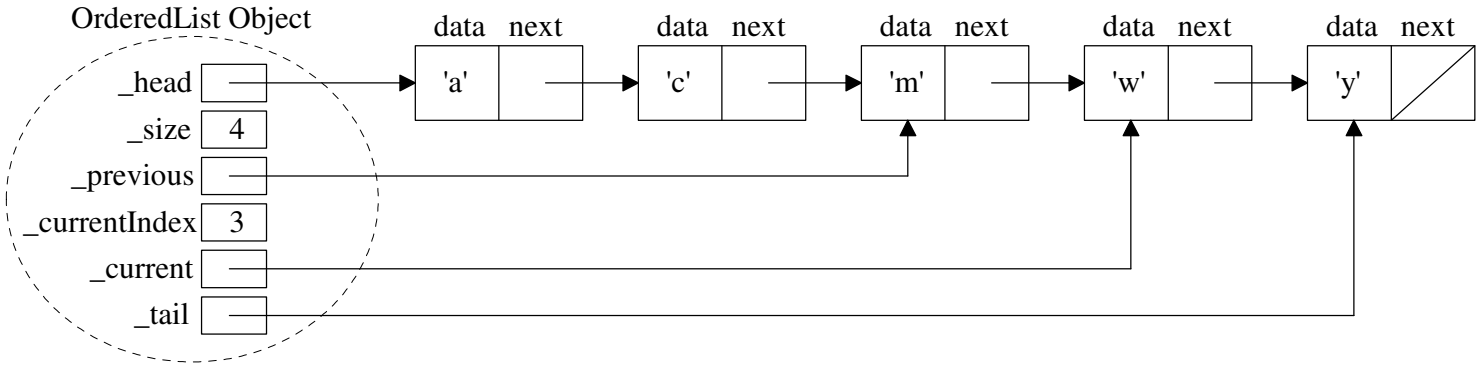
1. The textbook's **ordered list** ADT uses a singly-linked list implementation. I added the `_size`, `_tail`, `_current`, `_previous`, and `_currentIndex` attributes:



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?

Consider the `add(item)` method with the precondition: `item` is not in the list.

b) Write the precondition check at the start of the `add(item)` method.

c) Suppose you are adding the item value of 's'. Update the above picture for this "normal" case, and number the steps in the drawing.

d) What special cases need to be considered for the add method?

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):









main()
```

Program Output:

```
Enter count down start: 10

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

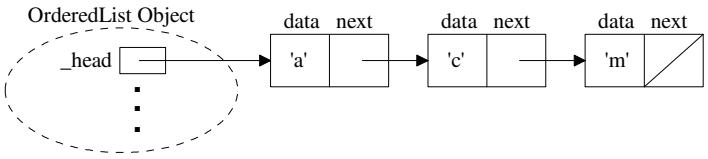
a) Trace the function call `countDown(5)` on paper by drawing the run-time stack and showing the output.

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

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

3. The non-recursive __str__ method for `OrderedList` object below would return: `"(head) a c m (tail)"`

```
def __str__(self):
   resultStr = "(head) "
   current = self._head
   while current != None:
       resultStr += str(current.getData())+" "
       current = current.getNext()
   return resultStr + "(tail)"
```



We can thing of building the string for the list as "a " + (string for the rest of the list)

a) 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):









   # Start of __str__ method execution
   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, ...

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 $N \geq 2$.

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

a) Complete the recursive function:

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

c) On my office computer, the call to fib(40) takes 22 seconds, the call to fib(41) takes 35 seconds, and the call to fib(42) takes 56 seconds. How long would you expect fib(43) to take?

d) How long would you guess calculating fib(100) would take on my office computer?

e) Why do you suppose this recursive fib function is so slow?

f) What is the computational complexity? *O*( )

g) How might we speed up the calculation of the Fibonacci series?

5. A VERY POWERFUL concept in Computer Science is *dynamic programming*. Dynamic programming solutions eliminate the redundancy of divide-and-conquer algorithms by calculating the solutions to smaller problems first, storing their answers, and looking up their answers if later needed instead of recalculating them.

We can use a list to store the answers to smaller problems of the Fibonacci sequence.
To transform from the recursive view of the problem to the dynamic programming solution you can do the following steps:

1) Store the solution to smallest problems (i.e., the base cases) in a list
2) Loop (no recursion) from the base cases up to the biggest problem of interest. On each iteration of the loop we:
   - solve the next bigger problem by looking up the solution to previously solved smaller problem(s)
   - store the solution to this next bigger problem for later usage so we never have to recalculate it

a) Complete the dynamic programming code:

```
def fib(n):
    """Dynamic programming solution to find the nth number in the Fibonacci seq."""

    # List to hold the solutions to the smaller problems
    fibonacci = []

    # Step 1:  Store base case solutions
    fibonacci.append(    )
    fibonacci.append(    )

    # Step 2:  Loop from base cases to biggest problem of interest
    for position in range(                              ):

        fibonacci.append(                                                    )

    # return nth number in the Fibonacci sequence
    return
```

Running the above code to calculate fib(100) would only take a fraction of a second.

b) One tradeoff of simple dynamic programming implementations is that they can require more memory since we store solutions to **all** smaller problems. Often, we can reduce the amount of storage needed if the next larger problem (and all the larger problems) don't really need the solution to the really small problems, but just the larger of the smaller problems. In fibonacci when calculating the next value in the sequence how many of the previous solutions are needed?