1. The textbook's ordered list ADT uses a singly-linked list implementation. I added the `size`, `tail`, `current`, `previous`, and `currentIndex` attributes:

```python
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 "else self._current.getData() > targetItem: return False" check?

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

```python
    def add(self, newItem):
        if self.search(newItem):
            raise ValueError("Cannot add duplicates to list.")

        temp = Node(newItem)
        if self._previous == None:
            self._head = temp
        else:
            self._previous.setNext(temp)
        temp.setNext(self._current)
        self._size += 1
```

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:
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!!!”.

```python
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: 25

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(3)? "infinite recursion"

c) Why is there a limit on the depth of recursion?
Non-recursive code:
```python
def count_down(count):
    while count > 0:
        print("count")
        count -= 1
        print("count")
```

Recursive:
```python
base case(s):
count == 0: print("Blast off!!!")
```
```python
recursive case(s):
count > 0: print(count)
count_down(count-1)
```

What happens on a function call?
- Push call-frame on run-time stack including:
  1. "return address" - where to return to in caller
  2. parameters
  3. local variables

What happens on a return?
- Continue execution at vet. addr. with the returned value or default None by popping call-frame

Run-time stack

Local Variables

Global Variables

Diagram of function calling:
```plaintext
main(1)
  A(x):
    B(y):
```

Diagram of memory layout:
```plaintext
A
main
```
3. Complete the recursive strHelper function in the str method of our OrderedList class.

```python
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 defined by recursive definitions. One example is the Fibonacci series:

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

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:

\[ \begin{align*}
    \text{Fib}_0 &= 0 \\
    \text{Fib}_1 &= 1 \\
    \text{Fib}_n &= \text{Fib}_{n-1} + \text{Fib}_{n-2} \quad \text{for } n \geq 2.
\end{align*} \]

a) Complete the recursive function:

```python
def fib(n):
```

b) Draw the call tree for fib(5).
1. The textbook's ordered list ADT uses a singly-linked list implementation. I added the \_size, \_tail, \_current, \_previous, and \_currentIndex attributes:

![Diagram of OrderedList Object with pointers]

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:

```python
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?

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

```python
def add(self, newItem):
    # Your code here
```