University of Toronto
CSC148h1s—Introduction to Computer Science, Spring 2001—Section L5101

Midterm Test Solutions

March 1, 2001

Duration: 50 minutes

Aids allowed: None

Weight: 15% of your course grade

This exam contains a total of 8 pages (including this one). Write your answers clearly in the spaces provided. Use the back pages for your rough work.

Family name: _____

First name: _____

Student #: _____

Tutor (circle one):

Amy Duong SS1070	Adam Foster MP137
Ryan McDonald LM148	Elena Gardinerman BN324

0: _____/ 2

1: _____/12

2: _____/ 8

3: _____/10

4: _____/ 8

TOTAL: _____/40

Good Luck!

Question 0. [2 MARKS]

Write your name (or your initials if your name is long) and student number legibly at the top of every page of this test.

Question 1. True or False [12 MARKS]

The following questions are about Java. For each statement, indicate whether it is true of false. [1 mark for each correct answer, -1 for each incorrect answer, 0 for each answer left blank]

1. A class which implements an interface but does not supply all the methods in the interface must be declared abstract. T
2. The member variables used to implement an interface should be declared public. F
3. The memory model is a useful tool to determine if a program will compile correctly. F
4. Suppose class A extends class B. Furthermore, suppose class B has a method `m()`, which class A overrides. The the only way to access method `B.m()` from an object of class A is to use the `super` keyword from a method within class A. T
5. The external documentation for a class should explain what each member variable is used for. F
6. Representation invariants are designed to help a programmer using your class understand how to use it appropriately. F
7. All Exceptions must be caught. F
8. Writing a sort method that accepts an array of Comparables as input is a good idea because it can then be used to sort (almost) any kind of data. T
9. Abstract classes are useful because they allow the programmer to indicate that every object that is an instance of a given class has a particular method, even if he or she can't provide the body of that method. T
10. All the nodes of a linked list must be stored contiguously in memory. F
11. The height of a tree with at least two nodes is always equal to one plus the height of its shortest subtree. F
12. Search time in a binary search tree is proportional to the number of leaves in the tree. F

SOLUTION: Displayed next to each question.

MARKING SCHEME: See question statement.

COMMENTS: When you see a question like this, make an effort to decide which questions you actually know, and which ones you're really not sure about. In most cases, you should have been either able to answer the question or know that you didn't know and leave it blank.

NOTE: Some copies has the last six questions first, so don't be surprised if the order of the questions on your copy was different than the order above.

Question 2. Linked List Manipulation [8 MARKS]

Consider the following `CharNode` class, which can be used to implement a linked list of characters.

```
class CharNode {
    char value;
    CharNode link;
}
```

Complete the `removeRepetitions()` method below according to its specification by filling in the blanks. Each blank may only be filled with one of `curr`, `front`, `link`, `next`, `null` or `value`.

```
// removeRepetitions: remove all successive nodes with the same value from
// the linked list of characters referred to by front, where two nodes x
// and y are considered to have the same value if (x.value == y.value).
// For example:
// 1. If front refers to the list (M,I,S,S,I,S,S,I,P,P,I), then after
//    calling removeRepetitions(front), the list will become (M,I,S,I,S,I,P,I).
// 2. If front refers to the list (A,A,B,C,A,A,A,D,E,E,E,E,E), then after
//    calling removeRepetitions(front), the list will become (A,B,C,A,D,E).
```

SOLUTION:

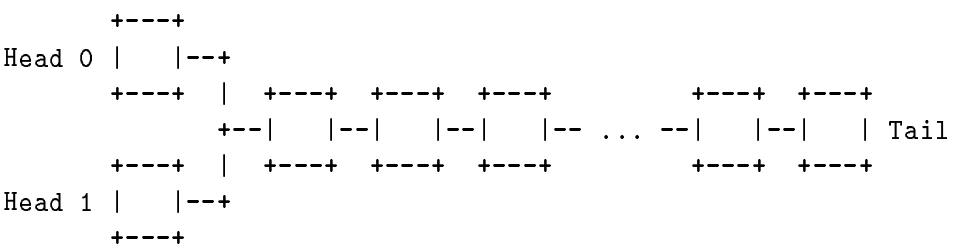
```
public static void removeRepetitions ( CharNode front ) {
    CharNode curr = front;
    while ( curr != null ) {
        CharNode next = curr.link;
        while ( next != null &&
                curr.value == next.value ) {
            next = next.link;
        }
        curr.link = next;
        curr = curr.link // This line was missing on the question
    }
}
```

MARKING SCHEME: -1 per mistake. However, since there was a mistake in the question, if it seemed like some errors you made were due to the error in the question, some penalties were not applied.

COMMENTS: This question was generally very well done.

Question 3. ADTs [10 MARKS]

A two-headed queue is an ADT which contains zero or more objects. It is like a regular queue in that objects enter the queue at the tail. It is unlike a regular queue in that it has two heads (instead of one), called head 0 and head 1, where objects can leave the queue. See diagram.



A two-headed queue can be used to simulate a lineup of customers at a bank where there are two tellers. The two objects at the two heads represent the two customers being served by the two tellers, and the objects in the rest of the queue represent the customers waiting to be served on a first come first served basis.
The operations which can be perforned on a two-headed queue are:

Size: returns the number of objects in the two-headed queue.

Enqueue: given an `Object`, put it into the two-headed queue. If one of the heads is empty, put the object at that head. If both heads are empty, put the object at head 0. If neither is empty, inseert it at the tail.

Dequeue: given `whichHead`, where `whichHead` is 0 or 1, remove object at head `whichHead` from the two-headed queue and move the first object from the rest of the two-headed queue, if there is one, to head `whichHead`. Also, return the object that was removed.

Suppose you are given a class `LinkedQueue` which implements the following (regular) `Queue` interface:

```
interface Queue {
    // size: returns how many objects are in me.
    public int size();

    // enqueue: put o at my tail.
    public void enqueue(Object o);

    // dequeue: remove object at my head and return it.
    //          return null if I'm empty.
    public Object dequeue();
}
```

The following class implements a two-headed queue, using a `LinkedQueue` and an array of two `Objects` to store the elements of the two-headed queue. The constructor, `size()` and `dequeue()` methods have been provided for you.

```
class TwoHeadedQueue {
    private Object[] heads;     // my heads
    private Queue rest;         // rest of my objects

    public TwoHeadedQueue() {
        heads = new Object[2];
        rest = new LinkedQueue();
    }

    public int size() {
        int numObjs = 0;
        if (heads[0] != null) numObjs++;
        if (heads[1] != null) numObjs++;
        return numObjs + rest.size();
    }

    public void enqueue(Object o) {
        // Body omitted (to be filled in below)
    }

    public Object dequeue(int whichHead) {
        Object retObj = heads[whichHead];
        heads[whichHead] = rest.dequeue();
        return retObj;
    }
}
```

Part (a) [4 MARKS]

Write representation invariants for this class.

SOLUTION:

```
    // - heads[0] points to head 0, or null if there is no head 0
    // - heads[1] points to head 1, or null if there is no head 1
    // - rest is empty (i.e., rest.size() == 0) if either head is null.
    // - The size of the two-headed queue is the number of non-null heads
    //   plus rest.size()
```

MARKING SCHEME: I tried to generally assess how close your invariants were to the above, so it's hard to describe this marking scheme exactly, although the following two penalties were applied:

- -1 if the meaning of either head being `null` was not explained.
- -2 if you didn't say that having either head null implied that rest was empty.

COMMENTS: This question was very poorly done. Representation invariants describe a snap shot of

the data structure, explaining the meaning of the different values the member variables might have, and what relationships are guaranteed to hold between them. Many students described, instead, what the methods of the class do, and what the parameters to each methods mean. These things belong with method documentation, not with representation invariants.

Part (b) [4 MARKS]

Complete the body of the `enqueue` method of class `TwoHeadedQueue`.

```
public void enqueue(Object o) {
```

SOLUTION:

```
    if (heads[0] == null)
        heads[0] = o;
    else if (heads[1] == null)
        heads[1] = o;
    else
        rest.enqueue(o);

}
```

MARKING SCHEME:

- 2 points for the if structure (i.e., for correctly detecting the different cases),
- 1 point for the first two cases (i.e., `heads[0/1] = o`),
- 1 point for the else case (i.e., `rest.enqueue(o)`).

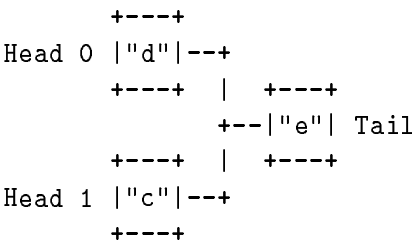
COMMENTS: This was generally well done, although many students included a fourth redundant case, testing separately for having both heads null or only the first head null. This extra case was not useful, but no marks were taken off for having it, since it isn't incorrect.

Part (c) [2 MARKS]

Using the notation on the first page of this question, draw the representation of the two-headed queue resulting from the following sequence of calls. (Note: in each box, you should also write the value of the object stored there.)

```
TwoHeadedQueue q = new TwoHeadedQueue();
q.enqueue("a");
q.enqueue("b");
q.enqueue("c");
q.dequeue(1)
q.dequeue(0)
q.enqueue("d");
q.enqueue("e");
```

SOLUTION:



MARKING SCHEME:

- 1 point for the structure,
- 1 point for the values.

COMMENTS: This question was generally well done, although many students didn't notice that after the two calls to `dequeue()`, head 0 would be left null, not head 1.

Question 4. Recursive Programming [8 MARKS]

The class `TreeNode` below implements a binary tree. A binary tree consists of a `Comparable` value, a (possibly empty) left substree, and a (possibly empty) right subtree.

Within this class, we've written a mystery method with its helper method. Your task is to analyze these methods and understand what they do.

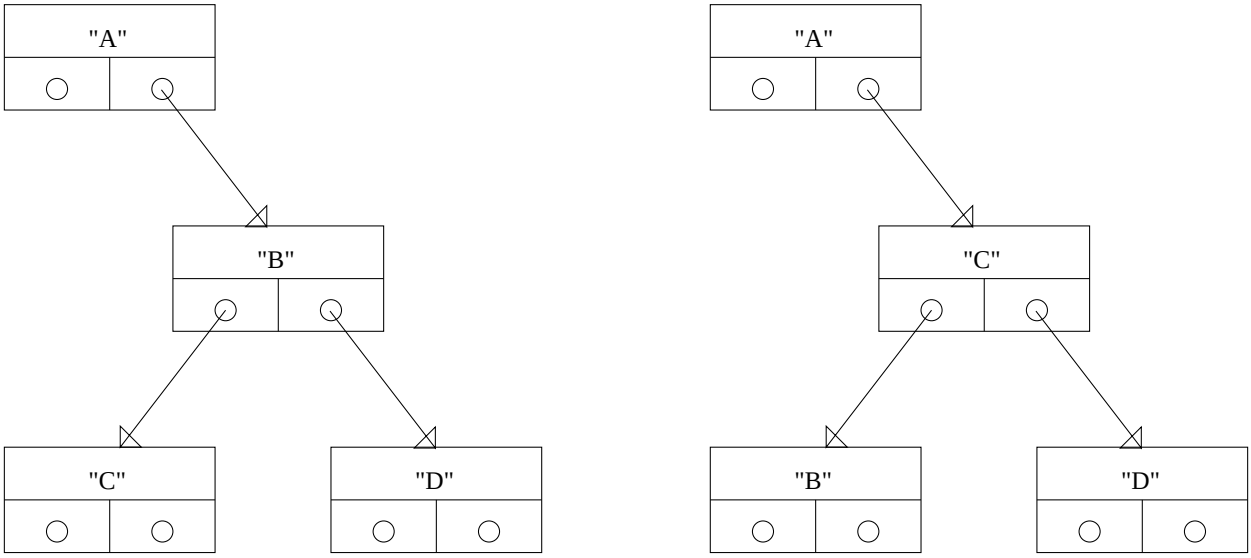
```
public class TreeNode {
    Comparable value;  // my value
    TreeNode left;     // my left child (or null if I have no left child)
    TreeNode right;    // my right child (or null if I have no right child)

    public static boolean mystery(TreeNode t) {
        return mysteryHelper(root, null, null);
    }

    private static boolean mysteryHelper(TreeNode t, Comparable a, Comparable b) {
        if ( t == null ) return true;
        else if ( a != null && t.value.compareTo(a) <= 0 ) return false;
        else if ( b != null && t.value.compareTo(b) >= 0 ) return false;
        else return mysteryHelper(t.left, a, t.value) &&
                    mysteryHelper(t.right, t.value, b);
    }
}
```

Part (a) [2 MARKS]

For the two trees drawn below, trace a call to `mystery` with the root of the tree as its argument, and write what the call returns. (Note: your answer only needs to say what each call returns.)



SOLUTION: `false` for the first tree, `true` for the second tree.

COMMENTS: If you didn't figure out what the method did, you should have at least recognized that it returns a `boolean` and made a guess.

Part (b) [6 MARKS]

Describe, in general, what `mystery()` and `mysteryHelper()` do. Your description should specify the purpose of every parameter and then say what the method returns. (Note: you don't need all the space available to answer this question.)

SOLUTION:
`mystery(t)` returns `true` if `t` is the root of a properly-formed Binary Search Tree, `false` otherwise. `mystery(t,a,b)` returns `true` if `t` is the root of a properly-formed Binary Search Tree whose nodes are all greater than `a` if `a` is not `null`, and whose nodes are all smaller than `b` if `b` is nut `null`; `false` otherwise.

MARKING SCHEME:

- 2 points for saying that `mystery(t)` tests if the tree is a BST,
- 1 point for saying that `t` is the root of the tree,
- 1 point for saying that `mystery(t,a,b)` tests if the tree is a BST,
- 1 point for saying that the values of the nodes of the tree must be between `a` and `b`,
- 1 point for saying what it means for `a` and `b` to be `null`.

COMMENTS: This was a difficult question, but you had sufficient tools to solve it, especially since we talked about BSTs in lecture just before the midterm. Common mistakes include:

- Not describing what the parameters did.
- Writing far too verbose solutions—a few lines were sufficient!
- Rewriting the code in English instead of summarizing what it does. When describing a recursive method (or any method for that matter), you should explain what it does overall, not describe its code line by line.

Total Marks = 40