CISC-235*
Test #1
January 31, 2018

Student Number (Required) ______________________

Name (Optional) ________________________________

This is a closed book test. You may not refer to any resources.

This is a 50 minute test.

Please write your answers in ink. Pencil answers will be marked, but will not be re-marked under any circumstances.

The test will be marked out of 50.

Question 1	/10
Question 2	/15
Question 3	/10
Question 4	/15
TOTAL	/50

Question 1 (10 marks)

Suppose $f(n) \in \Theta(n^5)$ and $g(n) \in \Theta(n^7)$

Let $p(n) = f(n) + g(n)$

a) [4 marks] Determine the O classification of $p(n)$

Solution: we know $f(n) \leq c_1 * n^5$ and $g(n) \leq c_2 * n^7$ $\forall n \geq$ *some* n_0

Therefore $p(n) = f(n) + g(n) \leq (c_1 + c_2) * n^7$

Therefore $p(n) \in O(n^7)$

Marking: A solution that is incorrect but shows understanding of O classification should get 2/4

b) [4 marks] Determine the Ω classification of $p(n)$

Solution: we know $f(n) \geq c_1 * n^5$ and $g(n) \geq c_2 * n^7$ $\forall n \geq$ some n_0

Therefore $p(n) = f(n) + g(n) \geq c_2 * n^7$

Therefore $p(n) \in \Omega(n^7)$

Marking: A solution that is incorrect but shows understanding of Ω classification should get 2/4

c) [2 marks] If possible, determine the Θ classification of $p(n)$

Solution: from a) and b), we see $p(n) \in \Theta(n^7)$

Marking: A solution that is incorrect but shows understanding of O classification should get 1/2

Question 2 (15 marks)

Let **Stack** be a class that implements the stack data structure. Each instance of a **Stack** has three defined methods:

push(x)	add x to the top of the stack
pop()	remove and return the top value of the stack
isEmpty()	return true iff the stack is empty

and a single accessible attribute:

count	the number of items currently in the stack

Let S be a stack containing integer values. Write an algorithm that will take a positive integer k as a parameter and move the bottom k values of the stack to the top. If the stack contains $\leq$ k values, it should not be changed.

For example if the stack contains $\begin{matrix}3\\10\\9\\8\\14\\50\end{matrix}$ and k = 2, the final result should be $\begin{matrix}14\\50\\3\\10\\9\\8\end{matrix}$

Your algorithm is allowed to create and use other stacks but cannot declare arrays, linked lists or other data structures.

You may use the next page for your answer (though it should not require a full page!)

Page for answering Question 2

My Solution:

```
def move(k):
     if S.count <= k :
          return        // optional return statement
     else:
          Stack temp1 = new Stack
          Stack temp2 = new Stack
          for (i=S.count;i>k;i--) :
               temp1.push(S.pop())
          for (i=1;i<=k; i++) :
               temp2.push(S.pop())
          while (! temp1.isEmpty()) :
               S.push(temp1.pop())
          while (! temp2.isEmpty()) :
               S.push(temp2.pop())
          return        // optional return statement
```

Marking:

Syntax is not crucial (since it is just pseudocode) – for example the for loops could be written with Python syntax, or as "for i = 1 to k", etc. There are probably lots of other ways to solve the problem – I think that what I have shown here is the simplest solution.

Solutions that correctly solve the problem should get full marks, even if the method differs from my solution.

Solutions that are correct but have a minor error (such as a loop which goes too far or ends too soon) should get about 13/15

Solutions that have the right idea but have significant errors – such as moving k elements from the top to the bottom, or making other modifications to the final order of the elements of the stack, should get about 10/15

Solutions that don't really come close to solving the problem but demonstrate a good understanding of stack operations should get about 8/15

Solutions that show only a weak understanding of stack operations should get about 4/15
A blank page should get 0/15

Question 3 (10 marks)

Here is the recursive **pre-order traversal** algorithm for binary trees.

```
Pre_Order(v):    # v is a vertex in a binary tree
    if v == nil:
        return
    else:
   print v.value
       Pre_Order(v.left_child)
       Pre_Order(v.right_child)
       print v.value
```

Write a non-recursive version of the same algorithm (ie for a tree with a root called **root**, the given algorithm Pre_Order(root) and your algorithm your_Alg(root) must produce exactly the same output)

If you like, your algorithm may use the Stack class as defined in Question 2 of this test.

Solution:

As many people realized, this question contains an error: the given algorithm is NOT a Pre-Order traversal. It is actually sort of a combination of Pre-Order and Post-Order.

Some students wrote an iterative version of Pre-Order, and some wrote an iterative version of the given genetically modified algorithm. I'll show Stack-based solutions for both.

Stack-based version of pre-order traversal:

The logic of this algorithm is simple. We initialize the stack with the root variable, then loop while the stack is non-empty. At each vertex we print the value, then push its children onto the stack in “right then left” order so that the left child will be popped off and processed before the right child.

```
def pre_order(root):
    if (root != nil):
        Stack s = new Stack
        s.push(root)

        while (! s.isEmpty()) :
            c = s.pop()
            print c.value
            if (c.right_child != nil):
                s.push(c.right_child)
            if (c.left_child != nil):
                s.push(c.left_child)

    return    // return statement is optional
```

Stack-based version of Frankenstein traversal:

This algorithm is more complex. Since each value is to be printed twice, we pop the top vertex off the stack and print it (as in the pre-order traversal), then check to see if it has children. If not, we simply print it again, as required. If it does have children but the most-recently popped vertex is a child of this vertex, then it is time to do the second printing of this vertex. If neither of these situations hold, we push the vertex back on the stack, followed by its children.

```
def monster(root):
    if (root != nil):
        Stack s = new Stack
        Vertex last_pop = root     // just an initialization
        s.push(root)

        while (! s.isEmpty()) :
            c = s.pop()
            print c.value
            if (c.left_child == nil) &&
                (c.right_child == nil) : // no children
                print c.value
            else if (last_pop == c.left_child) ||
                     (last_pop == c.right_child) :
                       // it's time to print this vertex again
                print c.value
                //  the action in the two clauses above is
                //  identical - they can be combined
            else :
                //  put c and its children back on the stack
                s.push(c)
                if (c.right_child != nil) :
                    s.push(c.right_child)
                if (c.left_child != nil) :
                    s.push(c.left_child)
            last_pop = c

    return    // return statement is optional
```

Marking Question 3:

Regardless of which interpretation of the question each student chose, fully correct solutions (which may or may not resemble mine) should get full marks.

Solutions that are close to correct should get about 7/10 or 8/10. However since the second interpretation required a more difficult solution, please be more lenient when marking those solutions.

Solutions that demonstrate a good understanding of what was required should get about 6/10, even if the solution has significant errors.

Solutions that show limited understanding of how to work with binary trees should get about 3/10

A blank page should get 0/10

Question 4 (15 marks)

Suppose we have a **Binary Search Tree** containing a set of n integers, some of which may be duplicates.

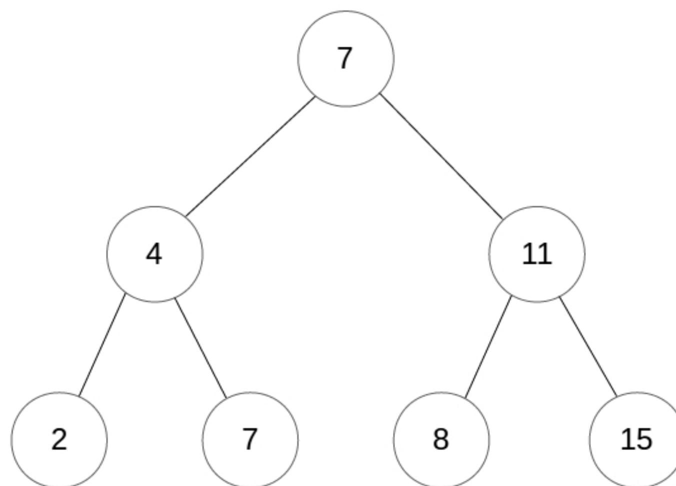
Write an algorithm called `closest` that takes two parameters:

t, which is a tree

x, which is a target integer

and returns the value in the tree that is closest to x. If there are two different values that tie for closeness, your algorithm should return the smaller of the two. Your algorithm should search only as much of the tree as it needs to.

For example, if the tree t is



then `closest(t,5)` should return the value `4` and `closest(t,10)` should return the value `11`

You may use iteration or recursion.

Please write your answer on the next page.

Page for answering Question 4

My solutions – iterative and recursive:

The logic behind both of these algorithms is the same: we basically execute the standard search algorithm for x. At each vertex on the search path, we will either have found x, or we are at a value < x, or > x.

If we have found x we can return it as the value that is closest to x

If we are at a value < x, no value in the left subtree can be closer to x than this one so we can continue the search in the right subtree

If we are at a value > x, no value in the right subtree can be closer to x than this one so we can continue the search in the left subtree

The only change to the search algorithm is to keep track of the seen value that is closest to x. In the iterative version this is done by using a variable called "closest" that holds the closest value to x seen so far. In the recursive version this is done by having each vertex on the search path return the value in its whole subtree that is closest to x. The value returned by the root is the closest in the entire tree.

Iterative:

```
def closest(t,x):
     if (t.root == nil) :
          **ERROR**
     else:
          current = t.root
          closest = current.value
          diff = abs(closest - x)
          done = FALSE
          while (! done) :
               if (current == nil):
                    done = TRUE
               else if (current.value == x) :
                    closest = x
                    done = TRUE
               else :
                    this_diff = abs(current.value - x)
                    if (this_diff < diff):
                         diff = this_diff
                         closest = current.value
                    else if (this_diff == diff) && (current.value < closest):
                         diff = this_diff
                         closest = current.value
                    if (current.value > x):
                         current = current.left_child
                    else:
                         current = current.right_child

          return closest
```

Recursive:

```
def closest(t,x):
     if (t.root == nil):
          **ERROR**
     else:
          return rec_closest(t.root,x)

def rec_closest(v,x):
     if (v.value == x):
          return x
     else:
          d1 = abs(v.value - x)
          if (v.value > x) && (v.left_child != nil):
               lc = rec_closest(v.left_child,x)
               d2 = abs(lc - x)
               if (d2 <= d1):
                    return lc
               else:
                    return v.value
          else if (v.right_child != nil):
               rc = rec_closest(v.right_child,x)
               d2 = abs(rc - x)
               if (d2 < d1):
                    return d2
               else:
                    return v.value
          else:
               return v.value
```

Marking:

Students do not have to raise an error condition when the tree is empty

Students do not have to duplicate my algorithms!

Solutions can be written in any form of code, pseudo-code or even text as long as it is clear how every step is accomplished. For example "current = current.left_child" could be expressed as "go down to the next vertex on the left"

A solution that finds the closest value to x without exploring parts of the tree that cannot contain the solution should get 15/15

A solution that finds the closest value to x but explores parts of the tree that cannot contain the solution should get 10/15

A solution that only finds the closest value to x sometimes should get around 7/15

A solution that does not really address the problem but shows an understanding of how binary trees are structured should get around 5/15

A solution that does not solve the solve the problem and does not show an understanding of binary trees should get around 2/15

A blank page should get 0/15