CSCA48 Fall 2015 Final Exam
Duration — 150 minutes
Aids allowed: none

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

Instructor: Brian Harrington and Anna Bretscher

Last Name: ______________________

First Name: ______________________

UtorID (Markus Login): ______________________

*Do **not** turn this page until you have received the signal to start.*

This exam consists of 6 questions on 20 pages (including this one). *When you receive the signal to start, please make sure that your copy is complete.* Proper documentation is required for all functions and code blocks. If you use any space for rough work, indicate clearly what you want marked. Please read all questions thoroughly before starting on any work.

We have provided you with grids for your answers, this is simply to help you show the indentation of your code and you are not required to adhere to the grids in any specific way. For you reference, we also give you an **extra copy of all code on the last two pages**. You may tear these two pages off the exam.

The University of Toronto's Code of Behaviour on Academic Matters applies to all University of Toronto Scarborough students. The Code prohibits all forms of academic dishonesty including, but not limited to, cheating, plagiarism, and the use of unauthorized aids. Students violating the Code may be subject to penalties up to and including suspension or expulsion from the University.

# 1:	______/ 4
# 2:	______/ 6
# 3:	______/ 6
# 4:	______/ 5
# 5:	______/12
# 6:	______/ 7
TOTAL:	______/40

[Use the space below for rough work. This page will not be marked unless you clearly indicate the part of your work that you want us to mark.]

Question 1. [4 MARKS]

Sort the following functions (of positive integer n) in big-Oh order. I.e., write them in an order so that each function is in big-Oh of the one immediately after it. If functions are big-Oh of each other, then group them together using "==".

For example, given three functions f_1, f_2 and f_3 such that $f_1 \in \mathcal{O}(f_2)$ and $f_2 \in \mathcal{O}(f_1)$ and $f_1, f_2 \in \mathcal{O}(f_3)$, we would order them $f_1 == f_2, f_3$.

$$2n^3,\ 2^n + n^2 + \log n,\ 3^n,\ 4(\log n),\ 48,\ 7n^2,\ 92^n,\ \log(n^2)$$

Question 2. [6 marks]

Recall Assignment 2 on *regular expressions* and the rules we used to build regular expression trees.

Part (a) [3 marks]

Draw the regex tree for the regex `(0*.(2*|(0.1))*)`.

Part (b) [3 marks]

Complete the following function by adding **one** Python statement, plus a comment that explains that statement.

Hint: *No loops are needed. First work out how one can determine the number of nodes without forming a tree. Then describe this in a comment. Finally use str methods to write your Python statement.*

```
def num_nodes(r):
    ''' (str) -> int
    Return the number of nodes in the regex tree that represents the
    regex r.
    REQ: r is a valid regex.
    '''
```

Question 3. [6 MARKS]

Consider inserting the integers 1 through 6 into an initially empty *max* heap using the algorithm described in class. Recall that during the bubble up process, a key within a node is sometimes swapped with the key in its parent node. We are interested in the ordering of the sequence of insertions, and the total number of key swaps.

Part (a) [2 MARKS]

What is the maximum number of key swaps that can be incurred by a sequence of operations that inserts the integers 1 through 6 into a max heap? Circle exactly one of the choices below.

5 6 7 8 9

Part (b) [2 MARKS]

Give a sequence of insertions that forces the maximum number of key swaps given in Part (a).

Part (c) [2 MARKS]

c) Give two sequences of insertions of the integers 1 through 6 that incurs no key swaps. The first sequence must insert 4 as early as possible, and the second sequence must insert 4 as late as possible.

First sequence:

Second sequence:

```
from math import *

def f1(a, b, r):
    ''' (int, int, float) -> list of (int, int)
    REQ: 0 < r < 1.
    '''
    if a >= b:
        return []  # return empty list
    c = floor(a + (b-a)*r)
    return f1(a, c, r) + f1(c+1, b, r) + [(a, b)]  # concatenate 3 lists
```

[Use the space below for rough work. This page will not be marked unless you clearly indicate the part of your work that you want us to mark.]

Question 4. [5 marks]

Consider the function `f1` on the previous page[1]:

In the space below, write the output of the following code:

```
print("STEP 1:", f1(1, 4, 0.5))
print("STEP 2:", f1(10, 12, 0.1))
```

[1] Boy, this sure looks familiar... good thing you had several weeks to review it, huh?

```
class CustomerNode:
    def __init__(self, ticket_num, cust_name, server_name, next):
        '''(IntNode, int, str, str, CustomerNode) -> NoneType
        Create a new CustomerNode with ticket number ticket_num,
        customer name cust_name, server name server_name and
        pointing to next.
        REQ: server_name must be either "Anna" or "Brian".
        '''

        self._ticket_num = ticket_num
        self._cust_name = cust_name
        self._server_name = server_name
        self.next = next

    def __repr__(self):
        """(CustomerNode) -> str
        Return a string representing self.
        """

        return ("CustomerNode(" + repr(self._ticket_num) + ", " +
                repr(self._cust_name) + ", " + repr(self._server_name) +
                ", " + repr(self.next) + ")")

    def get_ticket_num(self):
        return self._ticket_num

    def get_cust_name(self):
        return self._cust_name

    def get_server_name(self):
        return self._server_name
```

Question 5. [12 MARKS]

Anna's Deli has just two servers, Anna and Brian. It's a small operation. To control their long queues of customers, they use a roll of numbered tickets. Each successive ticket's number is one greater than its previous ticket's number. On entry every customer takes the next available ticket and lines up either in Anna's queue or in Brian's queue. Sometimes Anna or Brian must leave to deliver a lecture[2]. When this happens the two queues are merged into a single queue, and all customers are served in the order they arrived (i.e., ordered by their ticket number). When Anna or Brian is able to return, the queue splits again into Anna's queue and Brian's queue.

Nick decided to help Anna and Brian by writing some code to simulate the queues at Anna's Deli. He used linked lists to represent queues. First he created the class `CustomerNode` for the nodes in his linked list. Then he wrote a function called `reclistmerge()`, which takes two sorted (by ticket number) linked lists of customers, and returns a single sorted list of customers from the two input lists. His function is given here:

```
def reclistmerge(L1, L2):
    ''' (CustomerNode, CustomerNode) -> CustomerNode






     '''
    #
    #
    if L1 == None:
        return L2
    if L2 == None:
        return L1

    # debug printing
    print(L1.get_ticket_num(), L2.get_ticket_num())

    #
    #
    if L1.get_ticket_num() < L2.get_ticket_num():
        #
        #
        L1.next = reclistmerge(L1.next, L2)
        return L1
    else:
        #
        #
        L2.next = reclistmerge(L1, L2.next)
        return L2
```

[2] obviously you all need to pay more tuition so your lecturers don't have to hold down 2 jobs

Part (a) [3 marks]

Nick is trying to convince himself that his code is working properly. Notice the debugging `print` line in `recmerglist()`. Give the output of the following lines of code:

```
L_A = CustomerNode(2, "Alice", "Anna",
      CustomerNode(3, "Bob", "Anna",
      CustomerNode(5, "Carol", "Anna", None)))
L_B = CustomerNode(1, "Ziggy", "Brian",
      CustomerNode(4, "Yaz", "Brian",
      CustomerNode(6, "Xavier", "Brian", None)))
L = reclistmerge(L_A, L_B)
```

Part (b) [4 marks]

Then disaster happened...the CODE DECOMMENTER struck!!! The `docstring` was erased and all internal comments were removed. Please help Nick to recomment his code in function `reclistmerge()`. You should complete the `docstring` and write an informative comment wherever there is a group of #s on the given code on the previous page.

```
def listsplit(L):
    '''
    (CustomerNode) -> (CustomerNode, CustomerNode)
    Split the linked list headed by L into 2 lists, first one with server
    "Anna" and second one with server "Brian".
    Within each list, the original order of nodes is preserved.
    Return the heads of the two lists.
    '''

    # This version uses recursion (and no loops).

    # if list is empty, then return empty lists
    # initialize lists to return
    # list is nonempty, so place first node in appropriate list, then
    # return the split lists
    # split the rest of the list, and make the first node be the head of
    # the corresponding split list


    (L_A.next, L_B) = listsplit(L.next)
    (L_A, L_B.next) = listsplit(L.next)
    else:
    if L == None:
    if L.get_server_name() == "Anna":
    L_A = L_B = None  # A means Anna, B means Brian
    L_A = L
    L_B = L
    return (L_A, L_B)
    return (L_A, L_B)
```

Part (c) [5 marks]

Whenever Anna or Brian return to serve customers, the single queue splits into two queues with customers returning to their preferred server line. Nick also wrote a function to split the merged queue back into two queues. The function `listsplit()` is the "*inverse*" of `listmerge()` in the sense that

- `listmerge(listsplit(L))` and `L` are equal,
- `listsplit(listmerge(L_A, L_B))` and `(L_A, L_B)` are equal.

Nick was having a really rotten day. Just as he finished re-commenting his `reclistmerge()` code the CODE MANGLER struck. His `listsplit()` method on the previous page had all lines scrambled. Help Nick correct `listsplit()`.

```
def listsplit(L):
    ''' (CustomerNode) -> (CustomerNode, CustomerNode)
    Split the linked list headed by L into 2 lists, first one with server
    "Anna" and second one with server "Brian".
    Within each list, the original order of nodes is preserved.
    Return the heads of the two lists.
    '''
```

[Use the space below for rough work. This page will not be marked unless you clearly indicate the part of your work that you want us to mark.]

Question 6. [7 marks]

Consider the following `BTNode` class to hold integers in a binary tree.

```
class BTNode:
    def __init__(self, i):
    ''' (BTNode, int) -> None
    Create a new BTNode with integer i.
    '''
    self.my_int = i
    self.left = self.right = None  # pointers to left and right subtrees
```

We also assume the availability of a standard queue module with methods to instantiate a new queue, enqueue an object, dequeue an object, and test if a queue is empty. Consider the following function.

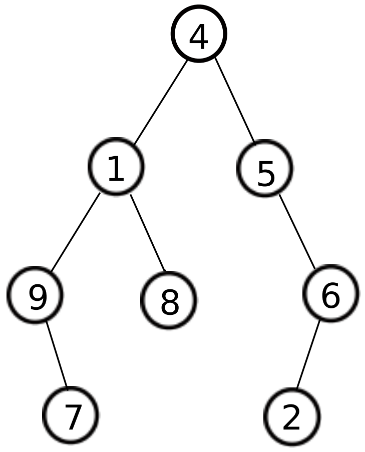
```
    def hmmm(root):
        ''' (BTNode) -> BTNode)
        '''
        return_node = None
        q = Queue()
        if root != None:
            q.enqueue(root)

        while (not q.is_empty()):
            node = q.dequeue()
            print(node.my_int)
            if node.left != None:
                q.enqueue(node.left)
            if node.right != None:
                q.enqueue(node.right)
            if node.my_int % 2 == 0:
                return_node = node

        return return_node
```

Part (a) [2 MARKS]

Suppose we run the line of code,

`hmmm(mytree)`

where mytree is the root of this tree.



Give the output that would be printed and circle the node on the tree that would be returned.

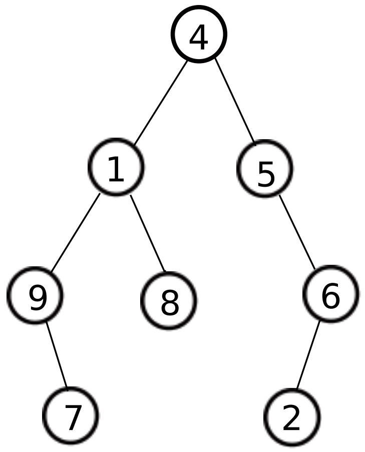
Output:

Part (b) [3 MARKS]

Describe what `hmmm(root)` returns.

Part (c) [2 MARKS]

Suppose that we now replace the `queue` ADT with a `stack` ADT.



Now each `enqueue` call is replaced by a `push` call and each `dequeue` call is replaced by a `pop` call. Repeat Part (a) using a `stack`. List your output below and circle the appropriate node on the tree that would be returned.

Output:

Short Python function/method descriptions:
You may tear this page off, but if you do so, you must not include any work on it (front or back) that you wish to have marked.

```
__builtins__:
  abs(number) -> number
    Return the absolute value of the given number.
  max(a, b, c, ...) -> value
    With two or more arguments, return the largest argument.
  min(a, b, c, ...) -> value
    With two or more arguments, return the smallest argument.
  isinstance(object, class-or-type-or-tuple) -> bool
    Return whether an object is an instance of a class or of a subclass thereof.
    With a type as second argument, return whether that is the object's type.
  int(x) -> int
    Convert a string or number to an integer, if possible.  A floating point argument
    will be truncated towards zero.
  str(x) -> str
    Convert an object into a string representation.

str:
  S.count(sub[, start[, end]]) -> int
    Return the number of non-overlapping occurrences of substring sub in
    string S[start:end].  Optional arguments start and end are
    interpreted as in slice notation.
  S.find(sub[,i]) -> int
    Return the lowest index in S (starting at S[i], if i is given) where the
    string sub is found or -1 if sub does not occur in S.
  S.isalpha() --> bool
    Return True if and only if all characters in S are alphabetic
    and there is at least one character in S.
  S.isdigit() --> bool
    Return True if and only if all characters in S are digits
    and there is at least one character in S.
  S.islower() --> bool
    Return True if and only if all cased characters in S are lowercase
    and there is at least one cased character in S.
  S.isupper() --> bool
    Return True if and only if all cased characters in S are uppercase
    and there is at least one cased character in S.
  S.lower() --> str
    Return a copy of S converted to lowercase.
  S.replace(old, new) -> str
    Return a copy of string S with all occurrences of the string old replaced
    with the string new.
  S.split([sep]) -> list of str
    Return a list of the words in S, using string sep as the separator and
    any whitespace string if sep is not specified.
  S.startswith(prefix) -> bool
    Return True if S starts with the specified prefix and False otherwise.
  S.strip() --> str
    Return a copy of S with leading and trailing whitespace removed.
  S.upper() --> str
    Return a copy of S converted to uppercase.
```

```
list:
  append(...)
    L.append(object) -- append object to end
  count(...)
    L.count(value) -> integer -- return number of occurrences of value
  index(...)
    L.index(value, [start, [stop]]) -> integer -- return first index of value.
    Raises ValueError if the value is not present.
  insert(...)
    L.insert(index, object) -- insert object before index
  pop(...)
    L.pop([index]) -> item -- remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.
  remove(...)
    L.remove(value) -- remove first occurrence of value.
    Raises ValueError if the value is not present.

set:
  pop(...)
    Remove and return an arbitrary set element.
    Raises KeyError if the set is empty.

dict:
  keys(...)
    D.keys() -> a set-like object containing all of D's keys
  get(...)
     D.get(k[,d]) -> returns D[k] if k is in D, otherwise returns d.  d defaults to None.

object:
  __init__(...)
    x.__init__(...) initializes x; called automatically when a new object is created
  __str__(...)
    x.__str__() <==> str(x)
```

```
class CustomerNode:
    def __init__(self, ticket_num, cust_name, server_name, next):
        '''(IntNode, int, str, str, CustomerNode) -> NoneType
        Create a new CustomerNode with ticket number ticket_num,
        customer name cust_name, server name server_name and
        pointing to next.
        REQ: server_name must be either "Anna" or "Brian".
        '''

        self._ticket_num = ticket_num
        self._cust_name = cust_name
        self._server_name = server_name
        self.next = next

    def __repr__(self):
        """(CustomerNode) -> str
        Return a string representing self.
        """

        return ("CustomerNode(" + repr(self._ticket_num) + ", " +
                repr(self._cust_name) + ", " + repr(self._server_name) +
                ", " + repr(self.next) + ")")

    def get_ticket_num(self):
        return self._ticket_num

    def get_cust_name(self):
        return self._cust_name

    def get_server_name(self):
        return self._server_name


def reclistmerge(L1, L2):
    ''' (CustomerNode, CustomerNode) -> CustomerNode
    '''
    #
    if L1 == None:
        return L2
    if L2 == None:
        return L1

    # debug printing
    print(L1.get_ticket_num(), L2.get_ticket_num())
    #
    if L1.get_ticket_num() < L2.get_ticket_num():
        #
        L1.next = reclistmerge(L1.next, L2)
        return L1
    else:
        #
        L2.next = reclistmerge(L1, L2.next)
        return L2
```

```
def listsplit(L):
    '''  (CustomerNode) -> (CustomerNode, CustomerNode)
    Split the linked list headed by L into 2 lists, first one with server
    "Anna" and second one with server "Brian".
    Within each list, the original order of nodes is preserved.
    Return the heads of the two lists.
    '''
    # This version uses recursion (and no loops).
    # if list is empty, then return empty lists
    # initialize lists to return
    # list is nonempty, so place first node in appropriate list, then
    # return the split lists
    # split the rest of the list, and make the first node be the head of
    # the corresponding split list

    (L_A.next, L_B) = listsplit(L.next)
    (L_A, L_B.next) = listsplit(L.next)
    else:
    if L == None:
    if L.get_server_name() == "Anna":
    L_A = L_B = None  # A means Anna, B means Brian
    L_A = L
    L_B = L
    return (L_A, L_B)
    return (L_A, L_B)
*****************************************************************************************
class BTNode:
    def __init__(self, i):
    ''' (BTNode, int) -> None
    Create a new BTNode with integer i.
    '''
    self.my_int = i
    self.left = self.right = None  # pointers to left and right subtrees

def hmmm(root):
    ''' (BTNode) -> BTNode)
    '''
    return_node = None
    q = Queue()  # create an empty queue
    if root != None:
        q.enqueue(root)  # enqueue root of input binary tree

    while (not q.is_empty()):  # while q not empty
        node = q.dequeue()  # dequeue node from q
        print(node.my_int)
        if node.left != None:
            q.enqueue(node.left)  # enqueue node.left into q
        if node.right != None:
            q.enqueue(node.right)  # enqueue node.right into q
        if node.my_int % 2 == 0:  # if my_int is even
            return_node = node

    return return_node
```