Name ____________________

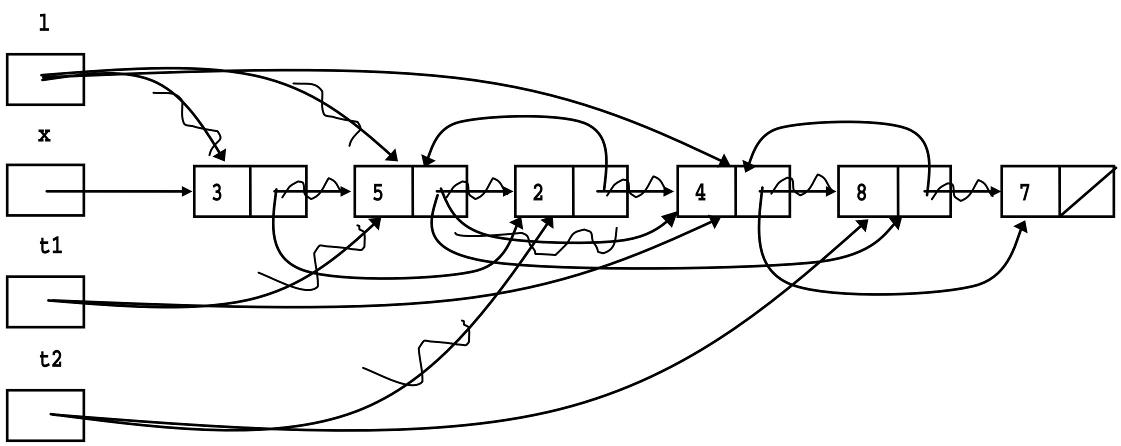
Lab # (1-8) ____________________

Recall the Academic Integrity statement that you signed. Write all answers clearly on these pages, ensuring your final answers are easily recognizable. The number of points for each problem is clearly marked, for a total of 25 points. I will post my solutions on the web on Monday, off the **Solutions** link, after class.

Download the `q6helper` project folder (available for Friday on the weekly schedule) in which to write/test/debug your code for problems 2-5. Submit your completed `q6solution.py` module online and write your answers on this quiz (hand in only one two-sided sheet: not multiple pages attached/stapled). Both are due at the start of class on Wednesday.

1. (6 pts) Examine the `mystery` method and hand simulate the call `mystery(x);` using the linked list below. Use the LN class from the lecture notes. Write your solution as I showed in class. **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. Probably it is best to do this problem first on another sheet of paper.

```
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
```



2. (5 pts) Define a **recursive** function named `count`; it is passed a binary tree (it doesn't matter if it is a binary **search** tree) and a predicate as arguments; it returns a count of all the values in the tree for which the predicate returns `True`.

```
def count(t,p):
    if t == None:
        return 0
    else:
        return int(p(t.value)) + count(t.left,p) + count(t.right,p)
```

or

```
        return (1 if p(t.value) else 0) + count(t.left,p) + count(t.right,p)
```

3. (4 pts) Define a **recursive** function named **equal**; it is passed two linked lists; it returns whether or not the linked lists contain exactly the same values in the same order. For example if we defined

```
a = list_to_ll(['one','two','three','four'])
b = list_to_ll(['one','two','three','four'])
c = list_to_ll(['one','two','three'])
d = list_to_ll(['one','two','three','four','five'])
e = list_to_ll(['one','two','four','four'])
```

the **equals(a,b)** is **True**; but **equals(a,c)**, **equals(a,d)**, and **equals(a,e)** are **False**.

```
def equal(ll1,ll2):
    if ll1 == None or ll2 == None:
        return ll1 == ll2
    else:
        return ll1.value == ll2.value and equal(ll1.next,ll2.next)
```

4. (4 pts) Define a **recursive** function named **min_max**; it is passed a linked list; it returns a 2-tuple containing the minimum value followed by the maximum value. If the linked list is empty, return **(None, None)**. For example, if we called this function as **min_max(list_to_ll([1, 3, 7, 2, 0]))** the returned result would be **(0, 7)**. Hint: Unlike most recursive functions, I bound the result of the recursive call to a name and then used that name to compute the result of the main call (using two conditional expressions).

```
def min_max(ll):
    if ll == None:
        return (None, None)
    else:
        min,max = min_max(ll.next)
        return (ll.value if min == None or ll.value < min else min,
                ll.value if max == None or ll.value > max else max)
```

5. (6 pts) Define a class named **Dumpable**. This class has one method: **get_state**; it is parameterless and returns a dictionary of all the names in **self** (look in its **__dict__**) that are not bound to function objects (test functions by using **inspect.isfunction** on the objects). When a class inherits from **Dumpable**, its instances can call **get_state()** to get a dictionary of all the names in its namespace (that are not functions) and their values. For example, if we first defined class **Modular_Counter(Dumpable):** and then **mc = Modular_Counter(0,3)** and then **print(mc.get_state())** it would print the following dictionary: **{'_value': 0, '_modulus': 3}**

```
class Dumpable:
    def get_state(self):
        return {k:v for k,v in self.__dict__.items() if not inspect.isfunction(v)}
```