e) The textbook's `remove(item)` and `index(item)` operations "Assume the item is present in the list." Thus, they would have a precondition like "Item is in the list." When writing a program using an UnorderedList object (say `myGroceryList = UnorderedList()`), how would the programmer check if the precondition is satisfied?

```
itemToRemove = input("Enter the item to remove from the Grocery list: ")
if myGroceryList.search(itemToRemove):
    myGroceryList.remove(itemToRemove)
```

f) The `remove(item)` and `index(item)` methods both need to look for the `item`. What is inefficient in this whole process?

User program calls search method to check precondition, then remove/index method calls search to validate precondition again.

g) Modify the `search(targetItem)` method code in (a) to set additional data attributes to aid the implementation of the `remove(item)` and `index(item)` methods.

h) Write the `index(item)` method including a check of its precondition(s).

```
def index(self, item):
    if self.search(item) == False:
        raise ValueError("Cannot find index if item is not in List")
    return self._currentIndex
```

i) Write the `remove(item)` method including a check of its precondition(s).

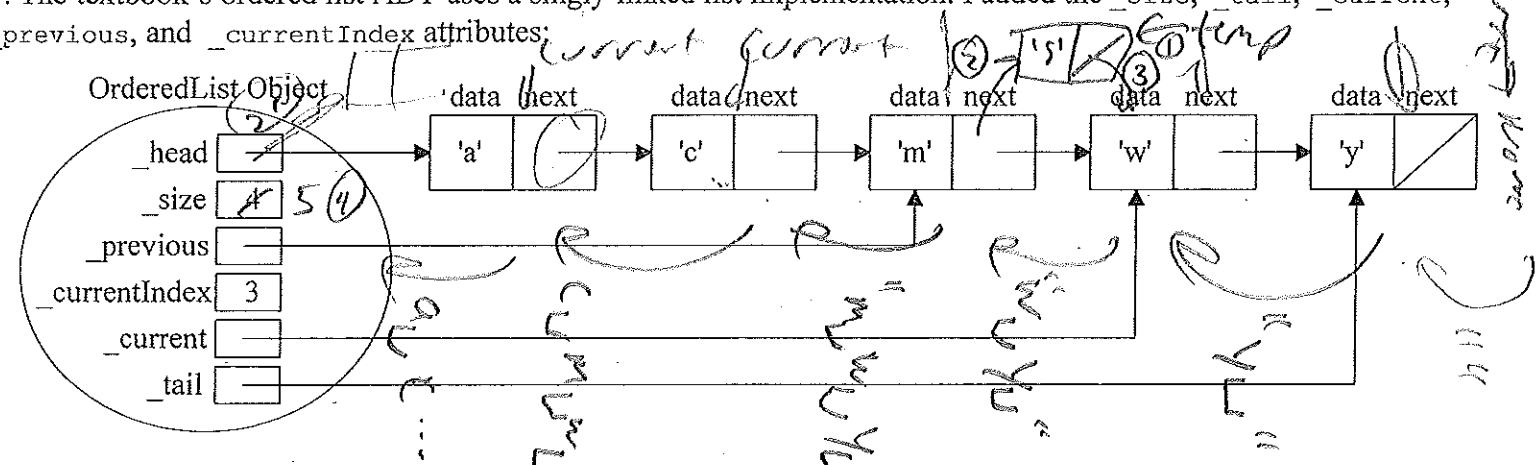
```
def remove(self, item):
    if not self.search(item):
        raise ValueError("Cannot remove item not in the list")

    (see actual code to follow)
```

```
""" File:  unordered_linked_list.py
    Description:  Unordered List ADT implemented using singly-linked list.
"""

from node import Node

class UnorderedList(object):

    def __init__(self):
        """ Constructs an empty unsorted list.
            Precondition:  none
            Postcondition:  Reference to empty unsorted list returned.
        """
        self._head = None
        self._tail = None
        self._size = 0
        self._current = None
        self._previous = None
        self._currentIndex = -1

    def search(self, targetItem):
        """ Searches for the targetItem in the list.
            Precondition: none.
            Postcondition:  Returns True and makes it the current item if
targetItem is in the list;
                            otherwise False is returned.
        """
        if self._current != None and self._current.getData() == targetItem:
            return True

        self._previous = None
        self._current = self._head
        self._currentIndex = 0
        while self._current != None:
            if self._current.getData() == targetItem:
                return True
            else: #inch-worm down list
                self._previous = self._current
                self._current = self._current.getNext()
                self._currentIndex += 1
        return False

    def add(self, newItem):
        """ Adds the newItem to the list.
            Precondition: newItem is not in the list.
            Postcondition:  newItem is added to the list.
        """
        if self.search(newItem):
            raise ValueError("Cannot not add since item is already in the
list!")
```

```
        temp = Node(newItem)
        if self._size == 0:
            self._tail = temp
        else:
            temp.setNext(self._head)
        self._head = temp
        self._size += 1

    def remove(self, item):
        """ Removes item from the list.
            Precondition:  item is in the list.
            Postcondition:  Item is removed from the list.
        """
        if not self.search(item):
            raise ValueError("Cannot remove item since it is not in the
list!")

        if self._current == self._tail:
            self._tail = self._previous

        if self._current == self._head:
            self._head = self._head.getNext()
        else:
            self._previous.setNext(self._current.getNext())
        self._current = None
        self._size -= 1

    def isEmpty(self):
        """ Checks to see if the list is empty.
            Precondition:  none.
            Postcondition:  Returns True if the list is empty; otherwise
returns False.
        """
        return self._size == 0

    def length(self):
        """ Returns the number of items in the list.
            Precondition:  none.
            Postcondition:  Returns the number of items in the list.
        """
        return self._size

    def append(self, newItem):
        """ Adds the newItem to the tail of list.
            Precondition: newItem is not in the list.
            Postcondition:  newItem is added to the tail of list.
        """
        if self.search(newItem):
            raise ValueError("Cannot not append since item is already in the
list!")
```

Data Structures (CS 1520) Lecture 9 Name:

1. The textbook's ordered list ADT uses a singly-linked list implementation. I added the `_size`, `_tail`, `_current`, `_previous`, and `_currentIndex` attributes:



The `search(targetItem)` method searches for `targetItem` in the list. It returns `True` if `targetItem` is in the list; otherwise it returns `False`. Additionally, it has the side-effects of setting `_current`, `_previous`, and `_currentIndex`. The complete `search(targetItem)` method code for the `OrderedList` is:

```
class OrderedList:

   def search(self, targetItem):
       if self._current != None and self._current.getData() == targetItem:
          return True

       self._previous = None
       self._current = self._head
       self._currentIndex = 0
       while self._current != None:
           if self._current.getData() == targetItem:
              return True
           elif self._current.getData() > targetItem:
              return False
           else: #inch-worm down list
              self._previous = self._current
              self._current = self._current.getNext()
              self._currentIndex += 1
       return False
```

a) What's the purpose of the "`elif self._current.getData() > targetItem:`" check? If we are searching for 's' and current points to node with 'w', then we can stop because the list is ordered and we will not find 's' after 'w'.

b) Complete the `add(item)` method including a check of it's precondition: `newItem` is not in the list.

```
def add(self, newItem):
    if self.search(newItem):
        raise ValueError("Cannot add duplicate items")
    temp = Node(newItem)
    if self._current == self._head:
        self._head = temp
    else: self._previous.setNext(temp)
    temp.setNext(self._current)
    if self._current == None:
        self._tail = temp
    self._size += 1
```

self._current = None

2. A *recursive function* is one that calls itself. Complete the recursive code for the `countDown` function that is passed a starting value and proceeds to count down to zero and prints "Blast Off!!!".

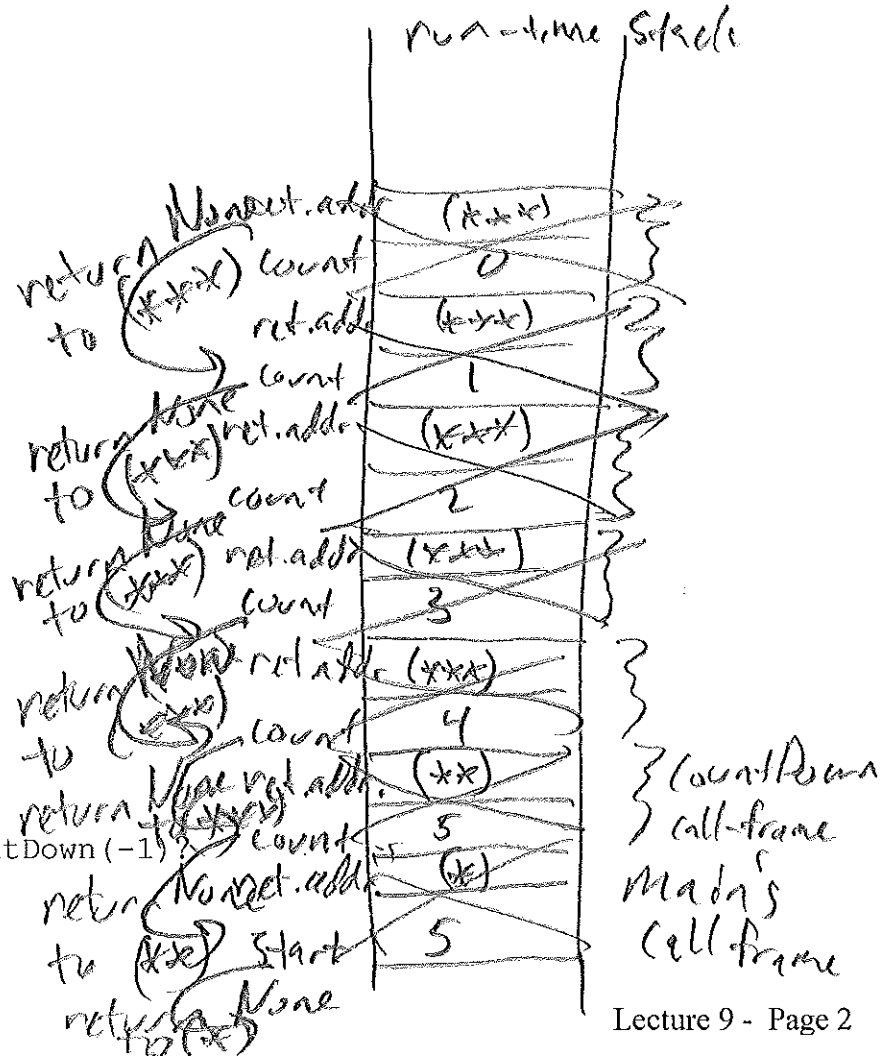
Hint: The `countDown` function, like most recursive functions, solves a problem by splitting the problem into one or more simpler problems of the same type. For example, `countDown(10)` prints the first value (i.e, 10) and then solves the simpler problem of counting down from 9. To prevent "infinite recursion", if-statement(s) are used to check for trivial *base case*(s) of the problem that can be solved without recursion. Here, when we reach a `countDown(0)` problem we can just print "Blast Off!!!".

```
""" File:  countDown.py """

def main():
    start = eval(input("Enter count down start: "))
    print("\nCount Down:")
    countDown(start)
(**)
def countDown(count):
    if count <= 0:
        print("Blast off!!!")
    else:
        print(count)
        countDown(count-1)
(***)
main()
(*)
```

Program Output:

```
Enter count down start: 10

Count Down:
10
9
8
7
6
5
4
3
2
1
Blast Off!!!
```

a) Trace the function call `countDown(5)` on paper by drawing the run-time stack and showing the output.



b) What do you think will happen if your call `countDown(-1)`?

c) Why is there a limit on the depth of recursion?

Because memory space for run-time stack is finite (default is 1000)

Run-time Stack

- allocate memory for function calls

On function call; push a call-frame on run-time stack containing:

(1) ret. addr.
(2) parameters
(3) local variables

On function return, we pop call frame and execute at ret. addr.
return specified value or default None

3. Complete the recursive `strHelper` function in the `__str__` method for our `OrderedList` class.

```
def __str__(self):
    """ Returns a string representation of the list with a space between each item. """

    def strHelper(current):
        if current == None:
            return
        else:
            return str(current.getData()) + " " + strHelper(current.getNext())


    return "(head) " + strHelper(self._head) + "(tail)"
```

4. Some mathematical concepts are defining by recursive definitions. One example is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

After the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined recursively as:

$Fib_0 = 0$

$Fib_1 = 1$

$Fib_N = Fib_{N-1} + Fib_{N-2}$ for $N \geq 2$.

a) Complete the recursive function:

```
def fib (n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

b) Draw the *call tree* for fib(5).