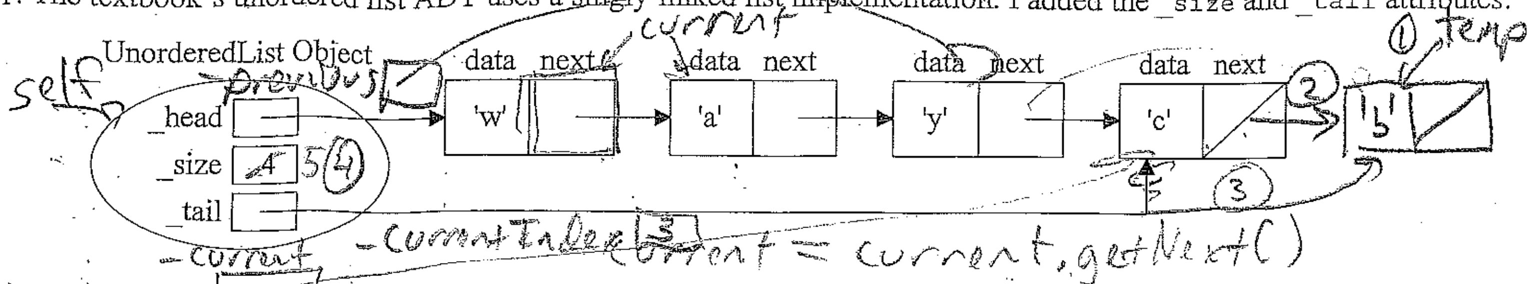
Data Structures (CS 1520) Lecture 8 Name: ______

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:

```
class UnorderedList:

    def search(self, targetItem):
        if self._current != None and self._current.getData() == targetItem:
            return True
        self._current = self._head
        self._previous = None
        self._currentIndex = 0
        while self._current != None:
            if self._current.getData() == targetItem:
                return True
            else:
                self._previous = self._current
                self._current = current.getNext()
                self._currentIndex += 1
        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?

item to add is not already in the list

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

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

d) Why do you suppose I added a `_tail` attribute? Make append O(1) instead of O(n)

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")
```

```
""" 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!")
```