Quiz #6: Linked Lists, Trees, Inheritance ICS-33 Winter 2017

Print only this page, answer problem #1 on it, and submit it on Friday, at the **start of lecture**, in a pile for your lab at the front of class.

Name _____

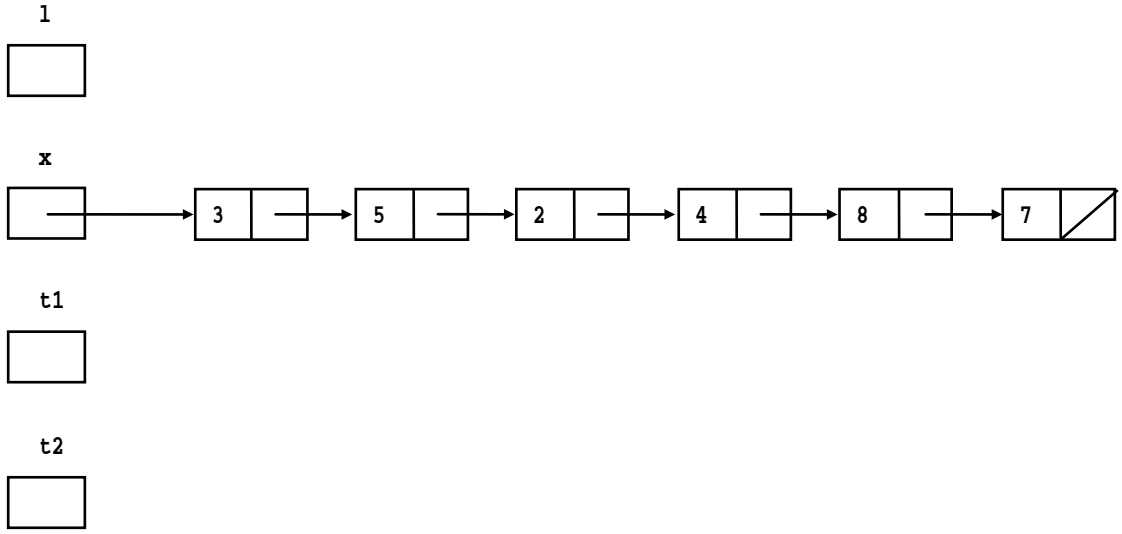
Lab # _____

When working on this quiz, recall the rules stated on the Academic Integrity statement that you signed. You can download the **q6helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write/test/debug your code. Submit your completed **q6solution** module online by Thursday, 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on Friday afternoon.

---

1. (7 pts) Examine the **mystery** method and hand simulate the call **mystery(x);** using the linked list below. **Lightly cross out** ALL references that are replaced and **Write in** new references: don't erase any references. It will look a bit messy, but be as neat as you can. Show references to **None** as **/**. Do work on scratch paper first.

```
def mystery(l):
  while l.next != None and l.next.next != None:
    t1      = l.next
    t2      = t1.next
    t1.next = t2.next
    t2.next = t1
    l.next  = t2
    l       = t1
```



2a. (4 pts) Define an **iterative** function named **pair_sum**; it is passed a linked list (**ll**) as an argument. It returns a reference to the front of a new linked list (without mutating the old one) that includes the sum of each adjacent pair of values (an **LN** object with the sum of the first and second, an **LN** object with the sum of the third and fourth, etc.). If the next **LN** object that is supposed to start a pair is not followed by another **LN** object (there are an odd number of **LN** objects), ignore it. For example if we defined

```
a = list_to_ll([1,2,3,4,5,6,7,8])
```

pair_sum(a) returns the linked list **3->7->11->15->None**. Adding another value to original list produces the same result. You **may not** use Python **lists**, **tuples**, **sets**, or **dicts** in your code: **use linked lists processing only**. Hint: this code both traverses the linked list while building the new list (it is a variant of the code for copying a list, which appears in the course notes); debug your code by hand simulation.

2b. (4 pts) Define a **recursive** function named **pair_sum_r** that is given the same argument and produces the same result as the iterative version, using recursion: use no looping, local variables, etc. Hint: use the 3 proof rules to help synthesis your code. You might try writing the recursive solution first, it is simpler.

3. (4 pts) Write the **recursive** function **count**; it is passed balanced binary (**any binary tree**, not necessarily a binary search tree) and a value as arguments. It returns the number of times the values is in the tree. In the binary tree below, **count(tree,1)** returns **1**, **count(tree,2)** returns **2**, **count(tree,3)** returns **4**.

```
..1
....3
3
....3
..2
......2
....3
```

Hint: use the 3 proof rules to help synthesis your code.

4. (6 pts) Define a derived class named **StringVar_WithHistory**, based on the **StringVar** class in **tkinter**; it remembers what sequence of values it was set to, and is able to undo each setting.

The **StringVar** class defines 3 methods: **__init__** (**self**), **get** (**self**), and **set** (**self,value**). The **set** method changes the state of the **StringVar** object to be **value**; the **get** method returns the string it is currently set to. The **StringVar_WithHistory** derived class inherits **get** and overrides **__init__** and **set**.

Define the derived class **StringVar_WithHistory** with only the following methods (get is purely inherited):

- **__init__** (**self**): initializes the base class; creates a history **list** for storing the values **set** is called with.
- **set** (**self,value**): if the value is different from the current value, **set** the **StringVar** to **value** and remember it in the history **list** (if it is the same as the current value, do nothing: no *new* selection).
- **undo** (**self**): undo the most recently selected option by updating the **StringVar** and the history **list** (but only if the currently selected option wasn't the first one: that selection cannot be undone).

You cannot test **StringVar_WithHistory** by itself, but must test it using the **OptionMenuUndo** class, which is in the download (itself class derived from **OptionMenu**). You can simulate the GUI (how I will test it) or can actually build/test a version of the GUI that allows you to click the GUI to select options and undo selections. See the simulation in the download (for how it should behave on one complex example).

Note: if you see the error message **AttributeError: 'NoneType' object has no attribute 'globalgetvar'** then you have not initialized the **StringVar** appropriately by calling its **__init__** method.