Last Name: _____ **First Name:** _____

Student #: |_|_|_|_|_|_|_|_|_| **Signature:** _____

UNIVERSITY OF TORONTO MISSISSAUGA
APRIL 2014 FINAL EXAMINATION

CSC148H5S
Introduction to Computer Science
Daniel Zingaro
Duration - 3 hours
Aids: None

The University of Toronto Mississauga and you, as a student, share a commitment to academic integrity. You are reminded that you may be charged with an academic offence for possessing any unauthorized aids during the writing of an exam. Clear, sealable, plastic bags have been provided for all electronic devices with storage, including but not limited to: cell phones, tablets, laptops, calculators, and MP3 players. Please turn off all devices, seal them in the bag provided, and place the bag under your desk for the duration of the examination. You will not be able to touch the bag or its contents until the exam is over.

If, during an exam, any of these items are found on your person or in the area of your desk other than in the clear, sealable, plastic bag; you may be charged with an academic offence. A typical penalty for an academic offence may cause you to fail the course.

*Please note, you **CANNOT** petition to re-write an examination once the exam has begun.*

This final examination consists of 8 questions on 16 pages (including this one). When you receive the signal to start, please make sure that your copy of the examination is complete.

If you need more space for one of your solutions, use the last pages of the exam and indicate clearly the part of your work that should be marked.
You will be given 20% for any question to which you respond "cannot answer".

MARKING GUIDE

# 1:	_____/10
# 2:	_____/ 6
# 3:	_____/ 9
# 4:	_____/ 6
# 5:	_____/10
# 6:	_____/ 6
# 7:	_____/ 9
# 8:	_____/ 8

Good Luck! TOTAL: _____/64

Question 1. [10 MARKS]

There are two types of CS students: undergrad students and grad students. Undergrad students are allowed to enroll in undergraduate courses (those whose course codes start with 'U') whereas grad students can enroll in any courses they wish. It should be possible to obtain a list of the courses in which a CS student is enrolled.

There are three U of T campuses: StG, UTM, and UTSc. Undergrad CS students can be registered at any of the campuses, but grad CS students must register at StG. It should be possible to obtain a list of registered CS students at a campus, and to register a student at a campus. It is OK if you allow a student to be registered at multiple campuses.

Write code for classes that could be used to model the above specification. Use inheritance, instance variables, methods, and constructors, as appropriate. Include a suitable `__str__` method for each class, and provide a type contract and description for each method. You can use this page and the following page.

Question 2. [6 MARKS]

In lecture and lab, we studied the stack ADT, and used a Python list to implement the stack operations. In this question, you will use a queue to implement three stack operations: `is_empty`, `push`, and `pop`.

The stack ADT here has one queue (not a list!) as an instance variable that you must use to hold the stack elements. The queue has **only** `is_empty`, `enqueue`, and `dequeue` operations. You may use additional temporary lists or queues within your methods, but do not add anything to `__init__`.

```
class Stack:

  '''A last-in, first-out (LIFO) stack of items'''

  def __init__(self: 'Stack') -> None:
    '''A new empty Stack.'''
    self._data = Queue()

  # write is_empty, push, and pop
```

Question 3. [9 MARKS]

A **palindrome** is a string that is the same forward and backward. `reviver` and `kayak` are two different examples. The empty string is also a palindrome.

An **almost-palindrome** is a string that is a palindrome, or one that can be made a palindrome by moving each character at most one position to the left or right.

Part (a) [1 MARK] `radar` is an almost-palindrome, and `ervievr` is an almost-palindrome. Explain why for both cases.

Part (b) [1 MARK] `vierevr` is **not** an almost-palindrome. Give an argument for why this must be the case.

Part (c) [7 MARKS]

Write the following function. My solution uses a helper function that keeps track of whether the left character and right character have been swapped.

```
def is_almost_palindrome(s):
  '''Return True iff s is an almost-palindrome.'''
```

Question 4. [6 MARKS]

Part (a) [4 MARKS]

Any problem can be solved iteratively (as studied in CSC108) or recursively (as studied in CSC148). When you see a new problem for the first time, how do you know whether to use iteration or recursion? What are the features of the problem specification or solution approach that suggest one technique over the other? I am looking for what you have learned in general; do not provide specific examples as your answer (though you may include examples to help make your points).

Part (b) [2 MARKS]

Why is it that we often use iteration on a BST (as in Exercise 4) but rarely use iteration on a binary tree?

Question 5. [10 MARKS]

Here is the code for class BTNode that we used this semester for representing a node in a binary tree.

```
class BTNode:
  '''A node in a binary tree.'''

  def __init__(self: 'BTNode', item: object,
               left: 'BTNode'=None, right: 'BTNode'=None) -> None:
    '''Initialize this node.'''
    self.item, self.left, self.right = item, left, right
```

Write the following function. You may write helper functions as appropriate.

```
def is_bst(t: 'BTNode') -> bool:
  '''Return True iff t is a BST.

  >>> is_bst(BTNode(1, BTNode(0), BTNode(11, BTNode(4, BTNode(3),\
  BTNode(30)), BTNode(18))))
  False
  '''
```

Question 6. [6 MARKS]

Here is a node in an n-ary tree. (This isn't a binary tree!)

```
class Tree:
  '''Tree ADT; nodes may have any number of children'''

  def __init__(self: 'Tree',
               item: object =None, children: list =None):
    '''Create a node with item and any number of children'''

    self.item = item
    if not children:
      self.children = []
    else:
      self.children = children[:]

  def __repr__(self: 'Tree') -> str:
    '''Return representation of Tree as a string'''

    if self.children:
      return 'Tree({0}, {1})'.format(repr(self.item), repr(self.children))
    else:
      return 'Tree({})'.format(repr(self.item))
```

Write the following function to remove all nodes whose item is equal to the item of its parent. Study the examples carefully before you start.

```
  def remove_equal(self: 'Tree') -> None:
    '''Remove every child that has the same item as its parent;
    any children of a removed node n become children of an ancestor of n.

    >>> t = Tree(1, [Tree(2, [Tree(1), Tree(2)]), Tree(1)])
    >>> t.remove_equal()
    >>> repr(t)
    'Tree(1, [Tree(2, [Tree(1)])])'
    >>> t = Tree(4, [Tree(4, [Tree(6)])])
    >>> t.remove_equal()
    >>> repr(t)
    'Tree(4, [Tree(6)])'
    >>> t = Tree(4, [Tree(4, [Tree(4, [Tree(4)])])])
    >>> t.remove_equal()
    >>> repr(t)
    'Tree(4)'
    >>> t = Tree(4, [Tree(4, [Tree(4, [Tree(6), Tree(7)]), Tree(8)]), Tree(9)])
    >>> t.remove_equal()
    >>> repr(t)
    'Tree(4, [Tree(6), Tree(7), Tree(8), Tree(9)])'
    '''
```

Question 7. [9 MARKS]

Part (a) [4 MARKS] Recall the `partition` function used as part of quicksort to partition the elements of a list around a pivot value.

Partition the following list using the rightmost element as pivot. When you're finished, state the index `i` that `partition` would return.

```
[7, 7, 8, 2, 1, 10, 7]
```

Part (b) [3 MARKS] Using the middle value of a list as pivot, is it possible for quicksort to run in $O(n^2)$ time? Argue why or why not, and include an example if it helps your argument.

Part (c) [2 MARKS] Rather than use the middle or rightmost element, one might consider calculating the average value of all elements in a sublist and using that average as the pivot. Is this a good idea? Explain.

Question 8. [8 MARKS]

Part (a) [3 MARKS]

$f(n) = 3n^3$. Is f an $O(n^4)$ algorithm? Formally prove why or why not.

Part (b) [2 MARKS]

Assume a balanced binary tree. What is the big-oh time complexity for searching for an item in the tree?

Part (c) [3 MARKS]

Here's a modification of quicksort: whenever we have ten items or fewer in a sublist, we sort the sublist using selection sort rather than further recursing with quicksort. Does this change the big-oh time complexity of quicksort? Explain.

Short Python function/method descriptions:

```
__builtins__:
  input([prompt]) -> str
    Read a string from standard input; return that string with no newline. The prompt string,
    if given, is printed without a trailing newline before reading.
  len(x) -> int
    Return the length of the list, tuple, dict, or string x.
  max(iterable) -> object
  max(a, b, c, ...) -> object
    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
  min(iterable) -> object
  min(a, b, c, ...) -> object
      With a single iterable argument, return its smallest item.
      With two or more arguments, return the smallest argument.
  range([start], stop, [step]) -> list-like-object of int
    Return the integers starting with start and ending with
    stop - 1 with step specifying the amount to increment (or decrement).
    If start is not specified, the list starts at 0.  If step is not specified,
    the values are incremented by 1.

dict:
  D[k] -> object
    Return the value associated with the key k in D.
  del D[k]
    Remove D[k] from D.
  k in D -> bool
    Return True if k is a key in D and False otherwise.
  D.get(k) -> object
    Return D[k] if k in D, otherwise return None.
  D.keys() -> list-like-object of object
    Return the keys of D.
  D.values() -> list-like-object of object
    Return the values associated with the keys of D.
  D.items() -> list-like-object of tuple of (object, object)
    Return the (key, value) pairs of D, as 2-tuples.

list:
  x in L -> bool
    Return True if x is in L and False otherwise.
  L.append(x) -> NoneType
    Append x to the end of L.
  L.index(value) -> int
    Return the lowest index of value in L.
  L.insert(index, x) -> NoneType
    Insert x at position index of L.
  L.pop() -> object
    Remove and return the last item from L.
  L.remove(value) -> NoneType
    Remove the first occurrence of value from L.
  L.reverse() -> NoneType
    Reverse L *IN PLACE*.
  L.sort() -> NoneType
    Sort L in ascending order *IN PLACE*.
```

```
str:
  x in S -> bool
    Return True if x is in S and False otherwise.
  str(x) -> str
    Convert an object into its string representation, if possible.
  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.index(sub) -> int
    Like find but raises an exception if sub does not occur in S.
  S.isdigit() -> bool
    Return True if all characters in S are digits and False otherwise.
  S.lower() -> str
    Return a copy of S converted to lowercase.
  S.lstrip([chars]) -> str
    Return a copy of S with leading whitespace removed.
    If chars is given and not None, remove characters in chars from S.
  S.replace(old, new) -> str
    Return a copy of S with all occurrences of the string old replaced
    with the string new.
  S.rstrip([chars]) -> str
    Return a copy of S with trailing whitespace removed.
    If chars is given and not None, remove characters in chars from S.
  S.split([sep]) -> list of str
    Return a list of the words in S; use string sep as the separator and
    any whitespace string if sep is not specified.
  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.
```