UNIVERSITY OF TORONTO
Faculty of Arts and Science

Final Examination

CSC148H1F

December 15, 2014 (**3 hours**)

Examination Aids: Cheat sheet on back, detachable!

Name:

Student Number:

Please read the following guidelines carefully!

- This examination has **7** questions. There are a total of **14 pages, DOUBLE-SIDED**.
- You may use helper functions unless explicitly told not to.
- Any question you leave blank or clearly cross out your work and write "I don't know" is worth **10% of the marks**.
- You must earn a grade of **at least 40% to pass this course**.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Total	Bonus
Grade									
Out Of	15	8	6	9	6	8	9	61	3

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.

It's been a real pleasure teaching you this term.

Good luck!

1. Answer each of the following using brief sentences or point-form. Diagrams and code examples are encouraged.

   (a) **[2]** Explain the difference between an **abstract data type** (ADT) and an **implementation** of that data type.

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

   (c) **[2]** Define the **BST property** and the **Heap property**.

   (d) **[2]** Draw a binary tree that satisfies the **heap property** but is not a heap.

   (e) **[2]** True or false: quicksort always runs in $O(n \log n)$ time. Justify your answer.

(f) **[1]** We saw in lecture that radix sort was particularly efficient for sorting integers. What is a limitation of radix sort? (Why don't we just use radix sort for all our sorting needs?)

(g) **[2]** What is the problem with this code? (You do not need to fix the problem, just explain what it is.)

```
1  def size(tree):
2      """ (Tree) -> int
3      Return the number of nodes in tree.
4      """
5      if tree.is_empty():
6          return 0
7      else:
8          count = 1
9          for subtree in tree.subtrees:
10             subtree.size()       count += subtree.size()
11         return count
```

In recursive part, after getting sizes of each subtrees, it forgot to accumulates them to "count".

(h) **[2]** What is the problem with this code? (You do not need to fix the problem, just explain what it is.)

```
1  def clear_list(lst):
2      """ (LinkedList) -> NoneType
3      Turn lst into an empty linked list.
4      (That is, remove all items from lst.)
5      """
6      lst = LinkedList([])   lst = LinkedList()
```

This will construct a new LL with a node whose content is "[]", what we do is simply re-initialize a new empty LinkedList object to lst.

2. **[8]** Recall from a past exercise the `Car` class, which starts with some fuel and a fuel efficiency, and can move to different locations. A `TeleportCar` is a special car which behaves as follows:

   - It always starts with 20 units of fuel, and fuel efficiency 2.
   - It starts with some non-negative number of teleportation charges.
   - When it tries to move to a new location, it first tries to move like a regular car. If that fails (and a `NotEnoughFuelError` is raised), it then spends a teleportation charge to move, using no fuel.
   - But if it has no teleportation charges remaining, a `NotEnoughFuelError` is raised.

   We are evaluating you on your ability to use *inheritance* and *try-except blocks* correctly; make sure to call the superclass methods when appropriate. We have provided below the docstrings for the `Car` methods.

```
class Car:
    def __init__(self, fuel, eff):
        """ (Car, int, int) -> NoneType
        Create a new car at position (0,0) with a starting fuel
        of "fuel" and a fuel efficiency of eff.
        """

    def move(self, new_x, new_y):
        """ (Car, int, int) -> NoneType
        Move this car to position (new_x,new_y) and decrease
        its fuel accordingly. Raise NotEnoughFuelError if the
        car does not have enough fuel to complete the move.
        """
```

In the space below, implement the `TeleportCar` class. Docstrings are not required.

```
class TeleportCar(Car):
    def __init__(self):
        Car.__init__(20, 2)
        self.tele_charge = tele_charge

    def move(self, new_x, new_y):
        try:
            super().move(self, new_x, new_y)
        except NotEnoughFuelError:
            if self.tele_charge >= 1:
                self.tele_charge -= 1
            else:
                raise NotEnoughFuelError
```

3. (a) **[4]** Write a function that takes two stacks of integers, and returns a new stack with the items of the first stack, and then above them the items of the second stack, in their original order. That is, the top of the new stack is the *top of the second stack*, and the bottom of the new stack is the *bottom of the first stack*. This is a **non-mutating function**: when the function ends, it should return a new stack, but the original two stacks should be unchanged!

   You may only use the Stack ADT methods in this question; no other data structures (like Python lists) are allowed.

```
1 def combine(stack1, stack2):
2     """ (Stack of int, Stack of int) -> Stack of int
3     Return a new stack containing the elements of stack1,
4     and then the elements of stack2 above them.
5     """
      res = Stack()

      new_stack1 = Stack()
      new_stack2 = Stack()

      while not stack1.is_empty():          T(2n)
          element = stack1.pop()
          new_stack1.append(element)

      while not stack2.is_empty():          T(2m)
          element = stack2.pop()
          new_stack2.append(element)

      while not new_stack1.is_empty():      T(3n)
          element = new_stack1.pop()
          stack1.append(element)
          res.append(element)

      while not new_stack2.is_empty():      T(3m)
          element = new_stack2.pop()
          stack2.append(element)
          res.append(element)

      return res
```

   (b) **[2]** Suppose `stack1` has n items, and `stack2` has m items. What is the asymptotic (Big-Oh) runtime of your function in part (a)? Justify your answer.

```
T(5n + 5m) = 5T(n + m) = O(n + m)
```

4. Suppose we want to store some numbers in a data structure and be able to use it to efficiently **insert** and **search** for items (we ignore deletion for this question). We know from lecture that we could use either a sorted array-based list, or a binary search tree.

   (a) **[2]** Give the worst-case asymptotic runtime of each of the two operations on a *sorted array-based list*, and explain your answer for each operation. Note that your Big-Oh expressions should be in terms of n, the number of items stored in the list.

   (b) **[2]** Give the worst-case asymptotic runtime of each of the two operations on a *binary search tree*, and explain your answer for each operation. Note that your Big-Oh expressions should be in terms of h, the height of the BST.

   (c) **[2]** When the BST is complete, what is the relationship between its height h and its size n? Using this, explain why BSTs are often preferable to sorted lists for supporting these two operations.

   (d) **[3]** Suppose we start with an empty BST, and want to insert the numbers 1-7 so that the resulting tree is *complete*. State an order in which we can insert the nodes to make this happen, and draw the resulting tree.

5. (a) **[4]** Define a function `insert_sorted`, which takes a **sorted recursive linked list**, and an item, an inserts that item into the correct spot, keeping the list sorted. Note that this is a *mutating* method.

   You may **not** use a loop, nor any `LinkedListRec` methods other than the constructor and `is_empty` in your solution. Instead, use recursion to access and set the attributes directly.

```
def insert_sorted(lst, item):
    """ (LinkedListRec, item) -> LinkedListRec
    Precondition: lst is sorted in non-decreasing order

    Insert item into the correct location in lst,
    so that lst is still sorted after the function completes.

    >>> lst = LinkedListRec([3, 7, 10])  # [3 -> 7 -> 10]
    >>> insert_sorted(lst, 5)            # lst is [3 -> 5 -> 7 -> 10]
    """
    if lst.is_empty(lst, item):
        lst._front = lst._back = LinkedListNode(item, None)
    else:
        if item < lst._front:
            new_node = LinkedListNode(item, None)
            new_node._next = lst.front
            lst.front = new_node
        else:
            last_front = lst.front
            lst.front = lst.front.next_
            insert_sorted(lst, item)
            last_front.next_ = lst.front
            lst.front = last_front
```

   (b) **[2]** Describe the **best case** inputs and **worst case** inputs for your algorithm for part (a). (Don't just give an example; you should use the sentences "the best/worst case occurs when...")

```
The best case occurs when item is less than the front of lst
The worst case occurs when item is greater than the back of lst
```

6. (a) **[2]** Consider the heap corresponding to the list `[20, 15, 3, 2, 12, 1]`. Write the **list representation** of this heap after inserting the item `17`, using the algorithm from lecture. (You may find it useful to draw tree diagrams in the space below, but your final answer should be a list!)

(b) **[2]** Consider the heap corresponding to the list `[20, 15, 3, 2, 12, 1]`. Write the **list representation** of this heap after removing the maximum item, using the algorithm from lecture. (You may find it useful to draw tree diagrams in the space below, but your final answer should be a list!)

(c) **[4]** Write a recursive function which takes a list representation of a complete binary tree (*including* the extra `None`), and an index, and returns a list of all the items in the subtree rooted at the node with the specified index. (As usual, loops are not allowed.)

```
1  def get_items(items, i):
2    """ (list, int) -> list
3    Precondition: items represents a complete binary tree,
4    and starts with None (which is not an item in the tree!!)
5
6    Return a list containing the items in the subtree rooted
7    at the node with index i, in any order.
8    Return an empty list if i is out of bounds.
9
10   >>> get_items([None, 10, 5, 3, 2, 12, 16, 0], 2)
11   [5, 2, 12]
12   """
```

7. Consider the problem of taking two lists, each containing no duplicates, and returning a list of the elements they have in common. Here is a rather naïve brute force approach:

```
1  def intersection(lst1, lst2):
2      """ (list, list) -> list
3      Precondition: lst1, lst2 have no duplicates.
4      Return a list containing the elements that appear in both lists,
5      in any order.
6
7      >>> intersection([1,5,2],[3,2,1])
8      [1,2]
9      """
10     common = []
11     for item in lst1:
12         if item in lst2:
13             common.append(item)
14     return common
```

(a) **[2]** Assume that the list `__contains__` method, used in `item in lst2`, takes $O(n)$ time in the worst case (where n is the length of the list being searched for), and `append` takes constant time ($O(1)$). If both `lst1` and `lst2` have length n, what is the asymptotic worst case running time of this algorithm, in terms of n? Justify your answer.

`T(n) * T(n + 1) = O(n²)`

(b) **[5]** Recall from our discussion of mergesort that merging two lists into a sorted list turned out to be much easier if the original two lists were sorted. Using a similar idea, implement `fast_intersection`, which does the same thing as `intersection`, except it works in linear time if the input lists are sorted.

```
def fast_intersection(lst1, lst2):
    """ (list, list) -> list
    Precondition: lst1 and lst2 have no duplicate elements, and
                  both lst1 and lst2 are sorted in ascending order.
    Return a list containing the elements that appear in both lists,
    in ascending order.
    """
    index1 = 0
    index2 = 0
    common = []
    while index1 < len(lst1) and index2 < len(lst2):
        if lst1[index] == lst2[index2]:
            common.append(lst1[index1])
            index1 += 1
            index2 += 1
        elif lst1[index] < lst2[index2]:
            index1 += 1
        else:
            common.append(lst2[index2])
            index2 += 1
    return common
```

(c) **[2]** Assuming that `lst1` and `lst2` both have length n, explain why your algorithm runs in time $O(n)$.

the algorithm is using indices and comparison to iterate the two lists at same time.
the worst case occurs when case2 and case3 appear alternately.
e.g. lst1 = [1, 3, 5, 7, 9]
lst2 = [2, 4, 6, 8, 10]
in this case: T(6n + 7n) = O(n)

Bonus Question [3]

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

Recall from Midterm 2 that it is possible to reconstruct a binary tree from just its preorder and inorder traversals. Write a function to do this. You may assume that all duplicates appear as descendants on the *left* subtree, but don't assume that you're dealing with a binary *search* tree!

Note: you should assume that you have access to a `BinaryTree` class, with the same constructor and attributes as the `BinarySearchTree` class found on the cheat sheet.

Use this page for rough work.

Use this page for rough work.

Use this page for rough work.