Practical Questions CSCA48 – Week 7

This week's practical are meant to be a little bit harder than week 6's. Make sure you are confident about week 6 practical questions

a) Consider the following recursive function:

```
def m(a, b, c):
  if a > b:
    return m(b, a, c)
  elif b > c:
    return m(a, c, b)
  else:
    return c
```

i. Give a good doc-string for **m**.
ii. Give an example of concrete values **a**, **b**, and **c** for which **m(a, b, c)** results in as deep a recursion as possible.
iii. Show the runtime stack after calling m with your values of **a**, **b**, and **c**, just before **return c** executes for the first time.

b) The following function prints the contents of a stack. Rewrite it so that it doesn't use any loops. You may use helper functions if you wish. (We recommend using 2 helper functions.)

```
def print_stack(s):
  temp = Stack()
  # Flip s into temporary stack temp.
  while not s.is_empty():
    temp.push(s.pop())
  # Flip temp back into s, printing as we go.
  while not temp.is_empty():
    item = temp.pop()
    print(item)
    s.push(item)

def no_loop_print_stack(s):
  pass
```

c) Consider the recursive function below:

```
def mystery(L, start, finish):
    if start == finish:
        ans = L[start]
    else:
        middle = floor((start + finish) / 2)
        one = mystery(L, start, middle)
        two = mystery(L, middle+1, finish)
        print("Results: " + str(one) + " " + str(two))
        ans = max(one, two)
    print("Ans: " + str(ans))
    return ans
```

what is the output of mystery([98, 4, 222, -1], 0, 3).

d) Consider the recursive function below:

```
def mystery(s):
    if len(s) < 3:
        return s
    else:
        return s[2] + mystery(s[2:]) + mystery(s[3:]) + s[0]

if __name__ == "__main__":
    print(mystery("CSCA48"))
    print(mystery("MYSTERY"))
```

what are the outputs.

e) Implement a LinkedList, with all the methods describe in class, but this time, all methods should be recursive!
Hint: some helper methods may come in handy

f) Implement a binary search tree, with all the methods described in class, but now, all methods should be recursive!

```
class BTNode():
  def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None
```

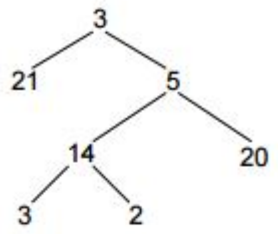
g) Consider the *BTNode* class, Write a *recursive* function **print_at_depth(root, dep)** that prints all the nodes of the tree that have a specified depth. Remember that the root has depth 1.

h) Consider the *BTNode* class, Write a *recursive* function **num_repeats(root, val)** that counts all the nodes of the tree that have a specified value.

i) Consider the following function, and the *BTNode* class it uses.

```
def mystery(root):
  if root == None:
    return 0
  else:
    left = mystery(root.left)
    right = mystery(root.right)
    return max(left, right) + root.data
```

i. What value is returned if we call **mystery** with a reference to the root of this binary tree:



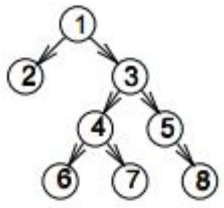
ii. How many calls to **mystery** occur, in total, as a result of that initial call to **mystery**?

iii. Write an appropriate comment that describes what **mystery** returns.

j) Consider the following function, and the BTNode class it uses.

```
def mystery(root):
  if (root.left == None) and (root.right == None):
    return root.data
  elif root.left == None:
    return mystery(root.right)
  elif root.right == None:
    return mystery(root.left)
  else:
    return mystery(root.left) + mystery(root.right)
```

i. What value is returned if we call **mystery** with a reference to the root of this binary tree:

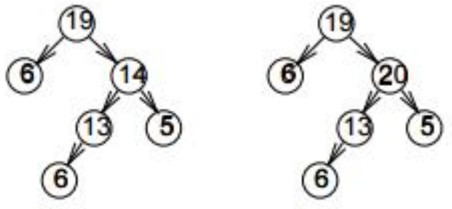


ii. How many calls to **mystery** occur, in total, as a result of that initial call to **mystery**? Include the initial call in your total.

iii. Write an appropriate comment that describes what **mystery** returns.

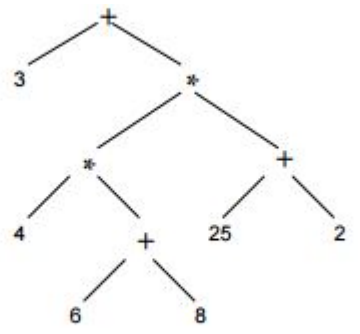
k) Consider the *BTNode* class, complete the function **is_bst(t)**. It returns true exactly when the tree rooted at **t** is a binary search tree. All the data in the tree are distinct. You can use a helper function but you **must** use recursion. Do not use any other data types such as lists or dictionaries.

l) A binary tree is “top heavy” iff, for every node is the tree, that node has a data greater than its children do (to the extent that it has children). For example, the tree on the left is top heavy and the tree on the right is not:



An empty tree is considered to be top heavy.
Complete the function **is_top_heavy(root)**. It returns whether or not the binary tree is top heavy. It uses the *BTNode* class.
Hint: Use recursion.

m) **<u>(Challenge)</u>**
Suppose we have a binary tree in which each leaf stores a float and each internal node stores an operator, either + or * (represented as a string). Such a tree can represent simple arithmetic expressions. For example, the following tree represents the expression (3+((4*(6+8))*(25+2))):



Complete the function **calculate(root)** for computing the value of an expression represented by such a tree. It uses the same *BTNode* class.

Discussion Questions:

1. In lecture we did hanoi(3) and hanoi(4), but stopped at hanoi(5). How long would it have taken? What about hanoi(6)? Or hanoi(10), or hanoi(100)? Assuming you move 1 piece per second, how long would it take to move the mythical tower of 100 levels? (After you've thought about this, take a look at http://larc.unt.edu/ian/TowersOfHanoi/index64.html).
2. Last week we discussed the concept of a stack overflow. This is a problem with recursion on large data, you relatively quickly run out of stack space. Is there a way around this? (hint: look up 'tail recursion')

Logic Question **(Challenge)**

You have 4 pieces of string. Each string takes exactly 1 hour to burn. However, the strings bun at inconsistent rates: one part of the string may burn quickly, another slowly, but overall they always take exactly 1 hour to fully burn.

How many different lengths of time can you accurately measure with these strings, and what are those lengths of time?