For Problems 1 and 2, print the **Answers.pdf** file (in the **q6helper** folder and on Gradescope), write your answers on it, then upload it to Gradescope by Thursday 5/19 at 11:30pm. If you cannot print files, write your answer as best you can on a blank sheet of paper and upload it to Gradescope. Upload your solutions to problems 3-6 in the **q6solution.py** file the normal way to Checkmate by Thursday 5/19 at 11:30pm.

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 morning.

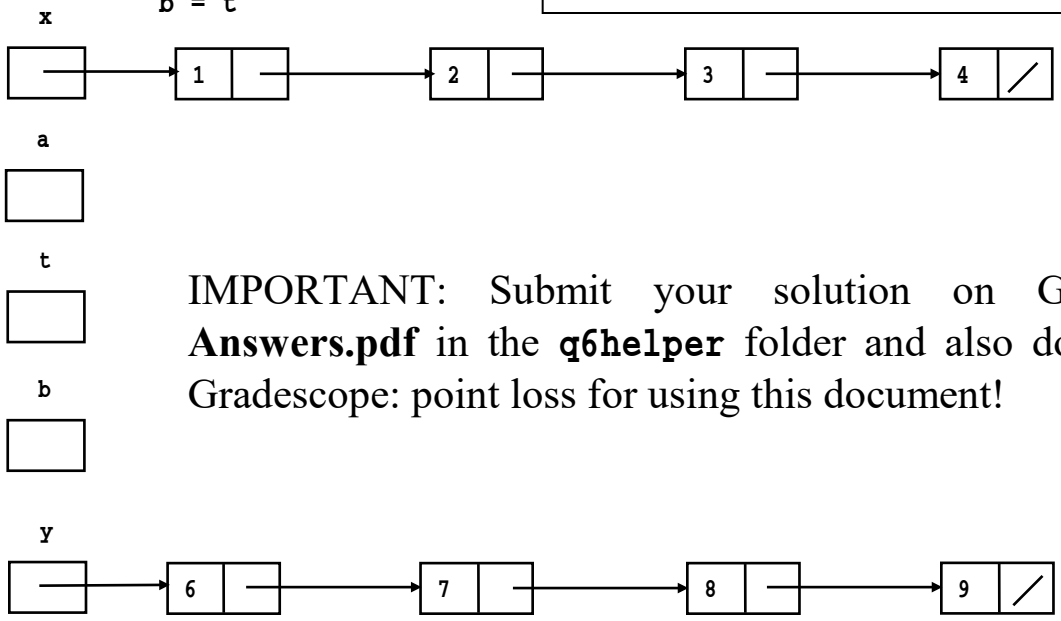
---

1. (6 pts) Examine the **mystery** method and hand simulate the call **mystery(x,y);** 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 your work on scratch paper first.

```
def mystery(a,b):
    while b != None:
        t = a.next.next
        a.next.next = b
        a = b
        b = t
```

```
class LN:
    def __init__(self, value, next=None):
        self.value, self.next = value, next

class TN:
    def __init__(self, value, left=None, right=None):
        self.value, self.left, self.right = value, left, right
```



IMPORTANT: Submit your solution on Gradescope using **Answers.pdf** in the **q6helper** folder and also downloadable from Gradescope: point loss for using this document!

2. (2 pts) Draw the binary search tree that results from inserting the following values (in the following order) into an empty binary search tree: **13**, **10**, **6**, **3**, **8**, **5**, **4**, **16**, **11**, **17**, **1**, **2**, **15**, **19**, **18**, **14**, **9**, **20**, **7**, and **12**. Draw the number for each tree node, with lines down to its children nodes. Space-it-out to be easy to read. Answer the questions in the box.

Size =
Height =

IMPORTANT: Submit your solution on Gradescope using **Answers.pdf** in the **q6helper** folder and also downloadable from Gradescope: point loss for using this document!

3a. (3 pts) Define an **iterative** function named **append_ordered**; it is passed two arguments: a linked list (**ll**) whose values are ordered from smallest to biggest (they can be **ints**, **strs**, anything that can be compared), and another value (**v**). It returns a reference to the front of a linked list that includes all the values of the original linked list, and **v**, all in order. We call it like **x = append_ordered(x,v)**. You may create exactly one new **LN**, for storing the value of **v**. For example, if we defined

```
x = list_to_ll([1, 3, 8, 12])
```

and wrote **x = append_ordered(x, 10)** , then **str_ll(x)** returns the string **"1->3->8->10->12->None"**. Your code should work correctly regardless of whether the other value is added at the front, middle, or rear of the linked list. You may not use any other data structures (e.g., you may not put all the values into a list, sort the list, and then put all the values into a linked list), nor call any other helper functions: write the iterative code.

3b. (3 pts) Define a **recursive** function named **append_ordered _r** that is given the same arguments and produces the same result as the iterative version specified above (and abiding by the same restrictions). Also, use no looping, no local variables, etc. Hint: use the 3 proof rules to help synthesis your code. You might first write code that always appends the value at the end of the linked list.

4. (4 pts) Write the **recursive** function **max_depth**; it is passed a binary (**any binary tree**, not necessarily a binary search tree) and a value as arguments. It returns the maxim depth in the tree at which that value occurs. Note the root is depth **0**; its children are at depth **1**; its grandchildren are at depth **2**, etc. Generally, if a parent is at depth **d**, its children are at depth **d+1**. In the binary tree below, **max_depth(tree,1)** returns **1**, **max_depth (tree,2)** returns **3**, **max_depth (tree,3)** returns **2**. The value may appear at many depths. If it is not in the tree, return some negative number or **None**.

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

Hint: use the 3 proof rules to help synthesis your code. In my **max_depth** function, I wrote a recursive helper function and returned the result it returns when called: this helper function had a third parameter specifying the depth of the node at the top of the tree it is processing (**0** for the actual root of the tree). My helper function bound two local variables to recursive calls and then used them (it simplified my code).

5. (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**, overrides **__init__** and **set**, and defines an **undo** method.

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.