**UNIVERSITY OF TORONTO**
**Faculty of Arts and Science**

# Midterm 1 - SOLUTIONS

**CSC148H1F**

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

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

Name:

Student Number:

**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**.

   In a stack, you remove items in *reverse order* you insert them, so the first item removed is the most recently-added. In a queue, the first item removed is the one that was inserted first.
   (Note that this description does not depend on the implementation of either the stack or the queue.)

   (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.

   An ADT describes how it stores data, and what operations (*methods*) one can perform on this data. This is abstract because there is no code required to understand an ADT. An **implementation** of an ADT is the (Python) code that actually defines a **class that behaves as described by the ADT**.
   Incorrect responses: an implementation is *not* the methods associated with the data (this is part of the ADT definition itself), nor is it an application of an ADT to a particular problem domain (doing so in code already relies on an implementation of the ADT). Finally, an implementation is not an object instance of a particular class representing an ADT - it is the code of the class definition itself.

   (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.

   A linked list is much more efficient in this case; removing and 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")
```

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` is printed.

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

`I'm a B!` is printed.

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

An error is raised (a has no attribute noshow)

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

`Bye` and `shhh` are printed (on separate lines).

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):
    # SOLUTION

    # This check is optional
    if n <= 0:
        return

    top_items = Stack()
    # Get the top n items from the stack
    for count in range(n):
        try:
            tmp = s.pop()
        except EmptyStackError:
            raise StackException

        top_items.push(tmp)

    # Remove the top item from the new stack (which is the nth item of s)
    top_items.pop()

    # Put items back onto the stack
    while not top_items.is_empty():
        s.push(top_items.pop())
```

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.)

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

**Solution:** When we create `new_node`, its `next` attribute is set to `None`. So setting `self.first.next = new_node` correctly inserts the new item into the second position, but all items after are lost! So `[1 -> 2 -> 3]` would become the list `[1 -> 10]`.

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!

```
def insert_second(self, item):
    # SOLUTION
    if self.first is None:
        raise IndexError
    else:
        new_node = Node(item)
        new_node.next = self.first.next
        self.first.next = new_node
```

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([])

    # SOLUTION
    curr = lst.first
    new_curr = new_list.first
    while curr is not None:
        if curr.item > 0:
            # Insert a new node into new_lst
            if new_curr is None:
                new_lst.first = Node(curr.item)
                new_curr = new_list.first
            else:
                new_curr.next = Node(curr.item)
                new_curr = new_curr.next
        curr = curr.next

    return new_lst
```

**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.

Use this page for rough work.

Use this page for rough work.

Name:

| | Q1 | Q2 | Q3 | Q4 | Total | Bonus |
|---|---|---|---|---|---|---|
| Grade | | | | | | |
| Out Of | 9 | 6 | 5 | 5 | 25 | 2 |