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 2/24 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 2/24 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 afternoon.

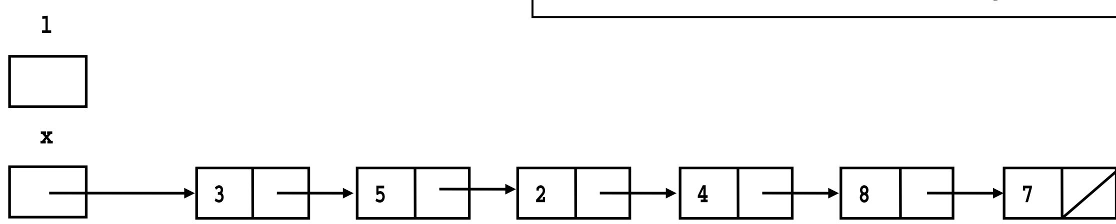
---

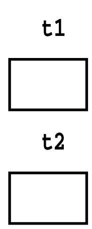
1. (6 pts) Examine the `mystery` method and hand simulate the call `mystery(x);` using the linked list below. **Lightly cross out** ALL references that are replaced and **Write in** new references: don't erase any references. It will look a bit messy, but be as neat as you can. Show references to `None` as `/.` Do your 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
```

```
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
```





Submit Solution on Gradescope using **Answers.pdf**; not this picture

2. (3 pts) Draw the binary search tree that results from inserting the following values (in the following order) into an empty binary search tree: `9`, `1`, `8`, `11`, `14`, `4`, `6`, `2`, `5`, `12`, `16`, `13`, `3`, `18`,`20`, `19`, `17`, `10`, `15`, `and 7`. 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 =

Submit Solution on Gradescope using **Answers.pdf**; not this picture

3a. (3 pts) Define an **iterative** function named **alternate_i**; it is passed two linked lists (**ll1** and **ll2**) as arguments. It returns a reference to the front of a linked list that alternates the **LN**s from **ll1** and **ll2**, starting with **ll1**; if either linked list becomes empty, the rest of the **LN**s come from the other linked list. So, all **LN**s in **ll1** and **ll2** appear in the returned result (in the same relative order, possibly separated by values from the other linked list). The original linked lists are **mutated** by this function (the **.next**s are changed; create no new **LN** objects). For example, if we defined

`a = list_to_ll(['a', 'b', 'c',])` and `b = list_to_ll([1,2,3,4,5])`

`alternate_i(a,b)` returns `a->1->b->2->c->3->4->5->None` and `alternate_i(b,a)` returns `1->a->2->b->3->c->4->5->None`. You **may not** create/use any Python data structures in your code: **use linked list processing only**. Change only **.next** attributes (not **.value** attributes). Hints: see the **q6solution.py** file for some hints and then hand-simulate the code your write to debug it (I wrote two different solutions that were 6 and 15 lines). This is hard, you might want to solve the recursive one in part 3b first.

3b. (3 pts) Define a **recursive** function named **alternate_r** that is given the same arguments and produces the same result as the iterative version above, but here using recursion. You **may not** create/use any Python data structures in your code: **use linked list processing only**: use no looping, local variables, etc. Hint: see the recursive code for appending a value at the end of a linked list; of course, try to use the 3 proof rules to help synthesize your code. You might try writing the recursive solution first, it is simpler! (mine was 7 lines)

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

5. (6 pts) Define a class named **bidict** (**bi**directional **dict**) derived from the **dict** class; in addition to being a regular dictionary (using inheritance), it also defines an auxiliary/attribute dictionary that uses the **bidict**'s values as keys, associated to a set of the **bidict**'s keys (the keys associated with that value). Remember that multiple keys can associate to the same value, which is why we use a **set**: since keys are hashable (hashable = immutable) we can store them in **set**s. Finally, the **bidict** class stores a class attribute that keeps track of a **list** of all the objects created from this class, which two static functions manipulate.

Define the class **bidict** with the following methods (some override **dict** methods); you may also add helper methods: preface them with single underscores):

- **__init__(self,initial =[],**kargs)**: initializes the **dict** in the base class and also creates an auxiliary dictionary (I used a **defaultdict**) named **_rdict** (**r**everse**dict**: you **must use this name** for the **bsc** to work correctly) whose key(s) (the values in the **bidict**) are associated with a **set** of values (their keys in the **bidict**): initialize **_rdict** by iterating through the newly initialized dictionary.

  For a **bidict** to work correctly, its keys and their associated values must all be hashable. We define any object as hashable if it (a1) has a **__hash__** attribute, and (a2) the attribute's value is not **None**; also (b) if the object is iterable (has an **__iter__** attribute) then every value iterated over is also hashable (by this same definition). Also note that **str** is hashable as a special case: if we apply the above definition it will create infinite recursion because every value that we iterate over in a **str** is a **str** that we can iterate over! Raise a

ValueError exception if any value is not hashable. Note you can use the **hasattr** and **getattr** functions (which I used in a recursive static helper method named **_is_hashable**).

Finally, **_rdict** should never store a key that is associated with an empty set: **__setitem__** and **__delitem__** must ensure this invariant property (I wrote a helper method to help them do it).

For example if we define, **bd = bidict(a=1,b=2,c=1)** then **_rdict** stores **{1: {'a', 'c'}, 2: {'b'}}**. If I tried to construct **bidict(a=[])** then **__init__** would raise a **ValueError** exception. We will continue using this example below.

- **__setitem__** (**self,key,value**): set **key** to associate with **value** in the dictionary and modify the **_rdict** to reflect the change. For example, executing **bd['a'] = 2** would result in the **_rdict** being changed to **{1: {'c'}, 2: {'a', 'b'}}**; then executing **bd['c'] = 2** would result in the **_rdict** being changed to **{2: {'a', 'b', 'c'}}**, with **1** no longer being a key in **_rdict** because it is no longer a value in the **bidict**.
- **__delitem__** (**self,key**): for any **key** in the dictionary, remove it and modify the **_rdict** to reflect the change.
- **__call__** (**self,value**): returns the set of keys (keys in the bidict) that associate with **value**: just lookup this information in the **_rdict**.
- **clear** (**self**): remove all keys (and their associated values) from the **bidict** modify the **_rdict** to reflect the change.
- **all_objects** (): a static method that returns a **list** of all the objects ever created by **bidict**. Think how/where to store this **list**.
- **forget** (**object**): a static method that forgets the specified **bidict** object, so **all_objects** doesn't return it.