UNIVERSITY OF TORONTO
Faculty of Arts and Science

Midterm 1

CSC148H1F

October 15, 2014 (**50 min.**)

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

Name: Rui Qiu

Student Number: 999292509

Please read the following guidelines carefully!

- The last page just has an aid sheet; detach this for your convenience during the exam.
- Please write your name on the front **and back** of the exam. The latter is to help us return the exams.
- This examination has **4** questions. There are a total of **10 pages, DOUBLE-SIDED**.
- Any question you leave blank or clearly cross out your work and write "I don't know" is worth **10% of the marks**.

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.

Good luck!

1. The following questions test your understanding of the terminology and concepts from the first four weeks of this course. You may answer in either point form or full sentences; **you do not need to write much to get full marks!**

   (a) [2] Explain the difference between a **stack** and a **queue**.

2

A stack is an ADT that whatever enters last, exits first.

A queue is an ADT that whatever enters first, exits first.

   (b) [2] Explain the difference between an **abstract data type (ADT)** and an **implementation** of that abstract data type. Give an example in your response.

05

The Stack is an ADT, OK and the implementations of it are the methods we defined for it.

E.g. def is_empty(self):
    return len(items)==0

Those implementations can be called, and can be defined in many different ways.

Solution: An ADT describes how it stores data, and what operations (methods) one can perform on this data. This is abstract b/c there is no code required to understand ADT. An implementation of an ADT is the (Python) code that actually defines a class that behaves as described by the ADT.

   (c) [3] Suppose we want to store data using a list, and in our application we'll be adding and removing elements mainly at the **front** of the list. Which of an array or a linked list should we use to store this data? Give a detailed explanation for your reasoning.

0

I think a ~~stack~~ queue would be needed. Since we only need to care about the 'front' end, whatever enters first gets ~~o~~ out first. → not a queue

Specifically, we can use enqueue to add elements at the front, and use dequeue to ~~o~~ remove the front element. OK

Sol: A linked list is needed.
Removing & deleting from the front involves changing just one or two links. (self.first, and any new nodes you would create). and so can be done in time that is independent of the size of the list.
However, inserting/deleting from the front of an array requires that you shift all other items over (to make sure items are stored in consecutive locations in memory), and this requires time proportional to the length of the array

(d) [2] The following classes are defined using **inheritance**.

```
1   class A:
2       def __init__(self, x):
3           self.x = x
4
5       def show(self):
6           print(self.x)
7
8   class B(A):
9       def __init__(self, x, y):
10          A.__init__(self, x)
11          self.y = y
12
13      def show(self):
14          print("I'm a B!")
15
16      def noshow(self):
17          A.show(self)
18          print("shhh")
```

1.5

Assume that we've loaded this source code into Wing, and run the following commands successfully.

```
1   >>> a = A('Hi')
2   >>> b = B('Bye', 'Hello')
```

Clearly state what happens/is output when each of the following commands is run. (No explanations necessary.)

```
1   >>> a.show()
```

Hi

```
1   >>> b.show()
```

I'm a B!

```
1   >>> a.noshow()
```

AttributeError

```
1   >>> b.noshow()
```

'Bye', 'Hello'

~~shhh~~

shhh

2. [6] Implement the following function. Of course, you may **only use methods from the Stack ADT.**

```
class StackException(Exception):
    pass

def remove_nth(s, n):
    """ (Stack, int) -> object
    Remove and return the n-th item from the top of s.
    Do nothing to the other items in s.
    If s has fewer than n elements, raise a StackException.

    Note: remove_nth(s, 1) does the same thing has s.pop(),
    and remove_nth(s, 2) is the same as remove_second(s) from the exercise.

    You may assume that if you call pop() on an empty stack,
    an EmptyStackError is raised. (This may or may not be helpful,
    depending on how you solve this problem.)

    >>> s = Stack()
    >>> s.push(1)
    >>> s.push(2)
    >>> s.push(3)
    >>> s.push(4) # After this step, s contains 1,2,3,4
    >>> remove_nth(s, 3)
    2 # s now contains 1,3,4
    """
    # YOUR CODE GOES HERE
```

~~#add an~~ ~~if loop~~ ~~If n~~

~~count = 0~~
~~temp = Stack()~~
~~while count != n-1~~
~~e = s.pop()~~
~~temp.push(e)~~
~~n = n+1~~

e2 = temp.pop()
s.push(e2)
count2 -= 1
return s

```
count = 0
temp = Stack()
while count != n-1:
    if s.is_empty():
        raise StackException
    else:
        e = s.pop()
        temp.push(e)
        n += 1
if s.is_empty():
    raise StackException
else:
    s.pop()
    count2 = n-1
    while count2 != 0:
```

count?

3. [5] This question refers to the LinkedList implementation found on the aid sheet. Consider the following **incorrect method** for inserting a new item at the second position in a linked list. (Note: this method would be included in the body of the `LinkedList` class.)

```
1  def insert_second(self, item):
2      """ (LinkedList, object) -> NoneType
3      Insert item at the second position of this list.
4      Raise IndexError if this list is empty.
5      >>> lst = LinkedList([1, 2, 3])  # [1 -> 2 -> 3]
6      >>> lst.insert_second(10)        # [1 -> 10 -> 2 -> 3]
7      """
8      if self.first is None:
9          raise IndexError
10     else:
11         new_node = Node(item)
12         self.first.next = new_node
```

Explain what goes wrong when you try to use this method on a linked list `[1 -> 2 -> 3]` to insert 10 at the second position.

Though 10 is linked with 1,
10 has not set up a link with 2. (previous element at second ~~index~~ position).

✓

Then, rewrite the code of this method to fix the problem and satisfy its docstring. You may not use any `LinkedList` methods here; access the class attributes directly!

```
1  def insert_second(self, item):
2      # YOUR CODE GOES HERE
       if self.first is None:
           raise IndexError

       elif len(items)==1:
           self.first.next=Node(item)

       else:
           curr = self.first
           curr.next = Node(item)
           for item in items[1:]:
               curr.next.next = curr.next

   else:
       new_node = Node(item)
       new_node.next = self.first.next
       self.first.next = new_node
```

4/5

4. [5] A very common operation to do on lists is filter them according to some property, e.g. "the videos about recursion" or "the students who are in first-year".

   Python has a built-in `filter` function that does this for regular lists; your task in this question is to implement a simpler version for the node-based linked list. (Again, the base code is found on the cheat sheet.)

   Your function should create new linked list – the original linked list should remain unchanged! In other words, this is a *non-mutating* function.

   You **may not** use built-in Python lists, nor any `LinkedList` methods we developed in class; only use the linked list and node attributes. Exception: we have used the constructor just once to help you get started. You should not create any more linked lists.

```
def filter_positive(lst):
    """ (LinkedList of int) -> LinkedList of int
    Return a new LinkedList whose items are
    the ones in lst that have value > 0.
    The items must appear in the *same order*
    they do in lst.

    >>> lst = LinkedList([3, -10 , 4, 0])  # [3 -> -10 -> 4 -> 0]
    >>> pos = filter_positive(lst)         # pos is [3 -> 4]
    """
    # Create a new, empty linked list. Hint: return new_lst at the end.
    new_lst = LinkedList([])

    # YOUR CODE GOES HERE
    lst2 = []
    for item in l:
        if item > 0:
            lst2.append(item)
    if len(lst2) == 0:
        return new_lst
    elif len(lst2) == 1:
        new_lst.first = lst2[0]
        new_lst.next = None
        return new_lst

    else:
        n = 1
        new_lst.first = lst2[0]
        curr = new_lst.first
        while n != len(lst2):
            curr.next = lst[n]
            curr = curr.next
            n += 1

    return new_lst
```

l = lst.items

what is l?

Need to create a new Node.

O(n)

```
curr = lst.first
new_curr = new_list.first
while curr is not None:
    if curr.item > 0:
        if new_curr is None:
            new_list.first = Node(curr.item)
            new_curr = new_list.first
        else:
            new_curr.next = Node(curr.item)
            new_curr = new_list.next
    curr = curr.next
return new_list
```

Bonus Question [2]

Warning: this is a difficult question, and will be marked harshly. Only attempt it if you have finished all of the other questions!

One of the major shortcomings of our linked list class is that it's only possible to move forwards at a node, but not backwards. A **doubly-linked list** is a linked list of nodes where the nodes store a reference to both the next node and previous node in the list. Storing the "previous" links enables the extra flexibility of moving backwards and forwards through a list, at the cost of extra memory.

Write the analogous classes and constructors for a node-based doubly-linked list to the standard linked list code found on the aid sheet. Call your classes `DoubleNode` and `DoublyLinkedList`. Note that the parameters to the constructors for each class (an object and a list, respectively) must stay the same. You do not need to implement any methods other than the constructors.

Note: The core of this question is converting a built-in Python list into a doubly-linked list.

I don't know.