PLEASE HAND IN

UNIVERSITY OF TORONTO

Faculty of Arts and Science

April 2015 Examinations

CSC 148H1S

Duration — 3 hours

PLEASE HAND IN

Allowed aids: one 8.5"x11" aid sheet (both sides)

Student Number: ___________

Last Name: ___________________________

First Name: ___________________________

Do **not** *turn this page until you have received the signal to start.*
(In the meantime, please fill out the identification section above, and read the instructions below.)

This exam consists of 8 questions on 20 pages (including this one). *When you receive the signal to start, please make sure that your copy of the exam is complete.*

Please answer questions in the space provided. You will earn 20% for any question you leave blank or write "I cannot answer this question," on. You may earn substantial part marks for writing down the outline of a solution and indicating which steps are missing.

You must achieve 40% of the marks on this final exam, or 29 out of 73, to pass this course.

Write your student number at the bottom of pages 2-20 of this exam.

There is a Python API on the last page of this exam that you may tear off for reference.

# 1:	______/ 9
# 2:	______/ 8
# 3:	______/11
# 4:	______/10
# 5:	______/10
# 6:	______/12
# 7:	______/ 5
# 8:	______/ 8
TOTAL:	______/73

Good Luck!

Question 1. [9 MARKS]

Users of music software like Spotify and Google Play Music can create playlists, and can share them with other users. In this question you will write two classes for implementing playlists. Doctest examples and method descriptions are not required, but you must write a type contract for each method.

Below, write a class called Tune that satisfies the following requirements:

- A tune has a title, artist, and length (in seconds). Client code is allowed to access these instance variables directly.
- Add further further instance variables as needed, to support the methods in the class. However, use an underscore to indicate that client code is not intended to access these directly.
- A tune provides a method called **play**, for recording that a particular user (identified by their email address) has played the tune.
- A tune provides a method called **plays_by**, for reporting the number of times that a particular user has played the tune.

```
class Tune:
```

On this page, write a class called Playlist that satisfies the following requirements:

- A playlist has an initially empty sequence of tunes. Their order matters.
- A playlist provides a method called add_tune that adds a tune to the end of the sequence of tunes in the playlist, even if that means repeating one that is already there.
- A playlist provides a method called play, for recording the fact that a particular user played the first n tracks on the playlist. If the playlist has fewer than n tracks, it records that the user has played all of the tracks.
- A playlist provides a method called total_time_played, for reporting the number of seconds of tunes from the playlist have been played by a particular user, including tunes that may have been played multiple times.

```
class Playlist:
```

Question 2. [8 MARKS]

Recall the definition of the class **BTNode**:

```
class BTNode:
    '''Binary Tree node.'''

    def __init__(self, data, left=None, right=None):
        ''' (BTNode, object, BTNode, BTNode) -> NoneType

        Create BTNode (self) with data and children left and right.
        '''
        self.data, self.left, self.right = data, left, right
```

Part (a) [2 MARKS]

Read the doctest examples in function occurs below. Draw the tree whose root is referred to by `whole`.

```
def occurs(root, s):
    ''' (BTNode or None, str) -> bool

    Return whether or not s equals a sequence of characters
    along some path from the root to a leaf, inclusive, and in
    that order. The empty str ("") is considered to occur in
    the empty tree, denoted None.

    Assume each node in the tree rooted at root contains a str of length 1.

    >>> left = BTNode('b', None, BTNode('d', BTNode('e'), None))
    >>> right = BTNode('c', BTNode('e'), BTNode('f', BTNode('h'), BTNode('i')))
    >>> whole = BTNode('a', left, right)
    >>> occurs(whole, 'acfh')
    True
    >>> occurs(whole, 'ace')
    True
    >>> occurs(whole, 'bde')
    False
    '''
```

Part (b) [6 MARKS]

On the next page, write the body of function occurs.

Write your function body here.

Question 3. [11 MARKS]

A **perfect binary tree** is one in which (a) every non-leaf node has exactly two children, and (b) all leaves occur at the same level. Notice that a tree consisting of a single node is a perfect binary tree of height 1.

An empty tree, represented by None, has is a perfect binary tree of height 0.

Part (a) [3 MARKS]

Recall that we defined height of a tree in such a way that a tree containing just a root node has height one. Draw a perfect binary tree of height 3.

Draw a binary tree of height 4 this time (not 3) that is not perfect because it satisfies condition (a) but not condition (b).

Draw a binary tree of height 4 that is not perfect because it satisfies condition (b) but not condition (a).

Part (b) [8 MARKS]

Recall the definition of the class **BTNode**:

```
class BTNode:
    '''Binary Tree node.'''

    def __init__(self, data, left=None, right=None):
        ''' (BTNode, object, BTNode, BTNode) -> NoneType

        Create BTNode (self) with data and children left and right.
        '''
        self.data, self.left, self.right = data, left, right
```

Write the body of function **TPBT**.

```
def TPBT(root):
    ''' (BTNode or None) -> (int, bool)

    Return a tuple containing: (1) the height of the tallest perfect binary
    tree within the tree rooted at root, and (2) whether or not that tallest
    perfect binary tree occurs at the root itself.
    '''
```

Question 4. [10 MARKS]

Read the declaration of class Tree and the docstring for function **unique_paths**. Then implement (write the body for) function **unique_paths** on the next page.

Hint: You may conclude there are unique paths in a tree if you traverse (visit) every node without finding a node that has been visited twice. A set is a convenient data structure for recording objects that have been seen.

```
class Tree:
    def __init__(self, value=None, children=None):
        ''' (Tree, object, list of Tree) -> NoneType

        Create Tree(self) with content value and 0 or more Tree children.
        '''
        self.value = value
        # copy children if not None
        self.children = children.copy() if children else []


def unique_paths(t):
    ''' (Tree) -> bool

    Return whether there is a unique path from t to each
    of its descendents.

    Assume that two Trees are the same if they have the same
    memory address, that is id(t1) == id(t2)

    >>> t1 = Tree(1)
    >>> t2 = Tree(2)
    >>> t3 = Tree(3, [t1, t2])
    >>> unique_path(t3)
    True
    >>> t4 = Tree(4, [t3, t1])
    >>> unique_path(t4)
    False
    >>> t3.children.append(t3)
    >>> unique_path(t3)
    False
    '''
```

Part (a) [2 MARKS]

Draw the tree rooted at **t4**, as it is after all the doctest code has been executed.

Part (b) [8 MARKS]

Write the body of unique_paths below:

Question 5. [10 MARKS]

Recall the _init_ methods for classes **LLNode** and **LinkedList** that we saw in class:

```
class LLNode:
    def __init__(self, value, nxt=None):
        ''' (LLNode, object, LLNode) -> NoneType

        Create LLNode (self) with data value and successor nxt.
        '''
        self.value, self.nxt = value, nxt

class LinkedList:
    def __init__(self):
        ''' (LinkedList) -> NoneType

        Create an empty linked list.
        '''
        self.front, self.back = None, None
        self.size = 0
```

Here is some room for rough work:

Part (a) [5 MARKS]

Suppose we have LinkedList lnk and that variable p refers to a node in it. Update the linked list to reverse the order of the two nodes after the one that p refers to. If there are not two nodes after the node p refers to, do nothing. You must solve this by updating nxt, not by updating value.

Part (b) [5 MARKS]

Suppose we have a LinkedList lnk, and that a variable p refers to a node in it. Update the linked list so that there is a second copy of the node after the one that p refers to. The new node should be adjacent to the node it duplicates. If there is no node after p, do nothing.

Question 6. [12 MARKS]

LeakyQueue is a subclass of **Queue** that implements a First In Usually First Out (FIUFO) Queue. **LeakyQueue** has one additional method **defer(from_value, to_value)**. What **defer** does is find the first occurrence of **from_value** and replace it with the first occurrence of **to_value** that follows it, removing **to_value** from its previous position in the queue. If there is no instance of **from_value** earlier than an instance of **to_value**, then this method does nothing.

Read over the implementation of class **Queue** below.

```
class LLNode:
    def __init__(self, value, nxt=None):
        ''' (LLNode, object, LLNode) -> NoneType

        Create LLNode (self) with data value and successor nxt.
        '''
        self.value, self.nxt = value, nxt


class Queue:
    def __init__(self):
        ''' (Queue) -> NoneType

        Create and initialize new queue self.
        '''
        self._front = self._back = None

    def enqueue(self, o):
        ''' (Queue, object) -> NoneType

        Add o at the back of this queue.
        '''
        new_node = LLNode(o)
        if self._back:
            self._back.nxt = new_node
            self._back = new_node
        else:
            self._back = self._front = new_node

    def dequeue(self):
        ''' (Queue) -> object

        Remove and return front object from self.
        '''
        new_value = self._front.value
        self._front = self._front.nxt
        return new_value

    def is_empty(self):
        ''' (Queue) -> bool

        Return True queue self is empty,
        False otherwise.
        '''
        return self._front == None
```

Part (a) [2 MARKS]

Complete the doctest example below to show what **defer** does. Use only the methods available in **LeakyQueue** — do not access instance variables.

```
class LeakyQueue(Queue):
   def defer(self, from_value, to_value):
      ''' (Leakyqueue, object, object) -> NoneType

      Find first node containing from_value and replace it with the first node
      after it that contains to_value. If there is no node with from_value
      occurring before a node with to_value, do nothing.

      >>> lq = LeakyQueue()
      >>> lq.enqueue(1)
      >>> lq.enqueue(2)
      >>> lq.enqueue(3)
      >>> lq.enqueue(4)










   '''
```

Part (b) [10 MARKS]

Now implement the method defer below. You don't need to repeat the docstring here.

```
    def defer(self, from_value, to_value):
```

Question 7. [5 MARKS]

Consider a state of the game Subtract a Square in which the current player is 'p1' and the current value is 7. Draw a tree diagram to show all the game states that the minimax algorithm would consider. We have drawn the root state for you. On the left side of each state show the minimax score for player 'p1' and on the right side show the minimax score for player 'p2'.

$\boxed{\text{p1 : 7}}$

Question 8. [8 MARKS]

From the list, circle the big-oh expression that gives the best upper bound for each code fragment, and briefly explain your choice.

Part (a) [2 MARKS]

```
sum, i = 0, 1
while 2 * i < n
    sum = sum + i
    i = 2 * i
```

$\mathcal{O}(1)$ $\mathcal{O}(\log_2 n)$ $\mathcal{O}(\sqrt{n})$ $\mathcal{O}(n)$ $\mathcal{O}(n \log_2 n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(n^3)$ $\mathcal{O}(2^n)$

Part (b) [2 MARKS]

```
i, j, sum = 0, 1, 0
while i < n**2:
    while j < n:
        sum = sum + i
        j = 2 * j
    i = i + n
```

$\mathcal{O}(1)$ $\mathcal{O}(\log_2 n)$ $\mathcal{O}(\sqrt{n})$ $\mathcal{O}(n)$ $\mathcal{O}(n \log_2 n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(n^3)$ $\mathcal{O}(2^n)$

Part (c) [2 MARKS]

```
i, sum = 0, 0
while  (i // 2) < n:
    if i % 2 == 0:
        for j in range(n):
            sum = sum + j
    else:
        for j in range(n**2):
            sum = sum + i
    i = i + 1
```

$\mathcal{O}(1)$ $\mathcal{O}(\log_2 n)$ $\mathcal{O}(\sqrt{n})$ $\mathcal{O}(n)$ $\mathcal{O}(n \log_2 n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(n^3)$ $\mathcal{O}(2^n)$

Part (d) [2 MARKS]

```
i, sum = 0, 0
while i < n * 2:
    sum = sum + i
    i = i + 1
```

$\mathcal{O}(1)$ $\mathcal{O}(\log_2 n)$ $\mathcal{O}(\sqrt{n})$ $\mathcal{O}(n)$ $\mathcal{O}(n \log_2 n)$ $\mathcal{O}(n^2)$ $\mathcal{O}(n^3)$ $\mathcal{O}(2^n)$

This page has been left intentionally (mostly) blank, in case you need space.

Total Marks = 73

YOU CAN TEAR THIS SHEET OFF IF YOU LIKE.

Short Python function/method descriptions:

```
__builtins__:
  len(x) -> integer
    Return the length of the list, tuple, dict, or string x.
  max(L) -> value
    Return the largest value in L.
  min(L) -> value
    Return the smallest value in L.
  range([start], stop, [step]) -> list of integers
    Return a list containing 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.
  sum(L) -> number
    Returns the sum of the numbers in L.

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

float:
  float(x) -> floating point number
    Convert a string or number to a floating point number, if
    possible.

int:
  int(x) -> integer
    Convert a string or number to an integer, if possible. A floating
    point argument will be truncated towards zero.

list:
  x in L -> boolean
    Return True if x is in L and False otherwise.
  L.append(x)
    Append x to the end of list L.
  L1.extend(L2)
    Append the items in list L2 to the end of list L1.
  L.index(value) -> integer
    Return the lowest index of value in L.
  L.insert(index, x)
    Insert x at position index.
  L.pop()
    Remove and return the last item from L.
  L.remove(value)
    Remove the first occurrence of value from L.
```

```
  L.sort()
    Sort the list in ascending order.

Module random:
  randint(a, b)
    Return random integer in range [a, b], including both end points.

str:
  x in s -> boolean
    Return True if x is in s and False otherwise.
  str(x) -> string
    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]) -> integer
    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.split([sep]) -> list of strings
    Return a list of the words in S, using string sep as the separator
    and any whitespace string if sep is not specified.

set:
  {1, 2, 3, 1, 3} -> {1, 2, 3}
  s.add(...)
    Add an element to a set
  set()
    Create a new empty set object
  x in s
    True iff x is an element of s

list comprehension:
   [<expression with x> for x in <list or other iterable>]

functional if:
   <expression 1> if <boolean condition> else <expression 2>
   -> <expression 1> if the boolean condition is True,
      otherwise <expression 2>
```