CSC 148H1 Summer 2009 Midterm Test
Duration — 80 minutes
Aids allowed: none

Student Number: |_|_|_|_|_|_|_|_|_|

Lab day: ____________________

Last Name: ____________________ **First Name:** ____________________

Lecture Section: L0101 **Instructor:** R. Danek

---

*Do **not** turn this page until you have received the signal to start.*
(Please fill out the identification section above, **write your name on the back of the test**, and read the instructions below.)
Good Luck!

This test consists of 6 questions on 10 pages (including this one). *When you receive the signal to start, please make sure that your copy is complete.*
Comments are not required except where indicated, although they may help us mark your answers. They may also get you part marks if you can't figure out how to write the code.
If you use any space for rough work, indicate clearly what you want marked.

# 1:	______/ 6
# 2:	______/ 8
# 3:	______/10
# 4:	______/ 8
# 5:	______/10
# 6:	______/12
TOTAL:	______/54

Question 1. [6 MARKS]

Suppose you are given a class Foo, with the following method definitions:

```
class Foo:
    def __init__(self):
       ''' Initialize an instance of Foo. '''
       # implementation omitted

    def foo_add(self, val):
       ''' Add the value given by val to the Foo instance. '''
       # implementation omitted

    def foo_remove(self):
       ''' Remove and return the next item from the Foo instance. '''
       # implementation omitted

    def size(self):
       ''' Return the number of elements in the Foo instance. '''
       # implementation omitted

    def is_empty(self):
       ''' Return True if the Foo instance is empty, False otherwise. '''
       # implementation omitted
```

You are told that Foo is either: (i) an implementation of the Stack ADT, where `foo_add` and `foo_remove` correspond to `push` and `pop` operations, respectively; or (ii) an implementation of the Queue ADT, where `foo_add` and `foo_remove` correspond to `enqueue` and `dequeue`, respectively.

Consider the following code snippet:

```
f = Foo()
f.foo_add(1)
f.foo_add(2)
f.foo_add(1)

while not foo.is_empty():
   print f.foo_remove()
```

Is it possible to determine whether Foo is a Stack or Queue by inspecting the values printed to the screen? If not, briefly explain why, and make as few changes as necessary to the code snippet to make it possible to determine whether Foo is a Stack or a Queue.

Question 2. [8 MARKS]

Suppose you are given a class Stack, which implements the Stack ADT:

```
class Stack:
    def __init__(self):
       ''' Initialize a new stack instance. '''
       self.stack = []

    def push(self, val):
       ''' Push the value given by val onto the top of the stack. '''
       self.stack.append(val)

    def pop(self):
       ''' Remove and return the top item on the stack. '''
       return self.stack.pop()

    def size(self):
       ''' Return the number of elements in the stack. '''
       return len(self.stack)

    def is_empty(self):
       ''' Return True if the stack is empty, False otherwise. '''
       return self.stack == []
```

Rewrite the `pop` method to raise an exception called `StackEmptyException` if someone tries popping the stack when it is empty. First define the exception class, and then provide the new implementation of `pop`.

Question 3. [10 MARKS]

Here is the order in which nodes are visited in a postorder traversal of a tree:

```
K R M Q P A F
```

Here is the order in which nodes are visited in an inorder traversal of the same tree's left subtree:

```
K M R
```

Here is the order in which nodes are visited in a preorder traversal of the same tree's right subtree:

```
A P Q
```

Draw a tree consistent with the above. (There may be more than one such tree, but you only have to draw one.)

Question 4. [8 MARKS]

For each of the following statements, circle whether the statement is True or False. If the statement is true, no justification is necessary. If the statement is false, explain why it is false using no more than two or three sentences.

Part (a) [2 MARKS]

Searching for an element in a binary search tree is always more efficient than searching for an element in a list.

True False

Part (b) [2 MARKS]

An inorder traversal of a binary search tree will visit the tree's nodes in non-descending order of the nodes' values.

True False

Part (c) [2 MARKS]

The Queue ADT has "lower priority" than the Priority Queue ADT, and thus should never be used in your code. Instead, you should always use a Priority Queue ADT whenever a Queue ADT is required.

True False

Part (d) [2 MARKS]

Memoization is a technique that improves the efficiency of certain recursive algorithms by remembering solutions to subproblems so that they don't have to be computed repeatedly.

True False

Question 5. [10 MARKS]

In this question you will be solving the Mirrored Towers of Hanoi problem. There are three two-sided pegs, and initially the first peg has `n` discs placed on it, where `n` is even. The discs are arranged so that the top half of the peg has `n/2` discs, each of different size, with smaller discs on top of larger discs. The bottom half of the peg has the other `n/2` discs and is a mirror image of the top half of the peg. (See the figure below for the initial state when `n = 8`.)

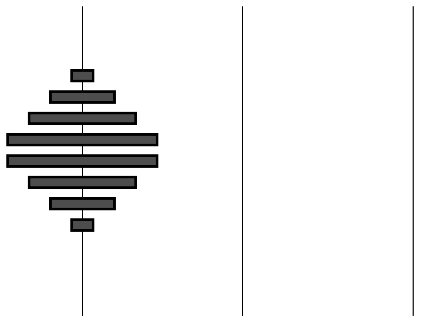

Figure 1: Initial state for n=8

Discs can be removed or added on either side of a peg. The goal is to move all discs from the first peg to the third peg, using the second peg, while obeying the following rules:

- Discs are always moved in pairs: one disc from the top half of a peg and one disc from the bottom half of the same peg.
- You can move exactly one pair of discs at a time.
- When moving a disc pair, both discs are transferred to the same peg. One of the discs is placed on the top half of the peg, and the other is placed on the bottom half of the same peg.
- You can never "sandwich" smaller discs in between larger discs.

Write a function `mirrored_hanoi(n, frompeg, topeg, withpeg)` that prints out a list of moves to make in order to get `n` discs from the `frompeg` to the `topeg` using the `withpeg`. For example, a print out from a call to `mirrored_hanoi(4, "A", "C", "B")` may look something like:

```
Move pair of discs from peg A to peg B.
Move pair of discs from peg A to peg C.
Move pair of discs from peg B to peg C.
```

Write your solution on the next page.

```
def mirrored_hanoi(n, frompeg, topeg, withpeg):
```

Question 6. [12 marks]

Implement a function `make_change(denoms, total)` that returns a list of all the possible ways to make change for the `total` number of dollars specified, assuming that the currency denominations that you are allowed to use are given by the `denoms` argument, and that you have an unlimited supply of each denomination.

The `denoms` argument is a tuple of arbitrary length, and each element in the tuple is unique and represents one of the currency denominations that you can use. You can assume that all currency denominations are positive integers, representing dollar values, and that `total` is a non-negative integer, also representing a dollar value.

Each element in the returned list should be a list whose i^{th} element is the number of bills of denomination `denoms[i]` used in making change for `total` dollars. There should be no duplicate elements in the returned list.

For example, the call `make_change((1,2),6)` should return the list `[[6,0], [4,1], [2,2], [0,3]]` (not necessarily in this order). This is because we can make change for 6 dollars using 6 one dollar bills and 0 two dollar bills, 4 one dollar bills and 1 two dollar bill, 2 one dollar bills and 2 two dollar bills, or 0 one dollar bills and 3 two dollar bills. There are no other ways to make change for 6 dollars using only 1 and 2 dollar currency denominations.

Hint: You may find it useful to create an additional helper function.

```
def make_change(denoms, total):
```

Use this page for rough work and for any answers that didn't fit.

Last Name: ____________________ **First Name:** ____________________