UNIVERSITY OF TORONTO
Faculty of Arts and Science

Final Exam

CSC148H1F

December 15, 2015 (**3 hours**)

Examination Aids: Provided aid sheet (back page, detachable!)

Name:

Student Number:

Please read the following guidelines carefully!

- This examination has **7** questions. There are a total of **14 pages**.
- You may use helper functions unless explicitly told not to.
- Any question you leave blank or clearly cross out your work and write "I don't know" is worth **10% of the marks**.
- You must earn a grade of **at least 40% to pass this course.**

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Total
Grade								
Out Of	10	6	8	8	8	12	10	62

Take a deep breath.

This is your chance to show us

How much you've learned.

We **WANT** to give you the credit

That you've earned.

A number does not define you.

It's been a real pleasure teaching you this term.

Good luck!

1. Answer each of the following using brief sentences or point-form. Diagrams and code examples are encouraged.

   (a) [2] Explain the difference between an **attribute** of a class and a **method** of a class.

   (b) [2] We have an abstract class `MyClass` which has some subclasses. `MyClass` has the following method:

   ```
   def my_method(self, n):
       """<docstring omitted>"""
       raise NotImplementedError()
   ```

   What is the purpose of having this method inside `MyClass`, even though it is not implemented?

   (c) [2] Which is faster: removing an item from the front of a Python list, or from the back? Why?

   (d) [2] Define the term **alias**, and show how to create an alias in Python code.

   (e) [2] Explain one benefit of marking a class attribute or method as **private**.

2. (a) [2] Implement the following function, which operates on a stack and removes its second-highest element. **Assume** that the input stack has at least two items. You should only use the Stack ADT methods (see aid sheet) to operate on the stack.

```
def remove_second(stack):
  """Return and remove the second-highest item on <stack>.
  Precondition: <stack> has at least two items.
  @type stack: Stack
  @rtype: object

  >>> s = Stack()
  >>> s.push(1)
  >>> s.push(2)
  >>> s.push(3)
  >>> s.push(4)
  >>> remove_second(s)
  3
  >>> s.pop()
  4
  """
```

(b) [4] Generalize your work from part (a) to implement the stack function `remove_nth`, which takes a stack and a positive integer n, and removes the n-th highest item from the stack. **Also**, raise a `ValueError` if the stack has fewer than n items. Once again, you should only use the Stack ADT methods on the stack.

```
def remove_nth(stack, n):
  """Return and remove the <n>-th highest item on <stack>.

  Raise a ValueError (this is built-in) if the stack has fewer than <n> items.

  @type stack: Stack
  @type n: int
    Precondition: n >= 1.
  @rtype: object
  """
```

3. (a) [4] You are designing a program to help landlords keep track of buildings and the people who are renting rooms in them. A building in the system has an address (stored as a string) and an age (in number of years), and should keep track of who is renting in that building. Each renter in the system has a name and lives in one building, and pays a certain amount of rent each month. Two people living in the same building can pay different amounts of rent.

   In the space below, write the **class docstrings** of a "building" class and a "renter" class that you could define as part of the desired program. Your docstrings should clearly state the type and description of all attributes; all attributes may be public.

(b) [4] Define a building method which takes an amount of money n, and permanently increases the rent paid by each renter in that building by n. Include both a docstring and implementation in your solution; however, you do *not* need to write any doctests. Your solution must be consistent with your class docstrings in part (a).

4. (a) [5] Implement a linked list method **insert_sorted**, which takes a **sorted linked list**, and an item, and inserts that item into the correct spot, keeping the list sorted. Note that this is a *mutating* method.

You must use a loop, and may not use any other **LinkedList** methods in your solution. We are looking for you to correctly access and set attributes of the **LinkedList** and/or **_Node** classes in your solution.

Hint: handle the cases of an empty linked list and the item to insert being smaller than all items in the linked list separately.

```
def insert_sorted(self, item):
  """Insert an item into <self> in the correct position.

  Precondition: <self> is sorted in non-decreasing order.
  <self> must still be sorted after this method completes.

  @type self: LinkedList
  @type item: object
  @rtype: None

  >>> lst = LinkedList([3, 7, 10])    # [3 -> 7 -> 10]
  >>> lst.insert_sorted(lst, 5)       # lst is [3 -> 5 -> 7 -> 10]
  """
```

(b) [3] What is the **worst-case** asymptotic (Big-Oh) running time of your method? Justify your answer.

5. (a) [**2**] Consider the following method, which attempts to mutate a **recursive linked list** (`LinkedListRec`) by inserting an item directly after the first item (**assume** the first item exists).

```
def insert_after_root(self, item):
  """Insert <item> immediately after the first item in <self>.
  Precondition: <self> has at least one item.

  @type self: LinkedListRec
  @type item: object
  @rtype: None

  >>> lst = LinkedListRec([3, 0, 10])    # [3 -> 0 -> 10]
  >>> lst.insert_after_root(4)           # lst is [3 -> 4 -> 0 -> 10]
  """
  self._rest._rest = self._rest
  self._rest._first = item
```

Sadly, this code is incorrect. State the output of the following, and **explain what goes wrong.**

```
>>> lst = LinkedListRec([3, 0, 10])
>>> lst.insert_after_root(4)
>>> lst._rest._rest._first
```

(b) [**2**] Here's a second attempt at implementing this method:

```
def insert_after_root(self, item):
  if self._rest.is_empty():
    self._rest = LinkedListRec([item])
  else:
    temp = self._rest._first
    self._rest._first = item
    self._rest.insert_after_root(temp)
```

What is the worst-case asymptotic running time of this algorithm? Justify your answer.

(c) [2] Give an implementation of this method (*must* be different from the ones above) that always runs in constant time. You may not use any `LinkedListRec` methods other than the constructor and `is_empty`.

```
def insert_after_root(self, item):
```

(d) [2] Briefly explain what is meant by "**constant time**," and why your above implementation always runs in constant time.

6. (a) [3] Define the **call stack**, and explain what information it stores about function calls in Python.

(b) [2] Here is a recursive function which operates on non-negative integers.

```
def my_f(n):
  if n == 0:
    return 0
  else:
    return 1 + my_f(n - 1)
```

Even though this function is correct, we run into a problem when calling it on moderately large inputs in Python. Explain why we get the following Python error.

```
>>> my_f(2000)
RuntimeError: maximum recursion depth exceeded...
```

(c) [2] Here is an incorrect implementation of a nested list function which searches for an item in a nested list. Explain the problem with this implementation. (You can write your answer beside the code.)

```
def search(obj, item):
  if isinstance(obj, int):
    return obj == item
  elif len(obj) == 0:
    return False
  else:
    for nested_list in obj:
      if search(nested_list, item):
        return True
      else:
        return False
```

(d) [2] David says, "The worst-case running time for searching in a binary search tree is O(log n), where n is the number of items in the tree." Is this statement true or false? Justify your answer.

(e) [3] Determine the asymptotic (Big-Oh) running time of the following function, which takes two positive integers as input. You may assume that the second argument `m` is a power of two.

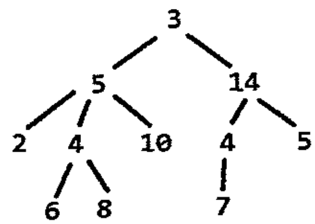
Hint: you may find it helpful to draw a diagram of the recursive calls.

```
def f(n, m):
  for i in range(n):
    print(i + m)

  if m == 1:
    return 1
  else:
    x = f(n, m / 2)
    y = f(n, m / 2) + 1
    return x * y
```

7. Recall that the *depth* of an item in a tree is the distance between itself and the root of the tree, counting items (so the root is always at depth 1).

   Your goal is to implement the `Tree` method `num_at_depth`, which takes a positive integer `d`, and returns the number of items at that depth.



   (a) [1] First, suppose we have a variable `tree` which refers to the above tree. State the output of `tree.num_at_depth(3)`.

   (b) [1] Note that the above tree has two subtrees (connected to the root). Fill in the output of both of the following expressions (assume the `_subtrees` list is in left-to-right order).

```
>>> tree._subtrees[0].num_at_depth(2)
```

```
>>> tree._subtrees[1].num_at_depth(2)
```

   (c) [4] Implement the method `num_at_depth`. You may not use any `Tree` methods here, other than `is_empty`.

```
def num_at_depth(self, d):
    """Return the number of items at depth <d> in this tree.

    @type self: Tree
    @type d: int
    @rtype: int
    """
```

(d) [4] Recall that the **width** of a tree is the maximum number of items at a fixed depth in the tree. For example, the tree on the previous page has **width 5**, since there is **1** item at depth 1, **2** items at depth 2, **5** items and depth 3, and **3** items at depth 4. (5 is the biggest number.) The width of an empty tree is 0.

Assume you have access to a tree method `height`, which returns the height of a given tree. Using it and `num_at_depth` or otherwise, implement the `Tree` method `width` below. (Note that you can get full marks on this question even if you don't complete part (c).) You may not use other Tree methods.

Hint: you may, if you wish, use the built-in `max` function, which takes a list of numbers and returns the maximum number in the list.

```
def width(self):
    """Return the width of this tree.

    @type self: Tree
    @rtype: int
    """
```

Use this page for rough work.

Use this page for rough work.

Data types

```
None
3, -4, 1.01, -2.0
True, False
'Hello, world!\n'
[1, 2.0, 'hi']
{'hi': 3, 'bye': 100}
```

Basic operators

```
True and False, True or False, not True
1 + 3, 1 - 3, 1 * 3
5 / 2 == 2.5, 5 // 2 == 2, 5 % 2 == 1
'hi' + 'bye'            # 'hibye'
[1, 2, 3] + [4, 5, 6]  # [1, 2, 3, 4, 5, 6]
```

List methods

```
lst = [1, 2, 3]
len(lst)            # 3
lst[0]              # 1
lst[0:2]            # [1, 2]
lst[0] = 'howdy'    # lst == ['howdy', 2, 3]
lst.append(29)      # lst == ['howdy', 2, 3, 29]
lst.pop()           # lst == ['howdy', 2, 3], returns 29
lst.pop(1)          # lst == ['howdy', 3], returns 2
lst.insert(1, 100)  # lst == ['howdy', 100, 3]
lst.extend([4, 5])  # lst == ['howdy', 100, 3, 4, 5]
3 in lst            # returns True
```

Control flow

```
if x == 5:
    y = 1
elif 4 <= 100:
    z = 2
else:
    y = 100

for i in [0, 1, 2, 3]:    # or, "for i in range(4):"
    print(i)

j = 1
while j < 10:
    print(j)
    j = j * 2
```

Exceptions

```
raise IndexError()
```

Class syntax

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def size(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

p = Point(3, 4)   # constructor
p.x               # attribute access: returns 3
p.size()          # method call: returns 5.0

class MyWeirdClass(Point):  # inheritance
    pass
```

Linked List (iterative)

```
class _Node:
    """A node in a linked list.

    === Attributes ===
    @type item: object
        The data stored in this node.
    @type next: _Node | None
        The next node in the list, or None if there are
        no more nodes in the list.
    """
    def __init__(self, item):
        """Initialize a new node storing <item>,
        with no 'next' node.

        @type self: _Node
        @type item: object
        @rtype: None
        """

class LinkedList:
    """A linked list implementation of the List ADT."""
    # === Private Attributes ===
    # @type _first: _Node | None
    #     The first node in the list,
    #     or None if the list is empty.

    def __init__(self, items):
        """Initialize a linked list with the given items.

        The first node in the linked list contains the
        first item in <items>.

        @type self: LinkedList
        @type items: list
        @rtype: None
        """
```

Linked List (recursive)

```
class LinkedListRec:
    """A recursive linked list."""
    # === Private Attributes ===
    # @type _first: object | None
    #     The first item in the list, or None
    #     if the linked list is empty.
    # @type _rest: LinkedListRec | None
    #     A list containing the other items after the
    #     first one, or None if the linked list is empty.
    #
    # === Representation Invariants ===
    # - _first is None if and only if _rest is None.
    #   This situation represents an empty list.

    def __init__(self, items):
        """Initialize a new linked list containing the
        given items.

        The first item in the linked list is the
        first item in <items>.

        @type self: LinkedListRec
        @type items: list
        @rtype: None
        """

    def is_empty(self):
        """Return whether this linked list is empty.

        @type self: LinkedListRec
        @rtype: bool
        """
```

Stack and Queues

```
s = Stack()
s.is_empty()
s.push(10)
s.pop()

q = Queue()
q.is_empty()
q.enqueue(10)
q.dequeue()
```

Tree

```
class Tree:
    # === Private Attributes ===
    # @type _root: object | None
    #     The tree's root item, or None.
    # @type _subtrees: list[Tree]
    #     A list of all subtrees of the tree.
    #
    # === Representation Invariants ===
    # - If _root is None then _subtrees is empty.
    #   This represents an empty Tree.
    # - _subtrees doesn't contain any empty trees

    def __init__(self, root):
        """Initialize a new Tree with the given root value.
        If <root> is None, the tree is empty.

        @type self: Tree
        @type root: object | None
        @rtype: None
        """

    def is_empty(self):
        """Return whether this tree is empty.
        @type self: Tree
        @rtype: bool
        """
```

BinarySearchTree class

```
class BinarySearchTree:
    # === Private Attributes ===
    # @type _root: object | None
    #     The BST's root value, or None.
    # @type _left: BinarySearchTree | None
    #     The left subtree, or None.
    # @type _right: BinarySearchTree | None
    #     The right subtree, or None.
    # === Representation Invariants ===
    # - If _root is None, then so are _left and _right.
    #   This represents an empty BST.
    # - If _root is not None, then _left, _right are BSTs.
    # - Every item in _left is <= _root, and
    #   every item in _right is >= _root.

  def __init__(self, root):
      """Initialize a new BST with a given root value.
      If <root> is None, the BST is empty.

      @type self: BinarySearchTree
      @type root: object | None
      @rtype: None
      """

    def is_empty(self):
      """Return whether this tree is empty.
      @type self: BinarySearchTree
      @rtype: bool
      """
```