



**PANGASINAN STATE UNIVERSITY**

URDANETA CITY CAMPUS

SECOND SEMESTER, AY 2021-2022

LABORATORY

REPORT

CpE222 SOFTWARE DESIGN

**SUBMITTED BY**

**Errol E. Lopez**

**SUBMITTED TO**

**Engr. Jay-Ar M. Pentecostes**

**TABLE OF CONTENTS**

**1 p.1**

## PreLab

**2 p.2**

## InLab

**3 p.3**

## PostLab

**Laboratory Activity 03**

1. PRELAB
   1. **Python DS**
      * Chapters 7 ,8 ,10 ,12 : Ref Lab3 Fundamentals Of Python Data Structures , Kenneth A. Lambert

In **Chapter 7**, which is **Stacks**, we observed and learned that A stack is a linear data structure that follows the principle **of Last in First Out (LIFO)**. This means the last element inserted inside the stack is removed first. You can think of the stack data structure as a pile of plates on top of another.

We can implement a stack in any programming language like C, C++, Java, Python, or C#, but the specification is pretty much the same.

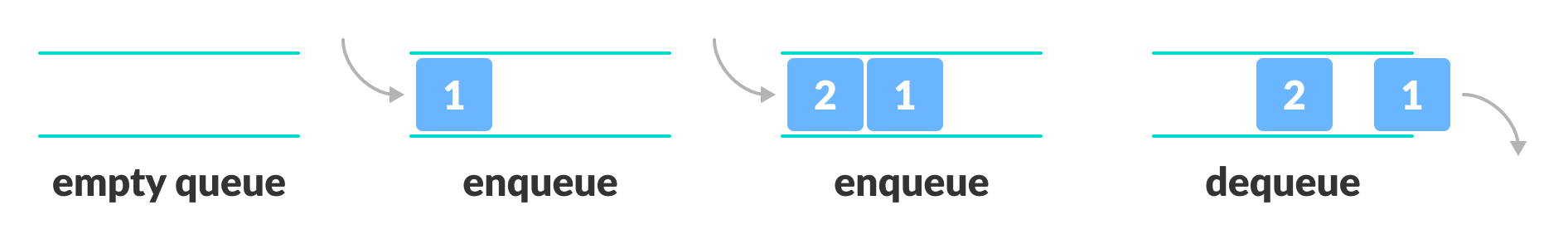
There are some basic operations that allow us to perform different actions on a stack.

* **Push**: Add an element to the top of a stack.
* **Pop**: Remove an element from the top of a stack
* **IsEmpty**: Check if the stack is empty
* **IsFull**: Check if the stack is full
* **Peek**: Get the value of the top element without removing it.

The most common stack implementation is using arrays, but it can also be implemented using lists.

The **Stack-Time Complexity** For the array-based implementation of a stack, the push and pop operations take constant time, i.e., **O(1).**

In **Chapter 8**, **Queues**, We Observe That a Queue, Like Stacks, Queues are linear collections. However, with queues, insertions are restricted to one end, called the rear, and removals to the other end, called the front. A queue thus supports a **first-in, first-out (FIFO)** protocol.

[](https://www.programiz.com/dsa/queue)

In the above image, since 1 was kept in the queue before 2, it is the first to be removed from the queue as well. It follows the **FIFO** rule.

We can implement the queue in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

A queue is an object (an abstract data structure - ADT) that allows the following operations:

* **add**: Add an element to the end of the queue
* **Pop**: Remove an element from the front of the queue
* **IsEmpty**: Check if the queue is empty
* **IsFull**: Check if the queue is full
* **Peek**: Get the value of the front of the queue without removing it
* **Clear**: make the stack empty

Most queues in computer science involve scheduling access to shared resources. The following list describes some examples:

• **CPU access**—Processes are queued for access to a shared CPU.

• **Disk access**—Processes are queued for access to a shared secondary storage device.

• **Printer access**—Print jobs are queued for access to a shared laser printer.

In **Chapter 10,** We observe that a **tree,** In the linear data structures you've explored thus far, all items except the first have a separate predecessor, and all items except the last have a distinct successor. In a tree, the concepts of predecessor and successor are substituted by parent and child.

Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Tree traversed in Python

Linear data structures like arrays, [stacks](https://www.programiz.com/data-structures/stack), [queues](https://www.programiz.com/data-structures/queue), and [linked list](https://www.programiz.com/data-structures/linked-list) have only one way to read the data. But a hierarchical data structure like a [tree](https://www.programiz.com/data-structures/trees) can be traversed in different ways.

In the **Chapter 12,** I Observe that It starts with a definition of several graph concepts. The two most popular graph representations are adjacency matrix and adjacency list. It then goes over a number of well-known and widely used graph-based algorithms. Among the algorithms of particular interest are graph traversals, minimum spanning trees, topologicsal sorting, and shortest-path issues. The chapter ends with a case study and the introduction of a graph class.

**Conclusion:**

I therefore conclude that a data structure is a data format for arranging and storing data so that any user may easily access and deal with relevant data to execute a program efficiently. Data structure is the method of logically or mathematically arranging computer memory data.

**1.2 Python DS**

* Chapters 7, 10 : Ref Lab3 Data Structures and Algorithms in Python Michael T. Goodrich, Roberto Tamassia et.al

In this chapter, which introduces the **linked list**, we observe and learn that a linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in form of a pointer. Python does not have linked lists in its standard library. We implement the concept of linked lists using the concept of nodes. linked list is a data structure that has many uses and instead using an array-based algorithms such as list we can use linked list. Array based algorithm or sequences and linked list are both that keeps elements in a certain order, but also in a very different approach. An array is more concentrated representation since it uses a single large chunk of memory that can handle references to several elements. In contrast, a linked list employs a more dispersed form in which each element is allocated a lightweight object known as a **node**. To reflect the sequence's linear order, each node keeps a reference to its element as well as one or more references to surrounding nodes.

One of the differences of an array-based sequence and linked list is their constant time complexity. An array-based algorithm has a linear time complexity of O(n) while linked list has a time complexity of O(1),

Conclusion:

I therefore conclude that a linked list can be simply created.it has a lot of benefits such as easy removal and insertion of items as the memory does not need to reorganized and the fact that items do not need to be stored contiguously in memory. It also has some drawbacks that linked list requires more memory to store the elements than an array and we observe that it is very difficult to traverse the nodes in a linked list because we cannot access randomly to any one node.

**1.3 Answers to Questions**

* Ref Lab3 Data Structures and Algorithms in Python Michael T. Goodrich, Roberto Tamassia et.al

Answer : R-7.1 to R-7.4 page 294

***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.*

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
# Function to find the second last  
# node of the linked list  
def findSecondLastNode(head):  
 temp = head  
 # If the list is empty or  
 # contains less than 2 nodes  
 if (temp == None or temp.next == None):  
 return -1  
 # Traverse the linked list  
 while (temp != None):  
 # Check if the current node is the  
 # second last node or not  
 if (temp.next.next == None):  
 return temp.data  
 # If not then move to the next node  
 temp = temp.next  
# Function to push node at head  
def push(head, new\_data):  
 new\_node = Node(new\_data)  
 # new\_node.data = new\_data  
 new\_node.next = head  
 head = new\_node  
 return head  
# Driver code  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Start with the empty list  
 head = None  
 # Use push() function to construct  
 # the below list 8 . 23 . 11 . 29 . 12  
 head = push(head, 12)  
 head = push(head, 29)  
 head = push(head, 11)  
 head = push(head, 23)  
 head = push(head, 8)  
  
 print(findSecondLastNode(head))

***R-7.2:*** *Describe a good algorithm for concatenating two singly linked lists L and M, given only references to the first node of each list, into a single list L that contains all the nodes of L followed by all the nodes of M.*

To merge two singly linked lists L and M into a single list L' that contains everything, traverse the first list to find the nodes of L, then the nodes of M, and lastly the nodes of M. Set the final node's next property to the top position of the second list.

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

*For the recursive call, a strictly smaller linked list is employed (containing one fewer node). Assuming length(l. next) computes the length of the list after the first node and returns a result one greater, the length of the whole list is computed.*

***R-7.4****: Describe in detail how to swap two nodes x and y (and not just their contents) in a singly linked list L given references only to x and y. Repeat this exercise for the case when L is a doubly linked list. Which algorithm takes more time?*

2.INLAB

Objectives

* To learn to implement different types algorithms
* To understand different types of algorithms

Implementation of Single / Circular / Doubly Linked List

*a. Write Python programs for the following operations on Single / Circular / Doubly Linked List.*

*(i) Creation (ii) insertion (iii) deletion (iv) traversal*

**Singly Linked List**

***(i)Creating the Single Linked List Class***

*Next, we need to create a class for the Linked List. This class will contain the methods to insert, remove, traverse and sort the list. Initially, the class will only contain one member****start\_node****that will point to the starting or first node of the list. The value of****start\_node****will be set to null using the constructor since the linked list will be empty at the time of creation. The following script creates a class for the linked list.*

class LinkedList:  
 def \_\_init\_\_(self):  
 self.start\_node = None

***(ii)Inserting Items***

*Depending upon the location where you want to insert an item, there are different ways to insert items in a single linked list.*

* ***Inserting Items at the Beginning***

The simplest way to insert an item in a single linked list is to add an item at the start of the list. The following function inserts item at the start of the list. Add this function to the **LinkedList** class that we created earlier.

def insert\_at\_start(self, data):  
 new\_node = Node(data)  
 new\_node.ref = self.start\_node  
 self.start\_node = new\_node

In the script above, we create a method **insert\_at\_start(),** the method accepts one parameter, which is basically the value of the item that we want to insert. Inside the method, we simply create an object of the **Node** class and set its reference to the **start\_node** since **start\_node** was previously storing the first node, which after insertion of a new node at the start will become the second node.

Therefore, we add the reference of **start\_node** to the ref variable of the new node. Now since the **new\_node** is the first node, we set the value of the **start\_node** variable to **new\_node**.

* ***Inserting Items at the End***

def insert\_at\_end(self, data):  
 new\_node = Node(data)  
 if self.start\_node is None:  
 self.start\_node = new\_node  
 return  
 n = self.start\_node  
 while n.ref is not None:  
 n = n.ref  
 n.ref = new\_node

In the above script, we create a function **insert\_at\_end()**, which inserts the element at the end of the linked list. The value of the item that we want to insert is passed as an argument to the function. The function consists of two parts. First we check if the linked list is empty or not, if the linked list is empty, all we have to do is set the value of the **start\_node** variable to **new\_node** object.

On the other hand, if the list already contains some nodes. We initialize a variable **n** with the start node. We then iterate through all the nodes in the list using a while loop as we did in the case of **traverse\_list** function. The loop terminates when we reach the last node. We then set the reference of the last node to the newly created **new\_node**.

Add the **insert\_at\_end()** function to the **LinkedList** class.

* ***Inserting Item after another Item***

We may need to add item after another item in a single linked list. To do so, we can use the insert\_after\_item() function as defined below:

def insert\_after\_item(self, x, data):  
  
 n = self.start\_node  
 print(n.ref)  
 while n is not None:  
 if n.item == x:  
 break  
 n = n.ref  
 if n is None:  
 print("item not in the list")  
 else:  
 new\_node = Node(data)  
 new\_node.ref = n.ref  
 n.ref = new\_node

The **insert\_after\_item()** function accepts two parameters: **x** and **data**. The first parameter is the item after which you want to insert the new node while the second parameter contains the value for the new node.

We start by creating a new variable n and assigning **start\_node** variable to it. Next, we traverse through the linked list using while loop. The while loop executes until **n** becomes **None**. During each iteration, we check if the value stored in the current node is equal to the value passed by the x parameter. If the comparison returns true, we break the loop.

Next, if the item is found, the **n** variable will not be **None**. The reference of the **new\_node** is set to reference stored by **n** and the reference of **n** is set to **new\_node**. Add the **insert\_after\_item()** function to the **LinkesList** class.

***(iii)Deleting Elements***

* Deletion from the Start

Deleting an element or item from the start of the linked list is straightforward. We have to set the reference of the start\_node to the second node which we can do by simply assigning the value of the reference of the start node (which is pointing to the second node) to the start node.

def delete\_at\_start(self):  
 if self.start\_node is None:  
 print("The list has no element to delete")  
 return  
 self.start\_node = self.start\_node.ref

In the script above, we first check if the list is empty or not. If the list is empty we display the message that the list has no element to delete. Otherwise, we assign the value of the **start\_node.ref** to the **start\_node**. The **start\_node** will now point towards the second element. Add the **delete\_at\_start()** function to the **LinkedList** class.

* Deletion by Item Value

To delete the element by value, we first have to find the node that contains the item with the specified value and then delete the node. Finding the item with the specified value is pretty similar to searching the item. Once the item to be deleted is found, the reference of the node before the item is set to the node that exists after the item being deleted.

def delete\_element\_by\_value(self, x):  
 if self.start\_node is None:  
 print("The list has no element to delete")  
 return  
  
 # Deleting first node  
 if self.start\_node.item == x:  
 self.start\_node = self.start\_node.ref  
 return  
  
 n = self.start\_node  
 while n.ref is not None:  
 if n.ref.item == x:  
 break  
 n = n.ref  
  
 if n.ref is None:  
 print("item not found in the list")  
 else:  
 n.ref = n.ref.ref

In the script above, we first check if the list is empty. Next, we check if the element to be deleted is located at the start of the linked list. If the element is found at the start, we delete it by setting the first node to the reference of the first node (which basically refers to the second node).

Finally, if the element is not found at the first index, we iterate through the linked list and check if the value of the node being iterated is equal to the value to be deleted. If the comparison returns true, we set reference of the previous node to the node which exists after the node which is being deleted.

***(iv)Traversing Linked List***

def traverse\_list(self):  
 if self.start\_node is None:  
 print("List has no element")  
 return  
 else:  
 n = self.start\_node  
 while n is not None:  
 print(n.item, " ")  
 n = n.ref

Let's see what is happening in the above function. The function has two main parts. First, it checks whether the linked list is empty or not. The following code checks that:

if self.start\_node is None:  
 print("List has no element")  
 return  
else:

If the linked list is empty, that means there is no item to iterate. In such cases, the **traverse\_list()**function simply prints the statement that the list has no item.

Otherwise if the list has an item, the following piece of code will execute:

else:  
 n = self.start\_node  
 while n is not None:  
 print(n.item, " ")  
 n = n.ref

As we said earlier, the start variable will contain a reference to the first nodes. Therefore, we initialize a variable n with start variable. Next, we execute a loop that executes until n becomes none. Inside the loop, we print the item stored at the current node and then set the value of n variable to n.ref, which contains the reference to the next node. The reference of the last node is None since there is no node after that. Therefore, when n becomes None, the loop terminates.

**Doubly Linked List**

***(i)Creating***

As always, let's first create a class for the single node in the list. Add the following code to your file:

class Node:  
 def \_\_init\_\_(self, data):  
 self.item = data  
 self.nref = None  
 self.pref = None

You can see in the above code, we create a Node class with three member variables: item, nref, and pref. The item variable will store the actual data for the node. The nref stores the reference to the next node, while pref stores the reference to the previous node in the doubly linked list.

Next, we need to create the DoublyLinkedList class, which contains different doubly linked list related functions. Add the following code:

class DoublyLinkedList:  
 def \_\_init\_\_(self):  
 self.start\_node = None

***(ii)Inserting***

The easiest way to insert an item in a doubly linked list is to insert an item in the empty list. The following script inserts an element at the start of the doubly linked list:

def insert\_in\_emptylist(self, data):  
 if self.start\_node is None:  
 new\_node = Node(data)  
 self.start\_node = new\_node  
 else:  
 print("list is not empty")

In the script above, we define a method insert\_in\_emptylist(). The method first checks whether the self.start\_node variable is None or not. If the variable is None, it means that the list is actually empty. Next, a new node is created and its value is initialized by the value passed as a parameter to the data parameter of the insert\_in\_emptylist() function. Finally, the value of self.start\_node variable is set to the new node. In case if the list is not empty, a message is simply displayed to the user that the list is not empty.

Add the insert\_in\_emptylist() method to the DoublyLinkedList class that you created earlier.

***(iii)Deleting***

The easiest way to delete an element from a doubly linked list is from the start. To do so, all you have to do is set the value of the start node to the next node and then set the previous reference of the start node to None. However before we do that we need to perform two checks. First, we need to see if the list is empty. And then we have to see if the list contains only one element or not. If the list contains only one element then we can simply set the start node to None. The following script can be used to delete elements from the start of the doubly linked list.

def delete\_at\_start(self):  
 if self.start\_node is None:  
 print("The list has no element to delete")  
 return  
 if self.start\_node.nref is None:  
 self.start\_node = None  
 return  
 self.start\_node = self.start\_node.nref  
 self.start\_prev = None;

***(iv)Traversal***

Traversing a doubly linked list is very similar to traversing a single linked list. The script is as follows:

def traverse\_list(self):  
 if self.start\_node is None:  
 print("List has no element")  
 return  
 else:  
 n = self.start\_node  
 while n is not None:  
 print(n.item, " ")  
 n = n.nref

**Circular Linked List**

***(i)Creating***

To create a circular linked list, we create two classes: the first one for nodes and the second one for the linked list that will use the nodes.

For the node class, we have two members. One to store data and the other to store the link to the next node. The class definition will be:

class Node:  
 def \_\_init\_\_(self, data = None):  
 self.data = data  
 self.next = self

So, initially, every new node created will either have or won’t have a data value depending on how it was created, but it will point to itself by default so that it’s like a single item circular linked list.

***(ii)Insertion***

Nodes can either be appended or inserted at a specified position in this implementation. To append, we simply call the insert method and send the size of the list as the index.

In the insert method, we first check if the specified index is valid or not, if not, we throw a ValueError. After passing the check, if the list is empty, we simply assign the new node to the head, increment the count, and return from the method.

If the list isn’t empty, we first reach the node before the specified index. For example, if the given index is 5, then we reach the node at the 4th index, and because the list is circular, if the given index is 0, then we reach the last node of the list.

Now, we assign the new node to the next of the node before the specified index, and we make the new node’s next link to the node at the specified index. This will make sure that the new node is inserted before the node that was at the specified index, and hence taken its index and pushed it ahead.

Now, if the given index was 0, we have inserted a node after the last node of the list, so we simply make the head point to the new node making it the new head of the list.

def insert(self, data, index):  
 if (index > self.count) | (index < 0):  
 raise ValueError(f"Index out of range: {index}, size: {self.count}")  
  
 if self.head == None:  
 self.head = Node(data)  
 self.count += 1  
 return  
  
 temp = self.head  
 for \_ in range(self.count - 1 if index - 1 == -1 else index - 1):  
 temp = temp.next  
  
 aftertemp = temp.next # New node goes between temp and aftertemp  
 temp.next = Node(data)  
 temp.next.next = aftertemp  
 if (index == 0):  
 self.head = temp.next  
 self.count += 1  
 return

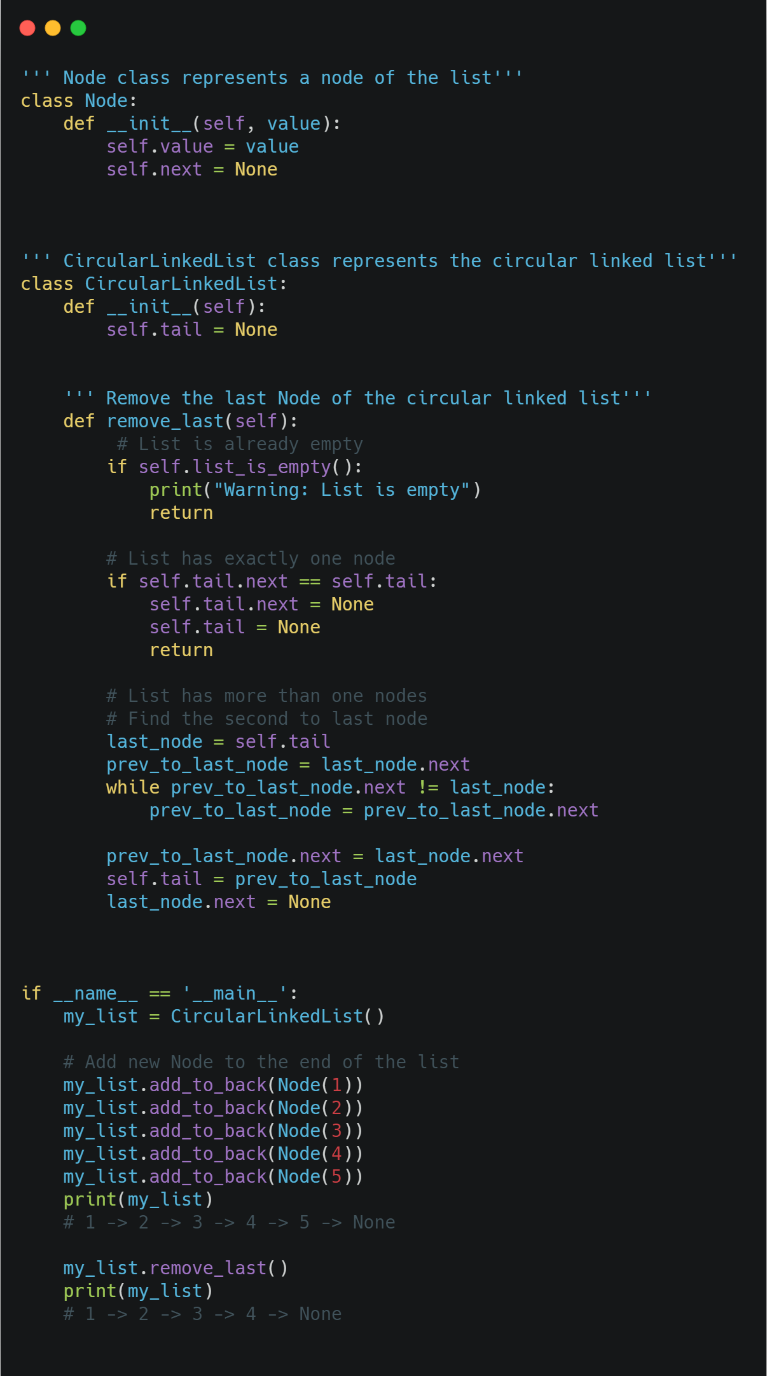
***(iii)Deletion***

def delete(self, index):  
 if (index >= self.count) | (index < 0):  
 raise ValueError(f"Index out of range: {index}, size: {self.count}")  
  
 if self.count == 1:  
 self.head = None  
 self.count = 0  
 return  
  
 before = self.head  
 for \_ in range(self.count - 1 if index - 1 == -1 else index - 1):  
 before = before.next  
 after = before.next.next  
  
 before.next = after  
 if (index == 0):  
 self.head = after  
 self.count -= 1  
 return

To remove an item we must specify where the item is to be removed from. If the specified index is out of range, we raise a ValueError. If there’s only one item on the list, we simply make the head None and the count 0, and return from the method.

Otherwise, we have to reach the node before the specified index and the node after the specified index. For example, if the specified index is 4 then we need to reach the 3rd node and the 5th node, and because the list is circular if the specified index is 0, we need to reach the last node (before it) and the 1st node (after it).

After this, we simply assign the node after the specified index to the next of the node before the specified index. This will skip the node at the specified index, hence removing it from the list. If the specified index is 0, then the head has been removed from the list, so we simply have to assign the node that was after the specified index to the head and the list will be restored. Don’t forget to decrement the count of the list.

***(iv)Traversal***

In traversing Circular Linkedlist we need to be a little careful about going through a circular linked list. That’s because, unlike the other types of linked lists we talked about, the nodes of a circular linked list are connected together forming a circle. This means there is no node with the *next*pointer to None. So it is our responsibility to define a way to stop traversal, otherwise, we will create an infinity loop. In the code below implement the dunder method \_\_str\_\_().

As we can see, in every iteration we check if the current node is the last node of the list. If it is True, we stop the iteration.

b. To store a polynomial expression in memory using Single / Circular / Doubly Linked List.

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
 self.prev = None  
  
  
class CircularDoublyLinkedList:  
 def \_\_init\_\_(self):  
 self.first = None  
  
 def get\_node(self, index):  
 current = self.first  
 for i in range(index):  
 current = current.next  
 if current == self.first:  
 return None  
 return current  
  
 def insert\_after(self, ref\_node, new\_node):  
 new\_node.prev = ref\_node  
 new\_node.next = ref\_node.next  
 new\_node.next.prev = new\_node  
 ref\_node.next = new\_node  
  
 def insert\_before(self, ref\_node, new\_node):  
 self.insert\_after(ref\_node.prev, new\_node)  
  
 def insert\_at\_end(self, new\_node):  
 if self.first is None:  
 self.first = new\_node  
 new\_node.next = new\_node  
 new\_node.prev = new\_node  
 else:  
 self.insert\_after(self.first.prev, new\_node)  
  
 def insert\_at\_beg(self, new\_node):  
 self.insert\_at\_end(new\_node)  
 self.first = new\_node  
  
 def remove(self, node):  
 if self.first.next == self.first:  
 self.first = None  
 else:  
 node.prev.next = node.next  
 node.next.prev = node.prev  
 if self.first == node:  
 self.first = node.next  
  
 def display(self):  
 if self.first is None:  
 return  
 current = self.first  
 while True:  
 print(current.data, end=' ')  
 current = current.next  
 if current == self.first:  
 break  
  
  
a\_cdllist = CircularDoublyLinkedList()  
  
print('Menu')  
print('insert <data> after <index>')  
print('insert <data> before <index>')  
print('insert <data> at beg')  
print('insert <data> at end')  
print('remove <index>')  
print('quit')  
  
while True:  
 print('The list: ', end='')  
 a\_cdllist.display()  
 print()  
 do = input('What would you like to do? ').split()  
  
 operation = do[0].strip().lower()  
  
 if operation == 'insert':  
 data = int(do[1])  
 position = do[3].strip().lower()  
 new\_node = Node(data)  
 suboperation = do[2].strip().lower()  
 if suboperation == 'at':  
 if position == 'beg':  
 a\_cdllist.insert\_at\_beg(new\_node)  
 elif position == 'end':  
 a\_cdllist.insert\_at\_end(new\_node)  
 else:  
 index = int(position)  
 ref\_node = a\_cdllist.get\_node(index)  
 if ref\_node is None:  
 print('No such index.')  
 continue  
 if suboperation == 'after':  
 a\_cdllist.insert\_after(ref\_node, new\_node)  
 elif suboperation == 'before':  
 a\_cdllist.insert\_before(ref\_node, new\_node)  
  
 elif operation == 'remove':  
 index = int(do[1])  
 node = a\_cdllist.get\_node(index)  
 if node is None:  
 print('No such index.')  
 continue  
 a\_cdllist.remove(node)  
  
 elif operation == 'quit':  
 break

*Implementation of Stack and Queue*

a. Design and implement Stack and its operations using List

class stack:  
  
 def \_\_init\_\_(self):  
  
 self.array = []  
 self.top = -1  
 self.max = 100  
  
 # Stack's member method to check  
  
 # if the stack is empty  
 def isEmpty(self):  
  
 if self.top == -1:  
 return True  
 else:  
 return False  
  
 # Stack's member method to check  
  
 # if the stack is full   
 def isFull(self):  
  
 if self.top == self.max - 1:  
 return True  
 else:  
 return False  
  
 # Stack's member method to  
  
 # insert an element to it   
  
 def push(self, data):  
  
 if self.isFull():  
 print('Stack OverFlow')  
 return  
 else:  
 self.top += 1  
 self.array.append(data)  
  
 # Stack's member method to  
  
 # remove an element from it  
 def pop(self):  
  
 if self.isEmpty():  
 print('Stack UnderFlow')  
 return  
 else:  
 self.top -= 1  
 return self.array.pop()  
  
  
# A class that supports all the stack   
# operations and one additional  
# operation getMin() that returns the  
# minimum element from stack at  
# any time. This class inherits from  
# the stack class and uses an  
# auxiliary stack that holds  
# minimum elements   
class SpecialStack(stack):  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.Min = stack()  
  
 # SpecialStack's member method to  
  
 # insert an element to it. This method  
 # makes sure that the min stack is also  
 # updated with appropriate minimum  
 # values  
 def push(self, x):  
  
 if self.isEmpty():  
 super().push(x)  
 self.Min.push(x)  
 else:  
 super().push(x)  
 y = self.Min.pop()  
 self.Min.push(y)  
 if x <= y:  
 self.Min.push(x)  
 else:  
 self.Min.push(y)  
  
 # SpecialStack's member method to   
  
 # remove an element from it. This  
 # method removes top element from  
 # min stack also.  
 def pop(self):  
  
 x = super().pop()  
 self.Min.pop()  
 return x  
  
 # SpecialStack's member method  
  
 # to get minimum element from it.  
 def getmin(self):  
  
 x = self.Min.pop()  
 self.Min.push(x)  
 return x  
  
  
# Driver code  
if \_\_name\_\_ == '\_\_main\_\_':  
 s = SpecialStack()  
 s.push(10)  
 s.push(20)  
 s.push(30)  
 print(s.getmin())  
 s.push(5)  
 print(s.getmin())

reference: <https://www.geeksforgeeks.org/design-and-implement-special-stack-data-structure/>

b. Design and implement Queue and its operations using List

# Python3 program to demonstrate linked list  
# based implementation of queue  
  
# A linked list (LL) node  
# to store a queue entry  
class Node:  
  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
  
# A class to represent a queue  
  
# The queue, front stores the front node  
# of LL and rear stores the last node of LL  
class Queue:  
  
 def \_\_init\_\_(self):  
 self.front = self.rear = None  
  
 def isEmpty(self):  
 return self.front == None  
  
 # Method to add an item to the queue  
 def EnQueue(self, item):  
 temp = Node(item)  
  
 if self.rear == None:  
 self.front = self.rear = temp  
 return  
 self.rear.next = temp  
 self.rear = temp  
  
 # Method to remove an item from queue  
 def DeQueue(self):  
  
 if self.isEmpty():  
 return  
 temp = self.front  
 self.front = temp.next  
  
 if (self.front == None):  
 self.rear = None  
  
  
# Driver Code  
if \_\_name\_\_ == '\_\_main\_\_':  
 q = Queue()  
 q.EnQueue(10)  
 q.EnQueue(20)  
 q.DeQueue()  
 q.DeQueue()  
 q.EnQueue(30)  
 q.EnQueue(40)  
 q.EnQueue(50)  
 q.DeQueue()  
 print("Queue Front " + str(q.front.data))  
 print("Queue Rear " + str(q.rear.data))

REFERENCE: <https://www.geeksforgeeks.org/queue-linked-list-implementation/>

*Implementation of Binary Search Tree*

a. Create a binary search tree.

# Binary Search in python  
def binarySearch(array, x, low, high):  
 # Repeat until the pointers low and high meet each other  
 while low <= high:  
 mid = low + (high - low)//2  
 if array[mid] == x:  
 return mid  
 elif array[mid] < x:  
 low = mid + 1  
 else:  
 high = mid - 1  
 return -1  
array = [3, 4, 5, 6, 7, 8, 9]  
x = 4  
result = binarySearch(array, x, 0, len(array)-1)  
if result != -1:  
 print("Element is present at index " + str(result))  
else:  
 print("Not found")

b. Traverse the above binary search tree recursively in pre-order, post-order and in-order.

class Node:  
 def \_\_init\_\_(self, key):  
 self.left = None  
 self.right = None  
 self.val = key  
  
  
# A function to do inorder tree traversal  
def printInorder(root):  
 if root:  
 # First recur on left child  
 printInorder(root.left)  
  
 # then print the data of node  
 print(root.val),  
  
 # now recur on right child  
 printInorder(root.right)  
  
  
# A function to do postorder tree traversal  
def printPostorder(root):  
 if root:  
 # First recur on left child  
 printPostorder(root.left)  
  
 # the recur on right child  
 printPostorder(root.right)  
  
 # now print the data of node  
 print(root.val),  
  
  
# A function to do preorder tree traversal  
def printPreorder(root):  
 if root:  
 # First print the data of node  
 print(root.val),  
  
 # Then recur on left child  
 printPreorder(root.left)  
  
 # Finally recur on right child  
 printPreorder(root.right)  
  
  
# Driver code  
root = Node(1)  
root.left = Node(2)  
root.right = Node(3)  
root.left.left = Node(4)  
root.left.right = Node(5)  
print  
"Preorder traversal of binary tree is"  
printPreorder(root)  
  
print  
"\nInorder traversal of binary tree is"  
printInorder(root)  
  
print  
"\nPostorder traversal of binary tree is"  
printPostorder(root)

c. Count the number of nodes in the binary search tree.

class BinaryTree:  
 def \_\_init\_\_(self, key=None):  
 self.key = key  
 self.left = None  
 self.right = None  
  
 def set\_root(self, key):  
 self.key = key  
  
 def inorder(self):  
 if self.left is not None:  
 self.left.inorder()  
 print(self.key, end=' ')  
 if self.right is not None:  
 self.right.inorder()  
  
 def insert\_left(self, new\_node):  
 self.left = new\_node  
  
 def insert\_right(self, new\_node):  
 self.right = new\_node  
  
 def search(self, key):  
 if self.key == key:  
 return self  
 if self.left is not None:  
 temp = self.left.search(key)  
 if temp is not None:  
 return temp  
 if self.right is not None:  
 temp = self.right.search(key)  
 return temp  
 return None  
  
  
def count\_nodes(node):  
 if node is None:  
 return 0  
 return 1 + count\_nodes(node.left) + count\_nodes(node.right)  
  
  
btree = None  
print('Menu (this assumes no duplicate keys)')  
print('insert <data> at root')  
print('insert <data> left of <data>')  
print('insert <data> right of <data>')  
print('count')  
print('quit')  
  
while True:  
 print('inorder traversal of binary tree: ', end='')  
 if btree is not None:  
 btree.inorder()  
 print()  
  
 do = input('What would you like to do? ').split()  
  
 operation = do[0].strip().lower()  
 if operation == 'insert':  
 data = int(do[1])  
 new\_node = BinaryTree(data)  
 suboperation = do[2].strip().lower()  
 if suboperation == 'at':  
 btree = new\_node  
 else:  
 position = do[4].strip().lower()  
 key = int(position)  
 ref\_node = None  
 if btree is not None:  
 ref\_node = btree.search(key)  
 if ref\_node is None:  
 print('No such key.')  
 continue  
 if suboperation == 'left':  
 ref\_node.insert\_left(new\_node)  
 elif suboperation == 'right':  
 ref\_node.insert\_right(new\_node)  
  
 elif operation == 'count':  
 print('Number of nodes in tree: {}'.format(count\_nodes(btree)))  
  
 elif operation == 'quit':  
 break

Implementation of Traversal Algorithm for Breadth first traversal

a. Create a traversal algorithm for Breadth first traversal

graph = {  
 'A' : ['B','C'],  
 'B' : ['D', 'E'],  
 'C' : ['F'],  
 'D' : [],  
 'E' : ['F'],  
 'F' : []  
}  
  
visited = [] # List to keep track of visited nodes.  
queue = [] #Initialize a queue  
  
def bfs(visited, graph, node):  
 visited.append(node)  
 queue.append(node)  
  
 while queue:  
 s = queue.pop(0)   
 print (s, end = " ")   
  
 for neighbour in graph[s]:  
 if neighbour not in visited:  
 visited.append(neighbour)  
 queue.append(neighbour)  
  
# Driver Code  
bfs(visited, graph, 'A')

Implementation of Traversal Algorithm for Breadth first traversal

a. Create a program that implements the concept of hashing using separate chaining.

# Function to display hashtable  
def display\_hash(hashTable):  
 for i in range(len(hashTable)):  
 print(i, end=" ")  
  
 for j in hashTable[i]:  
 print("-->", end=" ")  
 print(j, end=" ")  
  
 print()  
  
  
# Creating Hashtable as  
# a nested list.  
HashTable = [[] for \_ in range(10)]  
  
  
# Hashing Function to return  
# key for every value.  
def Hashing(keyvalue):  
 return keyvalue % len(HashTable)  
  
  
# Insert Function to add  
# values to the hash table  
def insert(Hashtable, keyvalue, value):  
 hash\_key = Hashing(keyvalue)  
 Hashtable[hash\_key].append(value)  
  
  
# Driver Code  
insert(HashTable, 10, 'Allahabad')  
insert(HashTable, 25, 'Mumbai')  
insert(HashTable, 20, 'Mathura')  
insert(HashTable, 9, 'Delhi')  
insert(HashTable, 21, 'Punjab')  
insert(HashTable, 21, 'Noida')  
  
display\_hash(HashTable)

REFERENCE: <https://www.geeksforgeeks.org/implementation-of-hashing-with-chaining-in-python/>

**Observation:**

I believe it is critical to understand various algorithms. An algorithm is a process procedure that specifies a set of instructions that must be carried out in a precise order to generate the desired result. Because algorithms are frequently designed independently of underlying languages, an algorithm can be implemented in more than one programming language.

Python is an excellent programming language for beginners since it makes it simple to construct programs ranging from simple text processing to web browsers to games. One of the primary challenges that many of us have encountered when programming is the lack of pointers in Python.

**Conclusion:**

I therefore conclude that understanding data structures and algorithms is the cornerstone for recognizing programmers, offering yet another reason for computer enthusiasts to better inspire them to pursue Python programming. This knowledge can have huge advantages and benefits and can widen our minds.

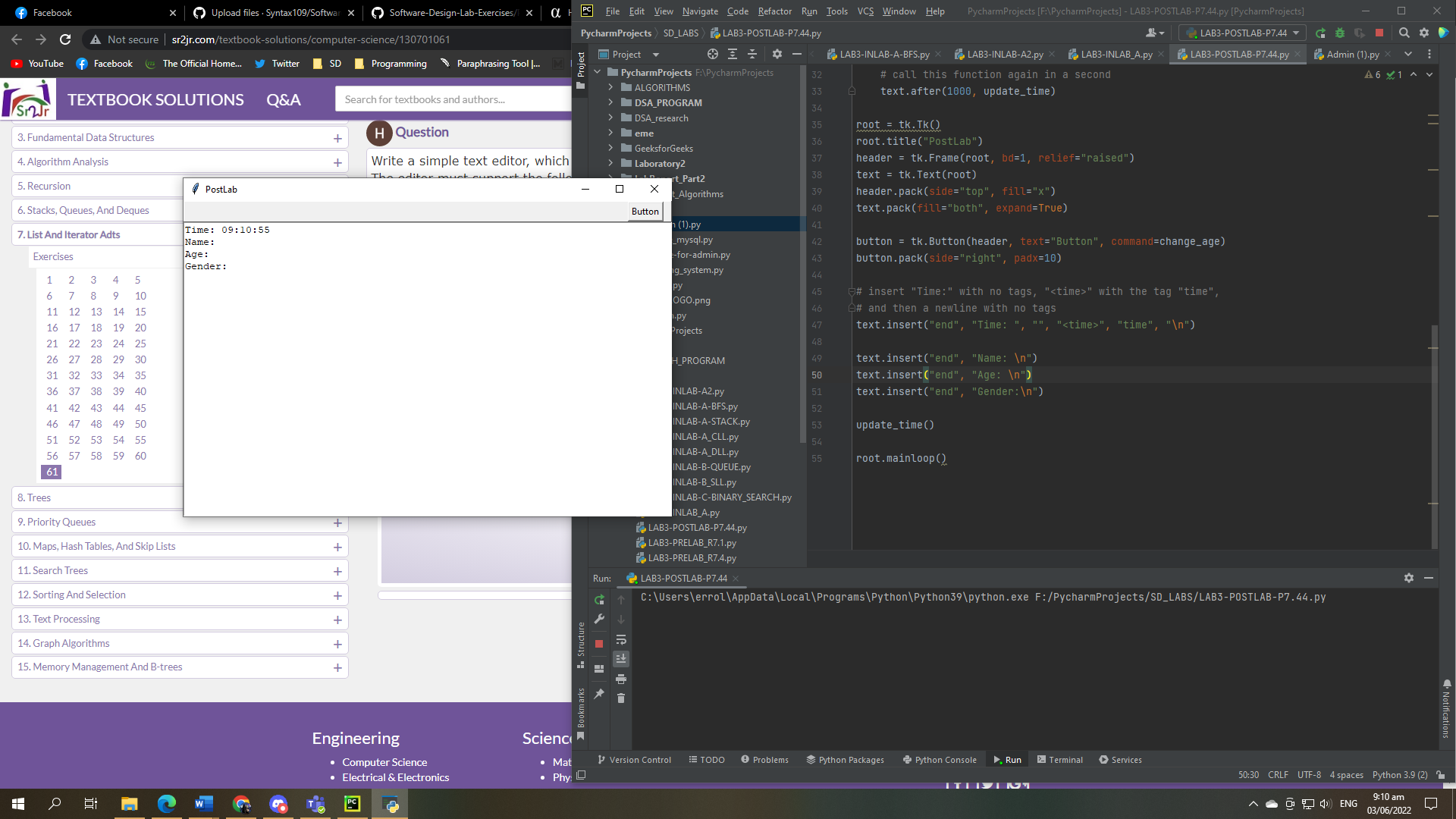
3.PostLab

***Projects***

* Ref Lab3 Data Structures and Algorithms in Python Michael T. Goodrich, Roberto Tamassia et.al: Chapter 7 Projects page 297 Do Projects P-7.44, P-7.45, P-7.46

**P-7.44:** Write a simple text editor that stores and displays a string of characters using the positional list ADT, together with a cursor object that highlights a position in this string. A simple interface is to print the string and then to use a second line of output to underline the position of the cursor. Your editor should support the following operations: • left: Move cursor left one character (do nothing if at beginning). • right: Move cursor right one character (do nothing if at end). • insert c: Insert the character c just after the cursor. • delete: Delete the character just after the cursor (do nothing at end).

import tkinter as tk  
from datetime import datetime  
from contextlib import contextmanager  
  
@contextmanager  
def preserve\_insert\_cursor(text):  
 *"""Performs an action without changing the insertion cursor location"""* saved\_insert = text.index("insert")  
 yield  
 text.mark\_set("insert", saved\_insert)  
  
def change\_age():  
 *"""Change the age on line 3"""* with preserve\_insert\_cursor(text):  
 text.delete("3.5", "3.0 lineend")  
 text.insert("3.5", "30")  
  
def update\_time():  
 with preserve\_insert\_cursor(text):  
 # find all ranges of text tagged with "time" and replace  
 # them with the current time  
 now = datetime.now()  
 timestring = now.strftime("%H:%M:%S")  
  
 ranges = list(text.tag\_ranges("time"))  
 while ranges:  
 start = ranges.pop(0)  
 end = ranges.pop(0)  
 text.delete(start, end)  
 text.insert(start, timestring, "time")  
  
 # call this function again in a second  
 text.after(1000, update\_time)  
  
root = tk.Tk()  
root.title("PostLab")  
header = tk.Frame(root, bd=1, relief="raised")  
text = tk.Text(root)  
header.pack(side="top", fill="x")  
text.pack(fill="both", expand=True)  
  
button = tk.Button(header, text="Button", command=change\_age)  
button.pack(side="right", padx=10)  
  
# insert "Time:" with no tags, "<time>" with the tag "time",  
# and then a newline with no tags  
text.insert("end", "Time: ", "", "<time>", "time", "\n")  
  
text.insert("end", "Name: \n")  
text.insert("end", "Age: \n")  
text.insert("end", "Gender:\n")  
  
update\_time()  
  
root.mainloop()

**output:**

**discussion:**

PyCharm may be used to debug our system. Debugging is essential because it allows software engineers and developers to fix vulnerabilities in a program before it is released to the public. It is a process that supplements testing by determining how a mistake affects the overall performance of a program.

**P-7.45** An array A is sparse if most of its entries are empty (i.e., None). A list L can be used to implement such an array efficiently. In particular, for each nonempty cell A[i], we can store an entry (i,e) in L, where e is the element stored at A[i]. This approach allows us to represent A using O(m) storage, where m is the number of nonempty entries in A. Provide such a SparseArray class that minimally supports methods getitem (j) and setitem (j, e) to provide standard indexing operations. Analyze the efficiency of these methods.

The primary methods of an Array List ADT are get(), set(), add(), delete(), size(), and isEmpty ( ). A list is a dynamic ordered tuple of homogenous items A0,A1,...A m-1, where Ai represents the list's i-th element. The array is sparse if the bulk of its entries are null. To efficiently implement such an array, A, a list L can be employed.

The primary methods of an Array List ADT are get(), set(), add(), delete(), size(), and isEmpty ( ). A list is a dynamic ordered tuple of homogenous items A0,A1,...A m-1, where Ai represents the list's i-th element. The array is sparse if the bulk of its entries are null. To efficiently implement such an array, A, a list L can be employed.

1. Make use of an array A of size m. The operation get(i) is implemented out in O(1) time by returning A[i].
2. The set(i,0) operation is performed in O(1) time by doing t=A[i], A[i]=0, and returning t.
3. In operation add(i,0), we need to make new element by shifting forward then n-i elements A[i],.....A[m-1]. In the worst case (i=0) then takes O(m) time.
4. In operation remove(i), we must fill the hole left by the eliminated element by moving backward then n-i-1 elements A[i+],......A[m-1]. In the worst case (i=0) then takes O(m) time.

The following are efficient approaches for performing array list ADT methods:

* + - 1. The space used by the data structure is O(m).
      2. The size, isEmpty, get and set run in O(1) time.
      3. Add and remove run on O(n) time in worst case.
      4. If use the array in a circular operations add(0,x) and remove(0,x) run in O(1) time.
      5. When an array is full during an add operation, instead of throwing an exception, we can replace it with a larger one.

**P-7.46** Although we have used a doubly linked list to implement the positional list ADT, it is possible to support the ADT with an array-based implementation. The key is to use the composition pattern and store a sequence of position items, where each item stores an element as well as that element’s current index in the array. Whenever an element’s place in the array is changed, the recorded index in the position must be updated to match. Given a complete class providing such an array-based implementation of the positional list ADT. What is the efficiency of the various operations?

One simple technique to use an array to implement a list is to store list items in array elements 0..n-1, where n is the current list length. We'd also need to maintain track of the current position and the number of items in the list, which we could do by adding instance variables to our class, which could look like this.

public class ListViaNaiveAry { private Object[] contents; // array in which list items are stored private int locOfLast; // points to array element holding last item private int crrntPos; // indicates item previous to current position

**Observers:**

lengthOf() : return locOfLast + 1; isEmpty() : return lengthOf() == 0; atFront() : return crrntPos == -1; atRear() : return crrntPos == locOfLast; prevItem() : if (atFront()) { throw an exception; } else { return contents[crrntPos]; } nextItem() : if (atRear()) { throw an exception; } else { return contents[crrntPos+1]; }

**Navigation Mutators:**

moveToFront() : crrntPos = -1; moveToRear() : crrntPos = locOfLast; moveForward() : if (atRear()) { throw an exception; } else { crrntPos = crrntPos + 1; } moveBackward() : if (atFront()) { throw an exception; } else { crrntPos = crrntPos - 1; }

**Node Mutators:**

replacePrev(x) : if (atFront()) { throw an exception; } else { contents[crrntPos] = x; } replaceNext(x) : if (atRear()) { throw an exception; } else { contents[crrntPos+1] = x; } insertRear(x) : if ( lengthOf() == contents.length ) { code to increase length of contents[] } contents[locOfLast+1] = x; locOfLast = locOfLast + 1;

Thus far, everything looks to be in good working order. However, as we will see, the insertion and deletion operations are difficult since there appears to be no way to make them run faster than in linear time (i.e., time proportional to the length of the list).

removePrev() : if (atFront()) { throw an exception; } else { // shift contents[crrntPos+1..locOfLast] to left one place for (int i = crrntPos; i != locOfLast; i = i+1) { contents[i] = contents[i+1]; } locOfLast = locOfLast - 1; } insertFront(x) : // shift contents[0..locOfLast] to right one place for (int i = locOfLast+1; i != 0; i = i-1) { contents[i] = contents[i-1]; } locOfLast = locOfLast + 1; contents[0] = x;

To conduct removePrev(), the values in the segment contents[crrntPos+1..locOfLast] are moved one place to the left. We should anticipate around half of the items in the list to be relocated on average, indicating that this is a linear time process.

InsertFront() is slightly worse (but still linear in time), because every value in the segment contents[0..locOfLast] must be relocated to the right one location to create room (at location 0) for the inserted value.

Both insertBefore() and insertAfter() are similar to insertFront(), with the main difference being that the array segment to be relocated to the right begins at crrntPos (in insertBefore()) or crrntPos+1 (in insertAfter()).

Given the constraints of the representation method we've chosen, there appears to be no way to avoid having the insertion and deletion processes take time proportional to the amount of elements in the list. This begs the issue of whether there is another array-based representation scheme that enables more efficient versions of these operations.

**Observation**

Python is a general-purpose programming language that can be used to build a text editor with tkinter frameworks, debug with pycharm, and store data in a variety of data kinds. It is one of the most popular programming languages among developers due to its wealth of libraries, adaptability, and simple structure.

**Conclusion**

I therefore conclude that whether you are an experienced programmer or a newbie, data structures and algorithms are essential in Python. These concepts are essential when carrying out data operations and improving data processing. While data structures help organize information, algorithms give direction for tackling data analysis challenges. They collaborate to provide a mechanism for computer scientists to process the data supplied as input.

**URL of Github repository**

<https://github.com/Syntax109/SoftwareDesign-Laboratory-Exercises>