1. The textbook’s unordered list ADT uses a singly-linked list implementation. I added the `size` and `tail` attributes:

   a) The `search(targetItem)` method searches for `targetItem` in the list. It returns `True` if `targetItem` is in the list; otherwise it returns `False`. Complete the `search(targetItem)` method code:

   ```python
   class UnorderedList:
   
   def search(self, targetItem):
       self._current = self._head
       while self._current != None:
           if self._current.GetData() == targetItem:
               return True
           else:
               self._current = self._current.Next()

       return False
   ```

   b) The textbook’s unordered list ADT does not allow duplicate items, so operations `add(item)`, `append(item)`, and `insert(pos, item)` would have what precondition?

   The item is not already in the list.

c) Complete the `append(item)` method including a check of its precondition(s)?

   ```python
   def append(self, item):
       if self.search(item) == True:
           raise Exception("Cannot append duplicate item to list.")
       temp = Node(item)
       if self._size == 0:
           self._head = temp
       else:
           self._tail.setNext(temp)
       self._size += 1
   ```

d) Why do you suppose I added a `tail` attribute? O(1) append
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 a UnorderedList object (say `myGroceryList = UnorderedList()`), how would the programmer check if the precondition is satisfied?

```python
itemToRemove = input("Enter the item to remove from the Grocery list: ")
if myGroceryList.search(itemToRemove) == True:
    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 of UnorderedList check precondition with search method starts with check of precondition too
- Method walks down whole list to do its job
- Inefficient because 3 walks down list performed.

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).

```python
def index(self, item):
    if self.search(item) == False:
        raise Exception("item to find index must be in list")
    return self._currentIndex
```

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

```python
def remove(self, item):
    if self.search(item) == False:
        raise Exception("(see attached)")
```
1. `temp = self._current`

2. `self._previous.setNext(temp.getNext())`

3. `self._size -= 1`

4. `self._current = None`

5. `return temp.getData()`

**Special Case(s):**

**Tail:**

1. Remove last node in list +

2. If `self._current == self._tail`:
   - `self._tail = self._previous`

3. Else:
   - `self._current = self._head`

4. `return self._head.getData()`
unordered_linked_list.py

''' 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!")
unordered_linked_list.py

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!"

temp = self._current
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
return temp.getData()

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!")
unordered_linked_list.py

list!

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

def index(self, item):
    """ Returns the position of item in the list.
    Precondition: item is in the list.
    Postcondition: Returns the position of item from the head of list.
    """
    if not self.search(item):
        raise ValueError("Cannot determine index since item is not in the list!")
    return self._currentIndex

def insert(self, pos, newItem):
    """ Inserts newItem at position pos of the list.
    Precondition: position pos exists in the list, and newItem is not in the list.
    Postcondition: The item has newItem inserted at position pos of the list.
    """
    if not isinstance(pos, int):
        raise TypeError("Position must be an integer!")
    if pos < 0 or pos >= self._size:
        raise IndexError("Cannot insert because index", pos, "is invalid!")

    if self.search(newItem):
        raise ValueError("Cannot insert because item is already in the list!")

    temp = Node(newItem)

    self._current = self._head
    self._previous = None.
    for count in range(pos):
        self._previous = self._current
        self._current = self._current.getNext()

    temp.setNext(self._current)
    if self._current == self._head:
        self._head = temp
unordered_linked_list.py

else:
    self._previous.setNext(temp)
    self._current = None
    self._size += 1

def pop(self, pos = None):
    ""
    Removes and returns the item at position pos of the list.
    Precondition: position pos exists in the list.
    Postcondition: Removes and returns the item at position pos of
    the list.
    ""
    if pos == None:
        pos = self._size - 1
    if not isinstance(pos, int):
        raise TypeError("Position must be an integer!")
    if pos >= self._size or pos < 0:
        raise IndexError("Cannot pop from index", pos, "-- invalid
    index!")

    self._current = self._head
    self._previous = None
    for count in range(pos):
        self._previous = self._current
        self._current = self._current.getNext()

    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())
    returnVal = self._current.getData()
    self._current = None
    self._size -= 1
    return returnVal

def __str__(self):
    ""
    Removes and returns the item at position pos of the list.
    Precondition: position pos exists in the list.
    Postcondition: Removes and returns the item at position pos of
    the list.
    ""
    resultStr = "(head)"
    current = self._head
    while current != None:
        resultStr += " " + str(current.getData())
        current = current.getNext()
unordered_linked_list.py

return resultStr + " (tail)"

def __iter__(self):
    """ Iterates though the list from the head to the tail and yields up the data in each Node. """
    current = self._head
    while current != None:
        yield current.getData()
        current = current.getNext()