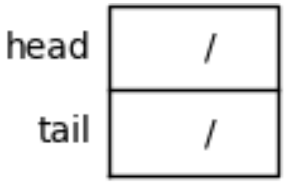
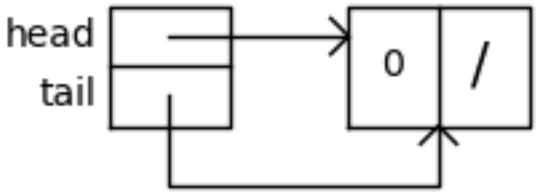
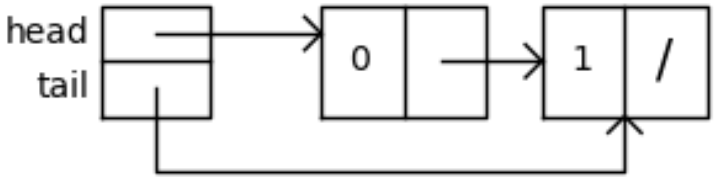
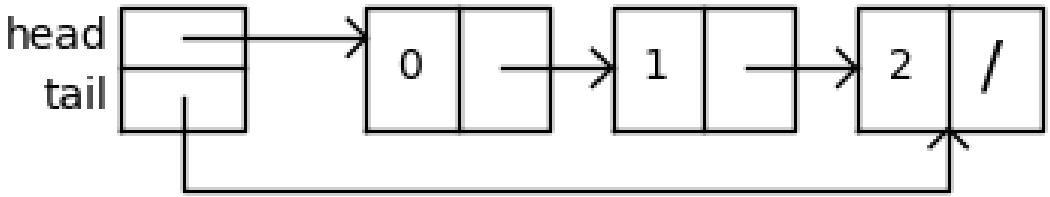
Lab II

Singly Linked List









Course: CS 2302

Section: 12:00 p.m. – 1:20 p.m.

Author: Oswaldo Escobedo

Instructor: Dr. Fuentes

TA: Harshavardhini Bagavathyraj

Introduction

The objective of this laboratory is to add nine methods to the singly_linked_list class, which Dr. Fuentes provided us in class. To create the code we must use tracing, python matplotlib and math class. The nine methods that we are going to add are the following:

- *insert(i,x)* - This method is intended to add a node with item x in the index i.

- *remove(x)* - This method is intended to eliminate the first node that contains the item x. If the item is not found in the list, it will throw a RaiseValue error.

- *pop([i])* - This method is intended to remove the node at position i from the list and return the value it contains. If the user does not choose an index, the value of the parameter will be -1, which means that the last node will be removed.

- *clear()* - This method is intended to remove all nodes from the list. In other words, this method will reset the list.

- *index(x[, start[, end]])* - This method is intended to return in which position the index of value x is in a start and end range. If the value is not found, it will throw a RaiseValue error.

- *count(x)* - This method has the purpose of counting how many times the element x appears in the list.

- *sort()* - This method has the purpose of sorting the list in ascending order.

- *reverse()* - This method is intended to reverse the list in-place.

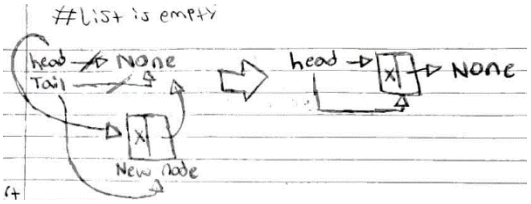
- *copy()* - This method has the purpose of copying all the elements of a list.

Proposed Solution Design and Implementation

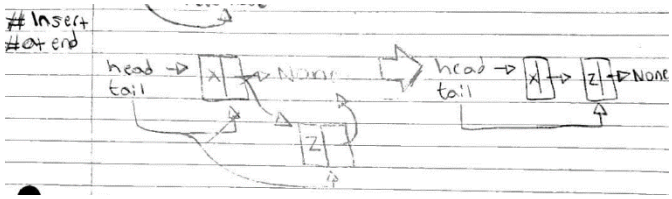
The first thing I did was understand the methods provided by Fuentes. Which was a method of appending nodes, extending and drawing a list. There is no need to explain these methods, since they are self-explanatory.

My first approach to start the laboratory was to review my Introduction to Computer Science and Data Structures and Algorithms notes, since it was in these classes where I learned and practiced this data structure. After analyzing and remembering again the behavior and algorithm of some methods of this data structure I set out to trace the methods we needed to do, in order to understand the behavior of the algorithm,, come up with an efficient solution, which base cases were needed and other stuff.

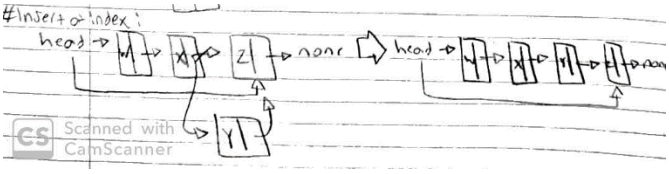
insert(i,x)



First, I started with the insertion method, which, as already mentioned, inserts / adds a node at the specified index. Therefore, to add a node in a position i, we must traverse the list and we must stop the iteration when we have reached the specified index or when we have reached the end of the list (when the pointer points to None).

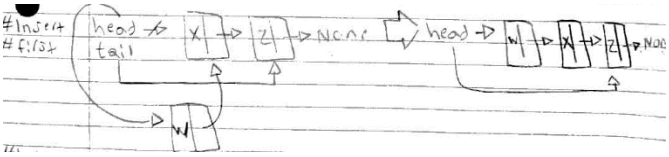


However, before we think about a solution, we must remember that this linked list has a tail, so before implementing an algorithm we have to take that into account. After analyzing the algorithm I started thinking about some base cases that were necessary to make this method work:



- If the list is empty we have to add a node and make head and tail point to that node.

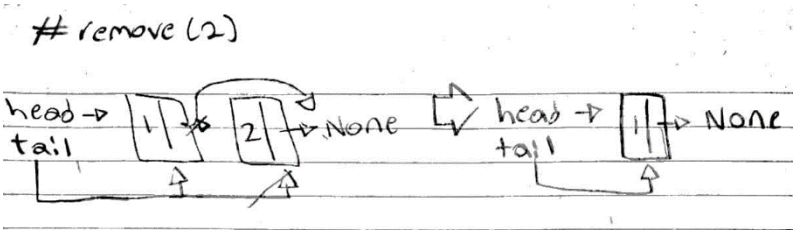
- If index is zero, then we have to create a node that will point to what the head is pointing and, in the end, we will update head to point to this new node.



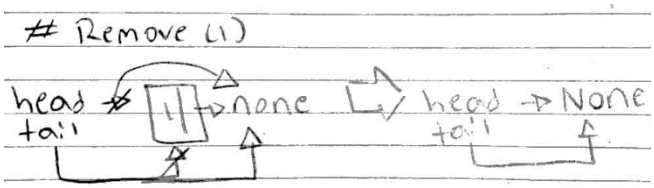
- If the node is added at the end of the list, we must update tail to point to this node.

After having this in mind I set out to do the code.

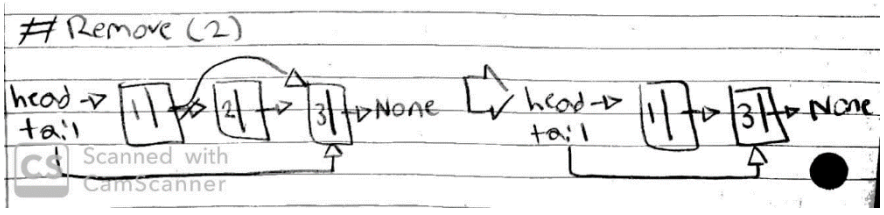
remove(x)



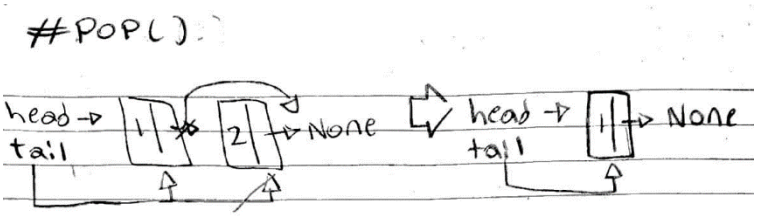
As already mentioned, this method removes the node that contains the first item x that appears in the list. It has a certain similarity with the method of insertion, only that the algorithm of this varies a little and at a certain point it becomes a bit complicated due to the fact that this list has a tail. Therefore, to remove the node we must traverse the list and stop until we have reached the item containing the item x or when we the pointer t points to None. After analyzing the algorithm I started thinking about some base cases that were necessary to make this method work:



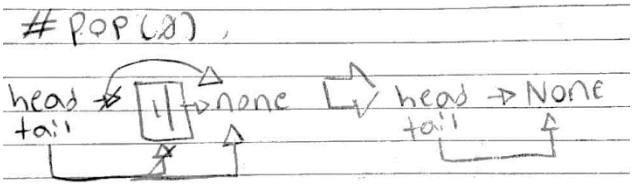
- If the list is empty, throw an exception. Specifically, it will throw a Raise Value Error, which, in other, words, means that the value was not found and the method will not remove anything.
- If the item we are looking for is in the first node of the list, then make head point to the second node. This will remove the first node.
- If we are at the end of the list and the item was not found, then throw a Raise Value error.
- If the item we are looking for is at the end of the list (meaning that we have to remove the last node), then make tail point to the node that is before the last node. This will remove the last node.



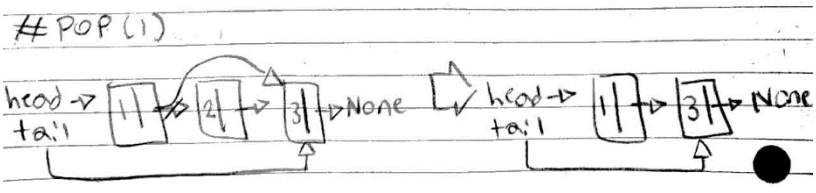
pop([i])



This was one of the last methods I did, not so much for the concept, but because its parameter contains square brackets, which at first I didn't know what it meant. At first, I thought we had to put the square brackets on the parameter of the method, but after asking tutors, classmates and researching online I realized that the square brackets only symbolized that the parameter was optional. In other words, it was up to the user whether to enter an index or not. After understanding this, I set i to have a default value of -1, this would be activated if the user called the method without an index and the last item in the linked list would appear. Moreover, the algorithm for this method will be similar to the insert method, only this time we are going to remove a node instead of adding it. After analyzing the algorithm I started thinking about some base cases that were necessary to make this method work:

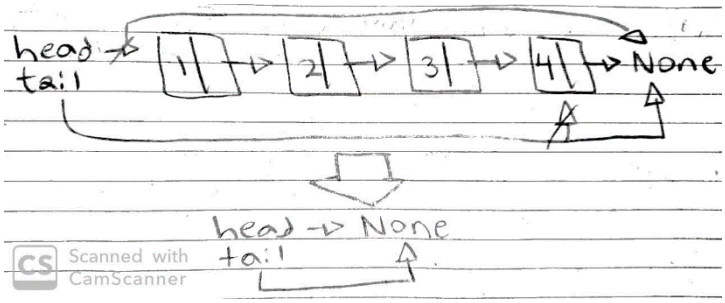


- If the list is empty, return -1 and do nothing.
- If the list has only one node, then pop it and return its data.
- If the index is zero, then make head point to the second node, and return the data of the first node of the list.
- If the user didn't entered an index, then I will be -1, meaning that we are going to pop the last element of the list. Because of this, we will make tail point to the node that is before the last node.



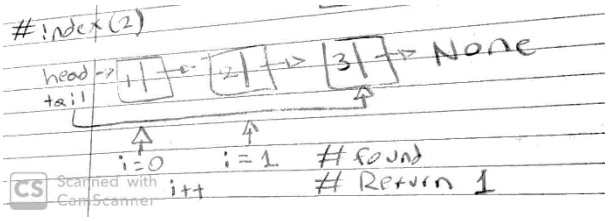
- If the user did enter an index and is the last element, pop it, update tail, and return its value.

clear()



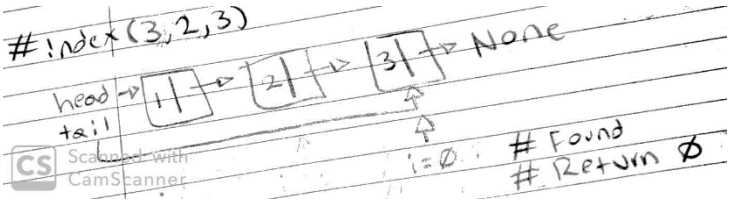
Despite how complicated this method may be, it is one of the simplest. Because, we just have to make the head and tail point to None, this will make their new reference to be None, thus causing the list to be empty. Note: we must remember that the elements that we "deleted" were not really deleted, but that the reference to these was lost, which will cause the trash collector to take action. In this particular method there will be no base cases, because it doesn't matter what tail and head are, in the end they will point to None.

index(x[, start[, end]])



This method, like the pop method, confused me for two things, 1. the notation of the parameter and 2. the purpose of the method. But as I did with the pop method, I asked several classmates and inquired online to find out the purpose of the method. After doing so, I understood that:

- x would be the item we are looking for and, start and end will be the range in which we will look for the item x.

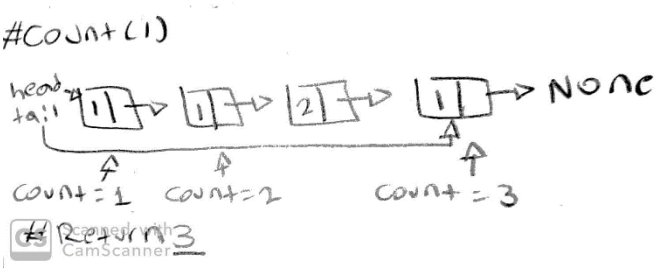


Therefore, my approach was to create the first while loop that would stop until the pointer t points to None or when we have reached the beginning. Next, I would create an index variable that would be incremented in each iteration of the second while loop, this while loop will stop until t is None or when we have reached the end or it will stop if the item x was found, this will immediately return the index in which the item was found. This will make that we search at an specified range. Moreover, we have to take into account that if the user doesn't specify a start or an end range, the default values would be start = 0 and end =

math.inf. After analyzing the algorithm I started thinking about some base cases that were necessary to make this method work:

- If the user puts that start is greater than end, it will throw a Raise Value error, because it is not a valid range.
- If the value was not found it will throw a Raise Value error.

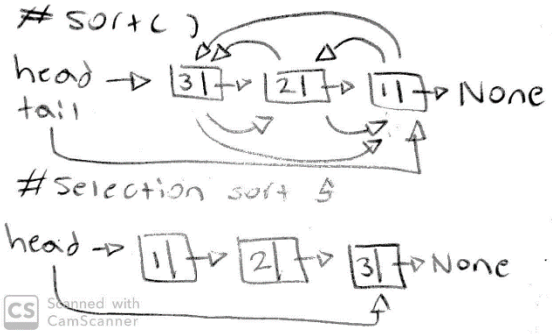
count(x)



We have already seen the algorithm of this method from Intro to Computer Science, so it is not complicated at all. The purpose of this method is to count how many times x appears in the list. Therefore, we must traverse the list until t points to None and we increment the counter by one if we found the item x. After analyzing the algorithm I started thinking about some base cases that were necessary to make this method work:

- If the list is empty, then immediately return 0.

sort()



To complete this sorting algorithm, I use selection sort, which is an in-place comparison-based algorithm that is divided into two parts, the ordered part on the left and the unordered part on the right. In order not to have to change all position nodes, all I did was compare their articles. Also, I had to use two pointers to help me, the first helps me to create a boundary of the unsorted array and the second one will help us find the minimum element of the unsorted part of the array. After analyzing the algorithm I started thinking about some base cases that were necessary to make this method work:

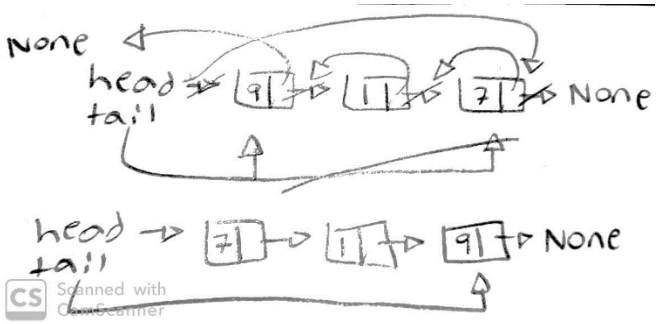
- We have to at least have two nodes to circumvent the list, otherwise the method will return immediately.

reverse()

My first approach to solving this method was the following:

First, it would create an empty list and then proceed to traverse through the original list.

Second, within the while loop, the head reference with the current node would continually change, this would reverse the list.

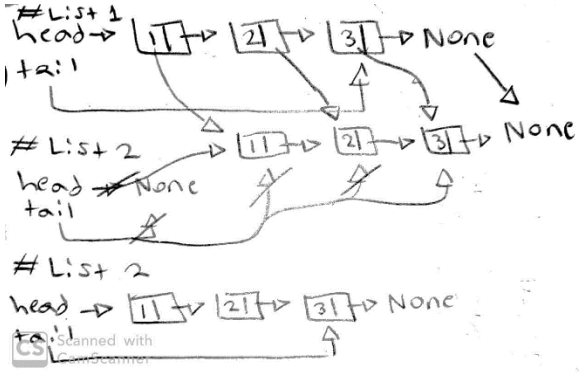


However, there is a better and creative way to improve this algorithm. Now, instead of creating an empty list and continually changing what the head points, we will reverse the direction of the pointers. In other words, instead of the pointers pointing to the right, we will make the pointers point to the opposite side. Moreover, we must create a variable that has a reference to what is head, since at the end of the algorithm this variable will become our new tail and we must also update and set head to be the last node we visited in the while loop. After analyzing the algorithm I started thinking about some base cases that were necessary to make this method work:

- If the list is empty then return.

copy()

For this method we must create a list, in which the nodes of the original list will be added. First, we will create our copy of the list (which will be empty), then we will traverse the original list until the pointer is none and, finally, we will add the nodes of the original list to that of the copy. It should be noted that in the first iteration, head and tail must be updated.

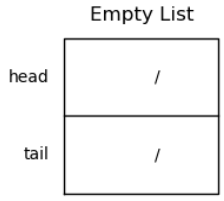


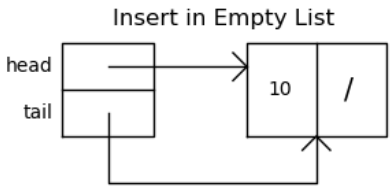
Experimental Results

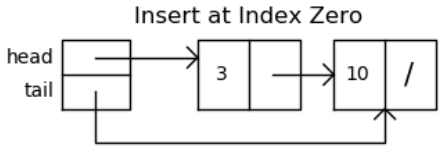
Now that we have finished writing the code, we must test the results of the methods to know if they work correctly or not. To do this, we will test the base and edge cases of the methods.

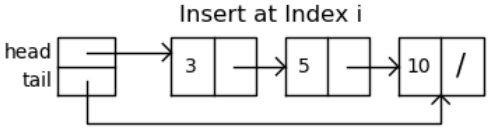
insert(i,x)

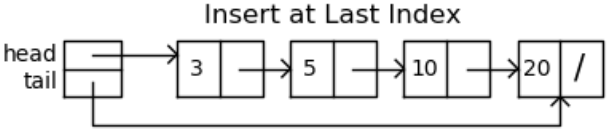
```
244 if __name__ == "__main__":
245
246     plt.close('all')
247
248     L = List()
249     L.draw('Empty List')
250
251     L.insert(10,10)
252     L.draw('Insert in Empty List')
253
254     L.insert(0,3)
255     L.draw('Insert at Index Zero')
256
257     L.insert(1,5)
258     L.draw('Insert at Index i')
259
260     L.insert(10,20)
261     L.draw('Insert at Last Index')
262
```







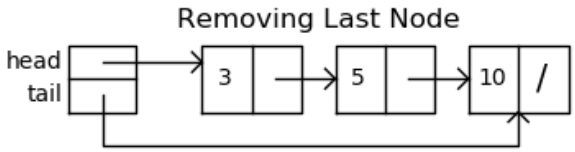


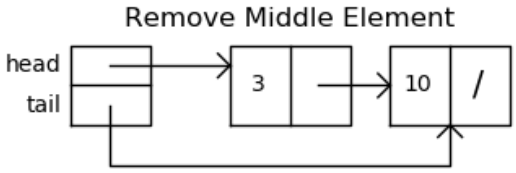


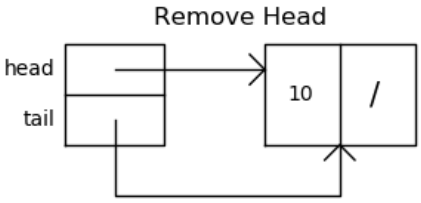
remove(x)

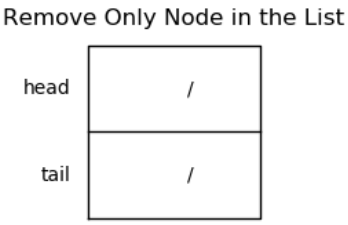
Now, we are going to remove those elements we added on the previous method and see if it works.

```
262
263     L.remove(20)
264     L.draw('Removing Last Node')
265
266     L.remove(5)
267     L.draw('Remove Middle Element')
268
269     L.remove(3)
270     L.draw('Remove Head')
271
272     L.remove(10)
273     L.draw('Remove Only Node in the List')
274
275     L.remove(30)
276     # Throws a RaiseValue Error Exception
```





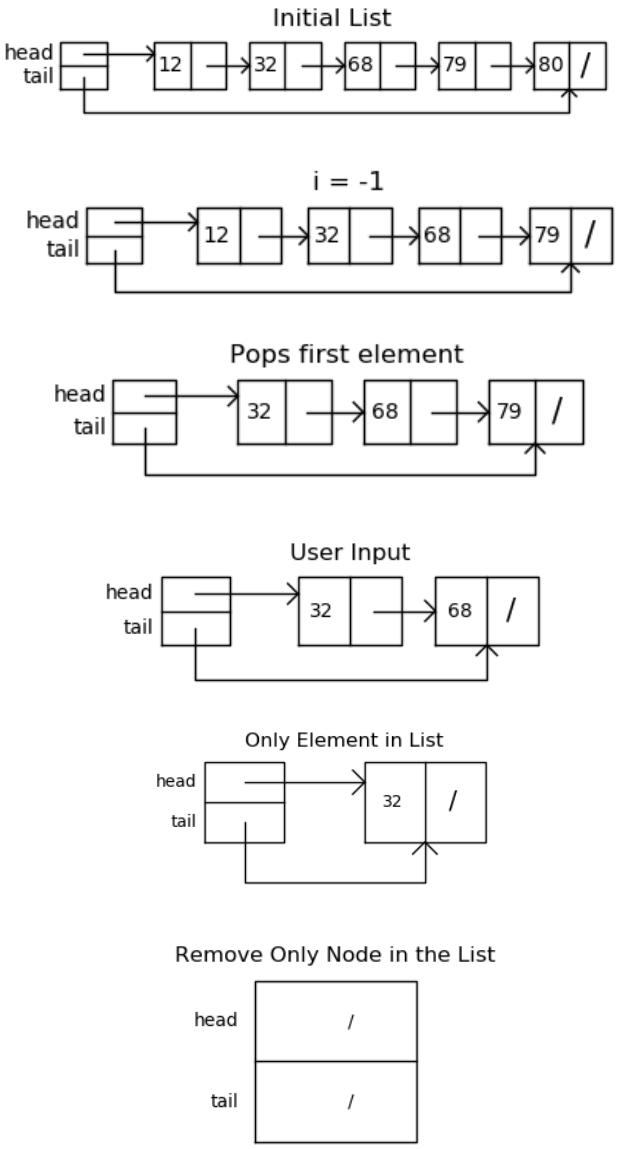





```
raise ValueError

ValueError
```

pop([i])

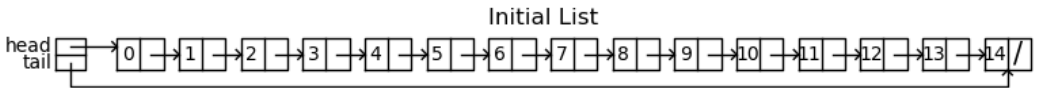
```
278    L.clear()
279
280    L.extend([12,32,68,79,80])
281    L.draw('Initial List')
282
283    # Pops last element
284    print(L.pop())
285    L.draw('i = -1')
286
287    print(L.pop(0))
288    L.draw('Pops first element')
289
290    # Pops user input index
291    print(L.pop(2))
292    L.draw('User Input')
293
294    # Pops only element in the list
295    print(L.pop(10))
296    L.draw('Only Element in List')
297
298    print(L.pop(10))
299
300    # Returns a -1, because the list is empty
301    print(L.pop(10))
302    L.draw('Empty List')
```



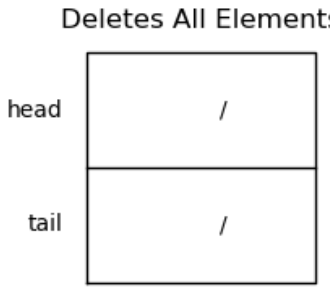
Console:

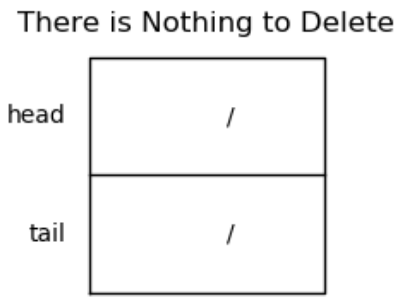
```
80
12
79
68
32
-1
```

clear()

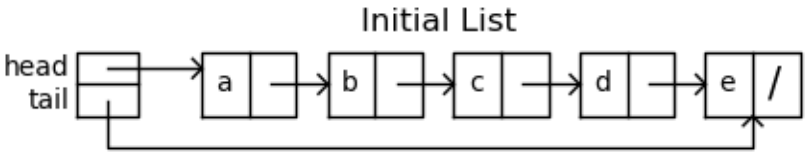



```
304  for i in range(15):
305      L.append(i)
306  L.draw('Initial List')
307  L.clear()
308  L.draw('Deletes All Elements')
309
310  L.clear()
311  L.draw('There is Nothing to Delete')
312
```





index(x, start, end)

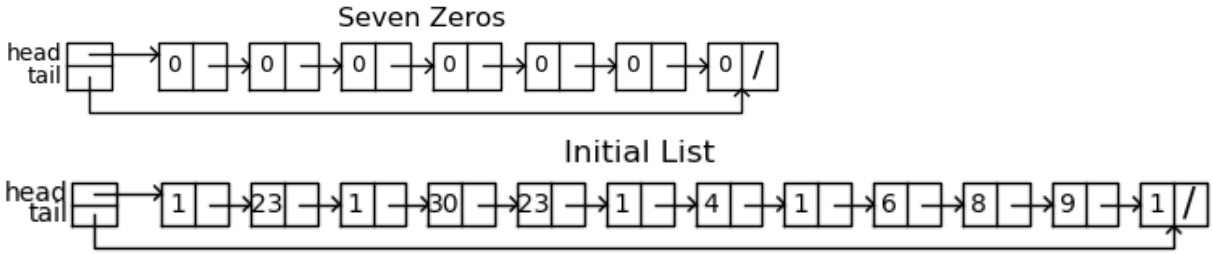



```
314  L.extend(['a','b','c','d','e'])
315  L.draw('Initial List')
316
317  # Returns the position of a from 0 to math.inf
318  print(L.index('a'))
319  # Returns the position of c from 0 to math.inf
320  print(L.index('c'))
321  # Returns the position of b from 1 to math.inf
322  print(L.index('b',1))
323  # Item was not found, throws a RaiseValue Exc.
324  print(L.index('f'))
325  #returns the position of d from 2 to 4
326  print(L.index('d',2,4))
```

Console:

```
0
2
0
    raise ValueError

ValueError
1
```

count(x)



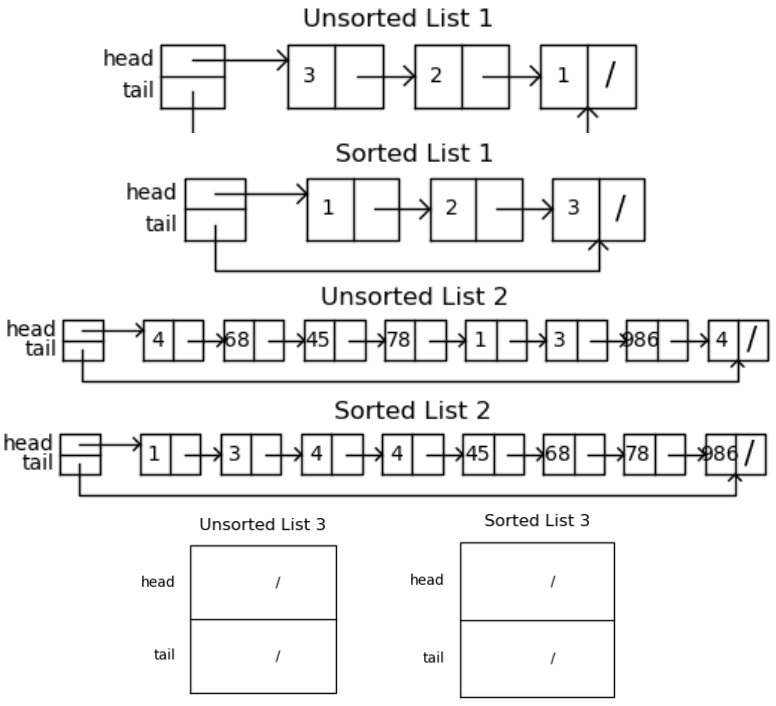

```
329
330 for i in range(7):
331     L.append(0)
332 L.draw('Seven Zeros')
333
334 # Counts how many times 0 appears in the List
335 print('\n\n\n\n')
336 print(L.count(0))
337 # Counts how many times 1 appears in the List
338 print(L.count(1))
339
340 L.clear()
341 L.extend([1,23,1,30,23,1,4,1,6,8,9,1])
342 L.draw('Initial List')
343
344 # Counts how many times 1 appears in the list
345 print(L.count(1))
```

Console:

```
7
0
5
```

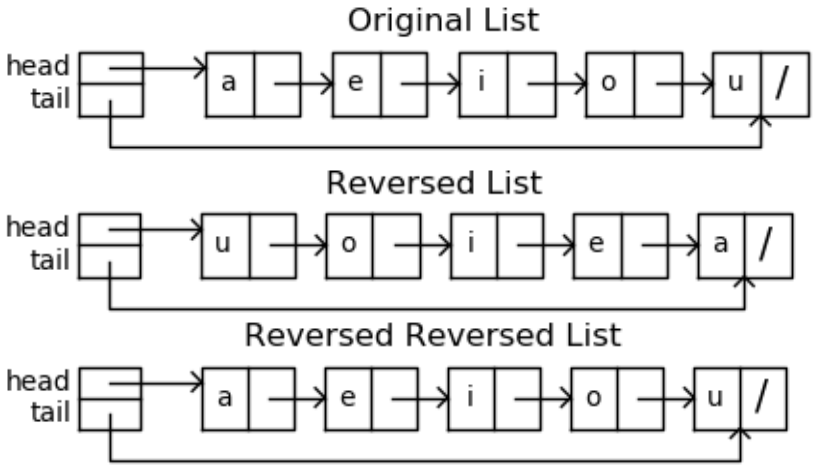
sort()

```
L.clear()
L.extend([3,2,1])
L.draw('Unsorted List 1')
L.sort()
L.draw('Sorted List 1')

L.clear()
L.extend([4,68,45,78,1,3,986,4])
L.draw('Unsorted List 2')
L.sort()
L.draw('Sorted List 2')

L.clear()
L.draw('Unsorted List 3')
L.sort()
L.draw('Sorted List 3')
```



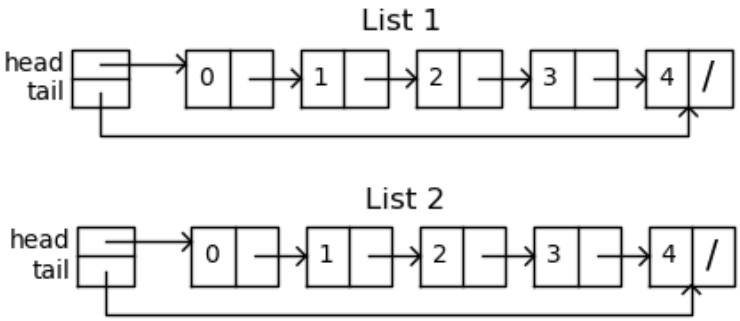
reverse()

```
L.clear()
L.extend(['a','e','i','o','u'])

L.draw('Original List')
L.reverse()

L.draw('Reversed List')
# We reverse the reversed list to return
# to original list
L.reverse()
L.draw('Reversed Reversed List')
```



copy()

```
L1 = List()
L1.extend([0,1,2,3,4])
L1.draw('List 1')
L2 = L1.copy()
L2.draw('List 2')
```



Conclusion

In conclusion, this laboratory helped me to practice and reinforce my knowledge and understanding about this data structure. In addition, it also helped me to understand why this structure is one of the most important in computer science and these reasons are the following: with a linked list we can add an endless number of nodes (which contain data) and we can increase or decrease the size of the list much more easily than in an array. For example, if we have a list that contains 100 nodes and we use the clear () method, we can delete the entire list in O (1), while in an array we would have to create a new array with size n+1, to do this process you would spend a lot of memory. On the other hand, to solve these methods we had to draw the algorithm of the methods so that we understand how the algorithm really works and thus produce an efficient algorithm.

Appendix

```
#   Course: CS 2302
#   Assignment: Lab III
#   Author: Oswaldo Escobedo
#   Instructor: Dr. Fuentes
#   TA: Harshavardhini Bagavathyraj
#   Date of Last Modification: 02/28/2020
#   Purpose of the Program: to implement functions
#   to the List Class.

import matplotlib.pyplot as plt
import numpy as np
import math


class ListNode:
    # Constructor
    def __init__(self, data, next=None):
        self.data = data
        self.next = next


class List:
    # Constructor
```

```
    def __init__(self,head = None,tail = None):
        self.head = head
        self.tail = tail

    def print(self):
        t = self.head
        while t is not None:
            print(t.data,end=' ')
            t = t.next
        print()

#%% Lab 3

    def append(self,x):
        if self.head is None: #List is empty
            self.head = ListNode(x)
            self.tail = self.head
        else:
            self.tail.next = ListNode(x)
            self.tail = self.tail.next

    def extend(self,python_list):
        for d in python_list:
```

```
            self.append(d)


    def insert(self,i,x):
        if self.head is None: # List is empty
            self.append(x)
            return
        if i == 0: # Inserts a node at the beginning of the list
            self.head = ListNode(x,self.head)
            if self.head.next is None: # Updates tail
                self.tail = self.head.next
            return
        t = self.head
        pos = 0
        while t.next is not None and pos < i - 1: # Traverses list until t is None and
            pos += 1                    # pos is less than the index
            t = t.next
        t.next = ListNode(x,t.next) # Inserts Node
        if t.next.next is None: # If the inserted node was the last one, then update tail
            self.tail = t.next


    def remove(self,x):
        if self.head is None: # List is empty
            raise ValueError
```

```
    if self.head.data == x: # Item is at the beginning of the list
        self.head = self.head.next # Updates head
        if self.head is None:
            self.tail = None
        return
    t = self.head
    while t.next is not None:
        if t.next.data == x: # If the current node has the item we are looking for,
            break         # then break
        t = t.next
    if t.next is None: # Item was not in the list
        raise ValueError
    else:
        next_node = t.next.next
        t.next = next_node # Removes the node containing x
        if next_node is None: # If the node we are removing is the last,
            self.tail = t    # then update tail


def pop(self,i=-1): # If no parameter was given then set i to -1
    if self.head is None: # List is empty
        return -1
    if self.head.next is None: # Only one node in list, therefore we must pop it
        item = self.head.data # Stores the node's data before pop it
```

```
        self.head = None
        self.tail = None
        return item
    if i == 0: # Pops the first node in the list
        item = self.head.data
        self.head = self.head.next
        return item
    t = self.head
    if i < 0: # i == -1, therefore we must pop the last node in the list
        while t.next.next is not None:
            t = t.next
        item = t.next.data
        self.tail = t # Updates tail
        t.next = None
        return item
    else: # i is a valid index
        pos = 0
        while t.next.next is not None and pos < i - 1:
            pos += 1
            t = t.next
        item = t.next.data
        next_node = t.next.next # Pops the node
        t.next = next_node
```

```
        if next_node is None: # If the node we are removing is the last, then
            self.tail = t # update tail.
        return item

def clear(self):
    self.head = None
    self.tail = None

def index(self,x,start=0,end=math.inf):
    if start > end or self.head is None: # List is empty or start is bigger than end
        raise ValueError
    t = self.head
    i = 0
    while t is not None and i != start: # While loop that helps us traverse
        i += 1                    # until the desired start.
        t = t.next
    index = 0
    while t is not None and i != end: # While loop that stops when we have reach end.
        if x == t.data: # If the current node has the data we are looking for
            return index # then return its index
        t = t.next
        index += 1
        i += 1
```

```
    raise ValueError # Value was not in the list


def count(self,x):
    if self.head is None: # List is empty
        return 0
    t = self.head
    counter = 0
    while t is not None:
        if t.data == x: # Current node has x, so increment counter.
            counter += 1
        t = t.next
    return counter


def sort(self):
    if self.head is None or self.head.next is None: # List is empty or has one node.
        return
    currNode = self.head
    nextNode = None
    while currNode is not None: # Creates the boundary of the unsorted array
        nextNode = currNode.next
        while nextNode is not None: # Help us find the minimum element of unsorted array.
            if currNode.data > nextNode.data: # Compares curr and next nodes data.
                currNode.data,nextNode.data = nextNode.data,currNode.data # swaps node data
```

```
            nextNode = nextNode.next
        currNode = currNode.next

    def reverse(self):
        if self.head is None: # List is empty
            return
        curr_node = self.head
        prev_node = None
        next_node = next
        tail = self.head # Variable that keeps a reference to head
        while curr_node is not None:
            next_node = curr_node.next # Saves next node of the curr node in the next pointer.
            curr_node.next = prev_node # Changes the next of the curr node to prev node.
            prev_node = curr_node # Makes prev point to curr
            curr_node = next_node # Makes curr to next
        self.head = prev_node # Sets head to be the last node we reached
        self.tail = tail # Sets tail to the first element we visit. For instance, the head of the original
list.

    def copy(self):
        clone = List() # Creates an empty list
        t = self.head
        while t is not None:
```

```
            if clone.head is None: # Clone is empty
                clone.head = ListNode(t.data) # Creates a node which has the value of the original list
                clone.tail = clone.head # Updates tail
                t = t.next
            else:
                clone.append(t.data) # Copies and creates a node containing the data of the original list
                t = t.next
        return clone


#%%
    def _rectangle(self,x0,y0,dx,dy):
        # Returns the coordinates of the corners of a rectangle
        # with bottom-left corner (x0,y0), dx width and dy height
        x = [x0,x0+dx,x0+dx,x0,x0]
        y = [y0,y0,y0+dy,y0+dy,y0]
        return x,y


    def draw(self,figure_name=' '):
        # Assumes the list contains no loops
        fig, ax = plt.subplots()
        x, y = self._rectangle(0,0,20,20)
        ax.plot(x,y,linewidth=1,color='k')
        ax.plot([0,20],[10,10],linewidth=1,color='k')
```

```
ax.text(-2,15, 'head', size=10,ha="right", va="center")
ax.text(-2,5, 'tail', size=10,ha="right", va="center")
t = self.head
x0 = 40
while t !=None:
    x, y = self._rectangle(x0,0,30,20)
    ax.plot(x,y,linewidth=1,color='k')
    ax.plot([x0+15,x0+15],[0,20],linewidth=1,color='k')
    ax.text(x0+7,10, str(t.data), size=10,ha="center", va="center")
    if t.next == None:
        ax.text(x0+22,10, '/', size=15,ha="center", va="center")
    else:
        ax.plot([x0+22,x0+40],[10,10],linewidth=1,color='k')
        ax.plot([x0+37,x0+40,x0+37],[7,10,13],linewidth=1,color='k')
    t = t.next
    x0 = x0+40
if self.head == None:
    ax.text(12,15, '/', size=10,ha="center", va="center")
else:
    ax.plot([10,40],[15,15],linewidth=1,color='k')
    ax.plot([37,40,37],[12,15,18],linewidth=1,color='k')

if self.tail == None:
```

```
    ax.text(12,5, '/', size=10,ha="center", va="center")
else:
    xt = 40
    t = self.head
    while t!= self.tail:
        t = t.next
        xt+=40
    ax.plot([10,10,xt+15,xt+15],[5,-10,-10,0],linewidth=1,color='k')
    ax.plot([xt+12,xt+15,xt+18],[-3,0,-3],linewidth=1,color='k')

ax.set_title(figure_name)
ax.set_aspect(1.0)
ax.axis('off')
fig.set_size_inches(1.2*(x0+200)/fig.get_dpi(),100/fig.get_dpi())
plt.show()
```