PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

April 2018 Examinations

CSC 148H1S
Duration — 3 hours
No aids allowed.

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 7 questions on 16 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. If you need additional space, clearly indicate on the question page where to find your answer.

You will earn 20% for any question you leave blank or write "I cannot answer this question" on. You might earn substantial part marks for writing down the outline of a solution and indicating which steps are missing. You may use helper functions, if you wish, provided you define them on this paper.

You must achieve 40% of the marks on this final exam to pass this course.

There is a Python API at the end of this exam.

1: _____/12

2: _____/10

3: _____/ 6

4: _____/ 8

5: _____/ 5

6: _____/ 5

7: _____/ 8

TOTAL: _____/54

Good Luck!

Question 1. [12 MARKS]

Write the output printed by the following programs.

Part (a) [3 MARKS]

```
def foo(k: list) -> None:
    k.append(k.append(1))

if __name__ == '__main__':
    q = [0]
    foo(q)
    print(q)
```

Please write the output in the space below:

Part (b) [3 MARKS]

```
def bar(list_: list) -> None:
    bigger_list = []
    for item in list_:
        bigger_list.append(item + 100)
    list_, bigger_list = bigger_list, list_


if __name__ == '__main__':
    list_ = [1, 2, 3]
    bar(list_)
    print(list_)
```

Please write the output in the space below:

Part (c) [3 MARKS]

Circle the complexity class that best characterizes the run-time of function `big_oh` if `list_` is a list with n elements. Write a brief explanation of your conclusion.

$\Theta(1)$ $\Theta(\lg n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(2^n)$

```
def method(list_: list) -> None:
    for i in range(len(list_)):
        print(list_[i])

def big_oh(list_: list) -> None:
    for i in list_:
        method(list_)
```

Part (d) [3 MARKS]

Circle the complexity class that best characterizes function `mystery` given the run times listed for various inputs. Write a brief explanation of your conclusion.

$\Theta(1)$ $\Theta(\lg n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(2^n)$

function call	run time
mystery(2)	0.01 seconds
mystery(2^2)	0.02 seconds
mystery(2^3)	0.03 seconds
mystery(2^4)	0.04 seconds
mystery(2^5)	0.05 seconds
mystery(2^6)	0.06 seconds
mystery(2^7)	0.07 seconds

Question 2. [10 MARKS]

Priority queues remove items based on a priority scheme, and not necessarily according to the order they were added. If queue items are all comparable, we can remove them smallest first or largest first. Recall the public interface for standard FIFO class `Queue`:

`add(object_)`: Add `object_` at the back of the queue.

`remove()`: Remove and return the object at the front of the queue.

`is_empty()`: Return whether this queue is empty or not.

Extend class `Queue` with a subclass `PriorityQueue` that adds two new methods:

`remove_largest()`: Remove and return an element that is greater than or equal to all the other elements. Assume the queue is not empty and that all elements are comparable.

`remove_smallest()`: Remove and return an element that is less than or equal to all other elements. Assume the queue is not empty and that all elements are comparable.

Declare subclass `PriorityQueue`, along with an appropriate docstring, and define and document any necessary methods. You may not import any other modules. Documentation includes type annotations, and a method description, but no examples are required.

Hint: You can't assume anything about the private, underlying, implementation of class `Queue`, for example you don't know whether it uses a list, a linked list, or a dictionary for its internal storage. However, you can certainly combine the three public methods of `Queue` with other built-in Python data types to implement your methods.

```
from csc148_queue import Queue
```

This page is left mainly blank, for things that don't fit elsewhere.

Question 3. [6 MARKS]

Read the declaration of class `LinkedListNode` and its `__str__` method below.

On the next page, we had written a solution to partition that was working, but unfortunately our solution got put through the laundry, resulting in some unreadable parts. Replace the underlines with Python code that makes it work. Do not create any new `LinkedListNodes` or change any `value` attributes.

Hint: Read the comments carefully, and draw diagrams.

LinkedListNode class

```
from typing import Union, Tuple


class LinkedListNode:
    """
    Node to be used in linked list

    === Attributes ===
    next_ - successor to this LinkedListNode
    value - data represented by this LinkedListNode
    """
    next_: Union["LinkedListNode", None]

    def __init__(self, value: object,
                 next_: Union["LinkedListNode", None]=None) -> None:
        """
        Create LinkedListNode self with data value and successor next

        >>> LinkedListNode(5).value
        5
        >>> LinkedListNode(5).next_ is None
        True
        """
        self.value, self.next_ = value, next_

    def __str__(self) -> str:
        """
        Return a user-friendly representation of this LinkedListNode.

        >>> n = LinkedListNode(5, LinkedListNode(7))
        >>> print(n)
        5 -> 7 ->|
        """
        self_str = "{} ->".format(self.value)
        current_node = self.next_
        while current_node is not None:
            self_str += " {} ->".format(current_node.value)
            current_node = current_node.next_
        return self_str + "|"
```

```
def partition(lnk: LinkedListNode, pivot: LinkedListNode) -> Tuple[LinkedListNode, LinkedListNode]:
    """
    Use pivot to return (lnk1, lnk2), where lnk1 is the first in a chain of all nodes from lnk that
    have values smaller than the value of pivot, and lnk2 is the first in a chain of all nodes from
    lnk that have values that are not smaller than the value of pivot, but are not pivot itself.
    Of course, lnk1 or lnk2 are set to None if there are no nodes that belong in them.

    Assume lnk is the first in a non-empty chain of nodes containing integer values, one of which is pivot.

    >>> lnk = LinkedListNode(0, LinkedListNode(1, LinkedListNode(2)))
    >>> lnk = LinkedListNode(-2, LinkedListNode(-1, lnk))
    >>> pivot = lnk.next_.next_
    >>> (lnk1, lnk2) = partition(lnk, pivot)
    >>> print(lnk1)
    -1 -> -2 ->|
    >>> print(lnk2)
    2 -> 1 ->|
    """

    # start lnk1 and lnk2 as empty

    lnk1 = lnk2 = _________________

    # start with lnk

    current_node = _________________

    # walk along lnk's nodes until the end

    while ____________________________________________________________________________________________:


        # save reference to following node

        next_node = _____________________________________


        # if we have a node with value less than pivot's value add it to nodes headed by lnk1
        if current_node.value < pivot.value:

            _____________________________, lnk1 = lnk1, _____________________________


        # if we have a node, other than pivot, with value at least as great as pivot's value, add it to
        # nodes headed by lnk2
        elif current_node is not pivot:

            _____________________________, lnk2 = lnk2, _____________________________


        # update node to visit

        _____________________________ = _____________________________

    return (lnk1, lnk2)
```

Question 4. [8 MARKS]

Read the declaration of class Tree on page 14. Then implement function odd_average below. Hint: You may find helper functions useful.

```
def odd_average(t: Tree) -> float:
    """
    Return the average of the odd values in t, 0 if none are odd.

    Assume all nodes have integer values.

    >>> t2 = Tree(2, [Tree(5), Tree(6)])
    >>> t3 = Tree(3, [Tree(7)])
    >>> t = Tree(1, [t2, t3, Tree(4)])
    >>> odd_average(t)
    4.0
    >>> odd_average(Tree(2))
    0.0
    """
```

This page is left mainly blank, for things that don't fit elsewhere.

Question 5. [5 MARKS]

Read the declaration of BTNode and its __repr__ method on page 14. Use them to implement function mutate below — do not use any other methods or functions associated with BTNode.

Hint: This function mutates (changes) the tree, so a list comprehension is probably not a suitable tool. You may want to draw a diagram of BTNode t in the docstring example.

```
def mutate(t: Union[BTNode, None], depth: int) -> None:
    """
    Change tree t by swapping the children of each node that has data
    that is less than its depth.  If the data is greater than or
    equal to its depth, replace the data by the sum of the data and
    the depth.

    Assume all data are ints.

    >>> t = BTNode(0, BTNode(0, BTNode(2), BTNode(3)), BTNode(1))
    >>> mutate(t, 0)
    >>> repr(t)
    'BTNode(0, BTNode(0, BTNode(5, None, None), BTNode(4, None, None)), BTNode(2, None, None))'
    """
```

This page is left mainly blank, for things that don't fit elsewhere.

Question 6. [5 MARKS]

Implement function freeze_list below. You may not import any modules; in particular you may not use deepcopy().

```
def freeze_list(list_: list) -> list:
    """ Return a new list equivalent to list_ but where
    every list contained in list_ is replaced by a new,
    equivalent, list object.

    >>> my_list = [1, [2, 3, [4]], 5]
    >>> new_list = freeze_list(my_list)
    >>> my_list == new_list
    True
    >>> my_list is new_list
    False
    >>> my_list[1] is new_list[1]
    False
    >>> my_list[1][2] is new_list[1][2]
    False
    """
```

Question 7. [8 MARKS]

Our `hash_table.py` example developed in class used a python list `self.table` to store small sub-lists (called buckets) containing tuples of the form `(key, value)`. We inserted new `(key, value)` tuples by calculating `hash(key) % len(self.table)` to find the index of the correct sub-list, and then appended the tuple to the sub-list. If the number of `(key, value)` tuples exceeds 0.7 times the length of `self.table`, we double the length of `self.table` and re-insert all `(key, value)` tuples.

Assume that a hash table begins with `self.table` having length 2. When it's empty, `self.table` it looks like this:

```
[[], []]
```

Suppose our `hash` function produces these remarkably short results for the following strings:

```
>>> hash("how")
1
>>> hash("now")
6
>>> hash("brown")
3
>>> hash("cow")
9
```

First, we insert key/value tuples `("how", "adverb")` and `("now", "adverb")`, so `self.table` looks like this:

```
[[], [("how", "adverb")], [("now", "adverb")], []]
```

In the space below, show what `self.table` will look like if we next insert `("brown", "adjective")` and `("cow", "noun")`.

Total Marks = 54

Tree class

```
class Tree:
    """
    A bare-bones Tree ADT that identifies the root with the entire tree.

    === Attributes ===
    value - value  of root node
    children - root nodes of children
    """
    value: object
    children: List["Tree"]

    def __init__(self, value: object, children: List["Tree"]=None) -> None:
        """
        Create Tree self with content value and 0 or more children
        """
        self.value = value
        # copy children if not None
        self.children = children[:] if children is not None else []
```

BTNode class

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

    === Attributes ===
    data - data this node represents
    left - left child
    right - right child
    """
    data: object
    left: Union["BTNode", None]
    right: Union["BTNode", None]

    def __init__(self, data: object,
                 left: Union["BTNode", None]=None,
                 right: Union["BTNode", None]=None) -> None:
        """ Create BTNode (self) with data and children left and right.

        An empty BTNode is represented by None.
        """
        self.data, self.left, self.right = data, left, right

    def __repr__(self) -> str:
        """ Represent BTNode (self) as a string that can be evaluated to
        produce an equivalent BTNode.

        >>> BTNode(1, BTNode(2), BTNode(3))
        BTNode(1, BTNode(2, None, None), BTNode(3, None, None))
        """
        return 'BTNode({}, {}, {})'.format(self.data, repr(self.left), repr(self.right))
```

Short Python function/method descriptions, and classes

```
__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) -> None
    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.pop(i)
    Remove and return L[i]
  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
  {1, 2, 3}.union({2, 4}) -> {1, 2, 3, 4}
  {1, 2, 3}.intersection({2, 4}) -> {2}
  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>
```