Chapter-7:

R-7.1 Give an algorithm for finding the second-to-last node in a singly linked list in which the last node is indicated by a next reference of None.

**In a function. Firstly the algorithm is going to check that the list is larger then a length of 1. Otherwise there is no second-to-last node to find. Then in the function there will be a variable called *CURRENT* that points to the head node of the linked list. Then in a loop, there will be an if statement that checks the next value of the node that current points to. If it is equal to NONE then the current node is the second to last. In this case simply return the element at the current node and break from the loop. If the next nodes pointer does not point to NONE then the loop will traverse the current variable to point to the next node in the list. This way the algorithm easily keeps track of the current node which will be the second to last without using too much memory. Time complexity is O(N).**

def second\_to\_last(self):  
 if self.\_size > 1:  
 node = self.\_head  
 while node.next.next != None:  
 node = node.next  
   
 return node

R-7.3 Describe a recursive algorithm that counts the number of nodes in a singly linked list.

**There will be a class variable that stores the value for the length of the list. And then in a function a node will be passed to the function (the first call of the function will pass the head). This function will increment the class variable by 1 to count the node. And then in an if statement there will be a check of the nodes next value. If it equals NONE then nothing will happen but if it doesn’t equal NONE, then there will be a recursive call of the function that passes the next node. This iterates down the list counting each individual node until NONE is found.**

R-7.6 Suppose that x and y are references to nodes of circularly linked lists, although not necessarily the same list. Describe a fast algorithm for telling if x and y belong to the same list.

**Assuming that the circular lists are singly linked lists. Pick either X or Y and use a loop to iterate through that linked list. If the other variable (X or Y) is found then they are part of the same list. If the loop iterates completely around to the original starting node then they are part of 2 different lists.**

**If the circularly linked lists are doubly linked lists. Then you can pick X or Y and use a loop to iterate both forwards and backwards down the lists. The loop will go until the pointers going both ways end to equal the same address (meaning they point at the same node). At this point it can be concluded that X and Y are nodes in different lists. If X or Y is found (depending on which list you decided to check) is found in that list then just break from the loop and conclude that X and Y are nodes in the same list.**

**You could also just add an identifier for every class object. So for example every list could have a variable in the class that acts as a unique identifier. With this all you’d have to do is access that variable from X and Y and check if that variable is the same for both.**

C-7.28 Describe a fast recursive algorithm for reversing a singly linked list.

**All you really have to do is reverse the pointer of every node in a list. So the next variable of a node will point to the previous node instead of the next. This can be done individually.**

**Firstly a variable will be made to point to the head nodes next. And then the head nodes next will be changed to point to NONE. Then a function will be called where the head is passed as the first node and the variable that was made will pass the second node. This second node is what will be used to pass along the next node to keep changing the direction of the list. In the function there will be a check to see if the second node that’s passed ever points to NONE. If so then there will be no more recursive calls. In the function, firstly there will be a recursive call that passes the second node as the first node and the second nodes next as the second node. Then the after that recursive call will change the second nodes next to point to the first node. What this should do is recursively call every set of two nodes until the end is reached and change the direction of the list on each node.**

***Here is some pseudocode:***

**Def Reverse\_List(self):**

**If size\_of\_list > 1:**

**Temp\_variable = self.\_head.next**

**self.\_head = NONE**

**Reverse\_Node(self.\_head, Temp\_variable):**

**temp\_variable = self.\_tail**

**self.\_tail = self.\_head**

**self.\_head = temp\_variable**

**# *This function is the what is called recursively.***

**Def Reverse\_Node(first\_node, second\_node):**

**If second\_node.next != NOINE:**

**Reverse\_Node(second\_node, second\_node.next) # *recursive call***

**Second\_node.next = first\_node *# order is switched***