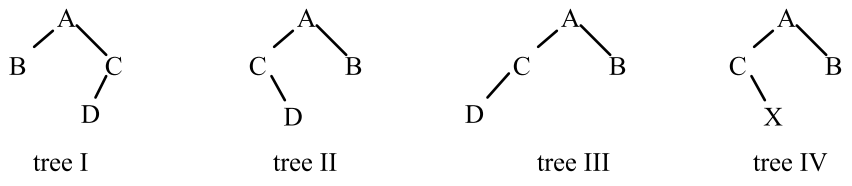
The method is begun for you:

```
public static int[] nodeCount(Node root) {



  if (root == null)
    return new int[3]; // all 0

  int[] result = new int[3];
  if (root.left == null && root.right == null) {
    // It's a leaf.
    result[1] = 1;
  }
  else {
    int[] lCount = nodeCount(root.left);
    int[] rCount = nodeCount(root.right);
    result[1] = lCount[1] + rCount[1];
    result[2] = lCount[2] + rCount[2] + 1; // This node is internal too.
  }
  result[0] = result[1] + result[2];
  return result;
}
```

continued …

3. (continued)

(b) Using the same tree structure as in part (a), write a method `mirror(tree1, tree2)` that returns `true` if `tree1` is the mirror image of `tree2`. For the purposes of this question, one tree is "the same as" another if they both contain exactly the same data values and have exactly the same branching structure. A "mirror image" of a tree is another tree that would be "the same" except that it is reflected in an imaginary vertical mirror. In other words, although the values are the same, the structure of the tree is exchanged between left and right.



In the example, trees I and II are mirror images of each other, but trees III and IV are *not* mirror images of tree I or of tree II, because tree III has the wrong structure and tree IV contains an "X" where there should be a "D".

To compare equality of data values, use the `equals()` method.

For part marks, your method may check whether the two trees are "the same" rather than "mirror images". To get the part marks, you must say that's what you're doing.

```
/** Return true if the tree rooted at root2 is the mirror image of the tree
 * rooted at root1; otherwise return false.
 */
public static boolean mirror(Node root1, Node root2) {



  if (root1 == null || root2 == null)
    return root1 == root2;

  return root1.data.equals(root2.data)
      && mirror(root1.left, root2.right)
      && mirror(root1.right, root2.left);
}
```

continued …

4. [5 marks]

Write an iterator that returns powers of 2 starting at 1 — that is, 2^0. The value returned will actually be an `Integer`, not an `int`, so the first call of `next()` will return 1, the second call will return 2, the third 4, and so on.

To get you started, we have provided a skeleton of the desired class, including all the required methods. You are free to add other methods and any instance or other variables you wish.

Ignore the fact that integers in a real computer have an upper limit.

```
import java.util.*;

public class PowersOfTwo implements Iterator<Integer> {
  // Your instance variables go here.


  private int next = 1;







  public void remove() {
    throw new UnsupportedOperationException();
    // Don't bother to change this method.
  }
  public Integer next() {


    Integer result = new Integer(next); // or just = next
    next *= 2;
    return result;
  }














  public boolean hasNext() {


    return true;
  }
}
```

continued …

5. [10 marks]

A *palindrome* is a word or sequence of words that is the same frontward and backward. Some examples are “rotor”, “noon” and “madam”. Phrases of many words can also be considered palindromes, but then we have to consider what to do with spaces and punctuation, which we prefer to ignore in this question.

Here we assume that the word has been taken apart into characters, which have been pushed onto a stack, with the first character at the bottom and the last at the top.

Complete the method below, with these restrictions:

- You may not declare any variables — not instance variables, not static variables, and not even local variables inside the method.
- You may not use any helper methods.
- You may, however, change the contents of the parameters `s1` and `s2`.

```
/** Return true if the characters on s1 form a palindrome. The data type of
 * elements stored in s1 and s2 is “char”.
 *
 * s1 and s2 may be altered by this method.
 *
 * @param s1 – A stack of characters, with the usual push(), pop() and
 *    isEmpty() methods, and also an extra method size() that returns the
 *    number of items currently on the stack.
 * @param s2 – An empty stack for characters, of the same type as s1.
 * @return true if the characters on s1 are a palindrome, otherwise false.
 */
public static boolean isPalindrome(Stack<char> s1, Stack<char> s2) {
  // Remember: no variables, but Stack has a size() method.




  while (s1.size() > s2.size())
    s2.push(s1.pop());
  if (s1.size() != s2.size())
    s2.pop();
  while (! s1.isEmpty())
    if (s1.pop != s2.pop())
      return false;
  return true;
}
```

continued …

6. [10 marks]

Here is an interface specifying a double-ended queue, or deque:

```
public interface Deque {
  /** Add o at the front of this Deque. */
  void addFront(Object o);
  /** Add o at the back of this Deque. */
  void addBack(Object o);
  /** Remove an item from the front of this Deque and return it. */
  Object removeFront();
  /** Remove an item from the back of this Deque and return it. */
  Object removeBack();
  /** Return the number of items in this Deque. */
  int size();
}
```

Notice that we are not using a generic type parameter; the values stored are `Objects`.

As an example, if `dq` is a `Deque` and we call the methods `dq.addFront("Alice")`, `dq.addBack("Bob")`, `dq.addFront("Carol")` then the contents of `dq` are, from front to back: `"Carol"`, `"Alice"`, `"Bob"`. If we then call the method `dq.removeFront()`, the contents of `dq` become: `"Alice"`, `"Bob"`.

List five test cases you might use to look for *different* errors in an implementation of `Deque`.

- You do not need to write `JUnit` code for your tests. Brief verbal descriptions are enough.
- If you write more than five tests, the *first* five will be marked.

```
Some tests that would be OK:
•  create new Deque, check size = 0
•  create, addFront(), check size, removeFront(), check equal, check empty
•  create, addFront(), addBack(), check size, removeFront(), check equal,
   check size
•  create, addBack(), check size, removeBack(), check equal, check empty
•  create, addBack(), addFront(), check size, removeBack(), check equal, check
   size

... and more ...
```

continued …

6. (continued)

(You may use this space, though you probably will not need it.)

continued …

7. [16 marks = 8 + 2 + 2 + 4]

```
public class Q {

  public static void main(String[] args) {

    String[] exp1 = {"3", "4", "5", "+", "*"};
    String[] exp2 = {"1", "2", "3", "4", "5", "*", "+", "6", "*",
        "*", "+"};
    String[] exp3 = {"10", "5", "/", "2", "/"};
    String[] exp4 = {"50", "10", "-", "3", "10", "-", "-", "2", "+"};

    System.out.println(eval(exp1));
    System.out.println(eval(exp2));
    System.out.println(eval(exp3));
    System.out.println(eval(exp4));
  }

  public static int eval(String[] exp) {
    IntStack s = new IntStack();

    for (int i = 0; i < exp.length; i++) {

      if (exp[i].equals("+")) {      s.push(s.pop() + s.pop());   }

      else if (exp[i].equals("-")) { int t = s.pop();                // **
                                      s.push(s.pop() - t);           // **
                                    }

      else if (exp[i].equals("*")) { s.push(s.pop() * s.pop());   }

      else if (exp[i].equals("/")) { int t = s.pop();
                                      s.push(s.pop() / t);         }

      else {                          s.push(Integer.parseInt(exp[i]));  }
    }
    return s.pop();
  }
}

class IntStack {

  private Node top;

  private class Node {
    private int val;
    private Node next;
    private Node(int v, Node n) { val = v; next = n; }
  }

  public void push(int v) {
    if (top == null) {   top = new Node(v, null);    }
    else {               top = new Node(v, top);     }
  }

  public int pop() {
    int t = top.val;
    top = top.next;
    return t;
  }
}
```

continued …

7. (continued)

On the opposite page is a Java program that compiles and runs without error.

(a) What is the output from the program?

```
27
277
1
49
```

(b) Look at the second branch of the big "`if`" statement — the one where the lines in the body are marked "`**`". Suppose we replace those lines with this single line:

```
s.push(s.pop() – s.pop());
```

(The condition in the surrounding "`else`" clause is not changed.) How does the output change?

```
No change.
```

(c) What is the running time (the big-O cost) of the methods `push()` and `pop()` in the `Stack` class? Justify your answer.

```
They are both O(1), because no repetition is needed.
```

(d) What is the running time of the method `eval()` in the class `Q`? Justify your answer.

(Hint: you may define *N* to be the length of the method's parameter, "`exp`".)

```
In the body of the for loop, each branch of the if statement has cost O(1),
because there is no repetition.
Thus, the if statement costs O(1), so the loop body costs O(1).
The loop is executed N times, so the loop costs N*O(1) = O(N).
The declaration of s costs O(1), because creation of an IntStack has no repe-
tition.
The return statement costs O(1).
The total cost is O(N) + O(1) + O(1) = O(N).
```

continued …

8. [10 marks = 2 + 4 + 4]

Here is a Java program that compiles without error messages or warnings. Some method bodies are omitted.

```
import java.io.*;
import java.util.Iterator;
import java.util.List;
import java.util.ArrayList;

public class Run {
  private static class Node {
    private int data;
  }
  private static BufferedReader br; // used in code that we omitted

  public static void main(String[] args) {
    init();
    List<Node> lis = new ArrayList<Node>();

    System.out.println("Reading ...");
    Node inNode = read();
    while (inNode != null) {
      System.out.println(inNode.data);
      lis.add(inNode);
      inNode = read();
    }

    System.out.println("Printing 1 ...");
    for (Iterator<Node> it = lis.iterator(); it.hasNext(); ) {
      System.out.println(it.next().data);
    }

    System.out.println("Printing 2 ...");
    print(lis.iterator());

    System.out.println("Printing 3 ...");
    for (Iterator<Node> it = lis.iterator(); it.hasNext(); ) {
      System.out.println(it.next().data);
    }
  }

  private static void init() {
    ...
  }

  private static Node read() {
    ...
  }

  private static void print(Iterator<Node> iter) {
    ...
  }
}
```

continued …

8. (continued)

When the program is run with certain input, there are no errors or exceptions, and this is the output that appears:

```
Reading ...
34
45
56
Printing 1 ...
34
45
56
Printing 2 ...
34
45
56
Printing 3 ...
56
45
34
```

Here, we are going to use the term "surprising" to describe something that the programmer would find undesirable or at least unexpected. In the code itself, "surprising" features are things that might be bugs or errors and that should be removed or fixed.

(a) What is surprising about the output?

`The order of the output is different after "Printing 3"!`

(b) The surprising feature of the output must be caused by something surprising in the program. Which method must be the one that contains the surprising code? How do you know that?

You may assume that the surprising code does not involve exceptions, the `remove()` method of iterators, or type parameters (the "generics" that are new in Java 5).

```
It must be print(). The code between "Printing 1" and "Printing 2" has no
calls to methods defined here, and similarly the code after "Printing 3" uses
only standard API methods. Also, the output after "Printing 2" is OK. The only
possibly suspect method called in this area of the program is print().
```

(c) What kind of surprising code should the programmer look for when fixing the program? That is, what kind of wrong, buggy or malicious code causes the surprising output?

```
The only thing print() is given is an iterator, and we're told the iterator
doesn't isn't used directly to change the list (because its remove() method is
not involved). So all print() can work with is the Nodes returned by the it-
erator. It can't change references to the Nodes, or the list would be broken
after "Printing 3" (and anyway it doesn't have access to the list itself), so
it must change the contents of the Nodes. Presumably it sneakily saves refer-
ences to some Nodes and interchanges their contents with the contents of other
Nodes. (It could also have values from 34, 56, 45 built in, but that's very
unlikely and could be checked with further tests.)
```