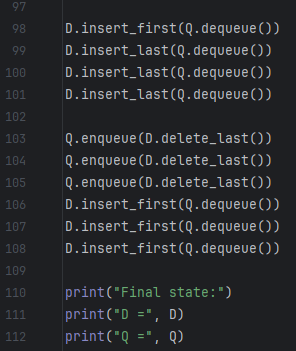
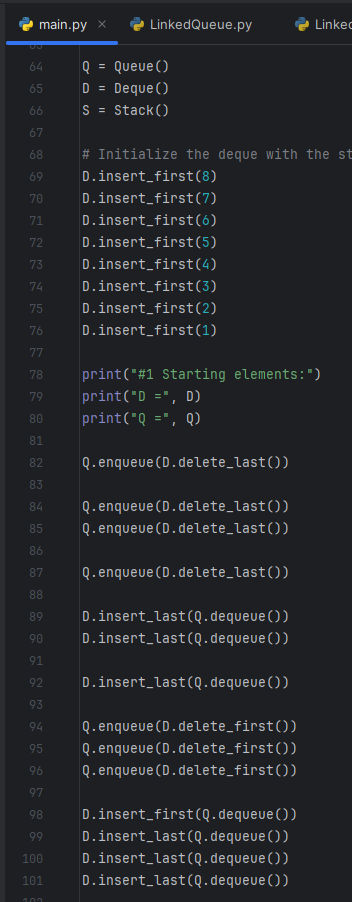
Molina, Joshua Ali S.

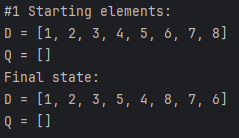
DSALGO1

IDB2

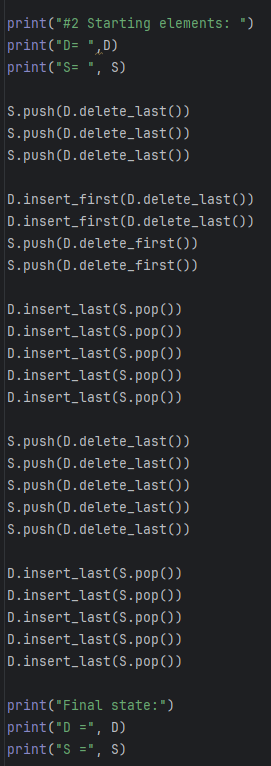
11/15/2024

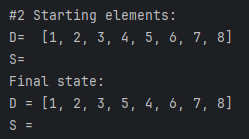
#1:





#2:





**LinkedQueue Class:**

class LinkedQueue:  
 *"""FIFO queue implementation using a singly linked list for storage."""* # ---------------- nested \_Node class ----------------  
 class \_Node:  
 *"""Lightweight non-public class for storing a singly linked node."""* \_\_slots\_\_ = '\_element', '\_next' # streamline memory usage  
  
 def \_\_init\_\_(self, element, next):  
 self.\_element = element  
 self.\_next = next  
  
 # --------------- queue methods ----------------  
 def \_\_init\_\_(self):  
 *"""Create an empty queue."""* self.\_head = None  
 self.\_tail = None  
 self.\_size = 0  
  
 def \_\_len\_\_(self):  
 *"""Return the number of elements in the queue."""* return self.\_size  
  
 def is\_empty(self):  
 *"""Return True if the queue is empty."""* return self.\_size == 0  
  
 def first(self):  
 *"""Return (but do not remove) the element at the front of the queue."""* if self.is\_empty():  
 raise Exception('Queue is empty')  
 return self.\_head.\_element # front aligned with the head of the list  
  
 def dequeue(self):  
 *"""Remove and return the first element of the queue (FIFO).  
  
 Raise Exception if the queue is empty.  
 """* if self.is\_empty():  
 raise Exception('Queue is empty')  
 answer = self.\_head.\_element  
 self.\_head = self.\_head.\_next  
 self.\_size -= 1  
 if self.is\_empty(): # special case as queue is empty  
 self.\_tail = None # removed head had been the tail  
 return answer  
  
 def enqueue(self, e):  
 *"""Add an element to the back of queue."""* newest = self.\_Node(e, None) # node will be new tail node  
 if self.is\_empty():  
 self.\_head = newest # special case: previously empty  
 else:  
 self.\_tail.\_next = newest  
 self.\_tail = newest # update reference to tail node  
 self.\_size += 1  
  
 def \_\_str\_\_(self):  
 *"""Return a string representation of the queue's contents."""* elements = []  
 current = self.\_head  
 while current is not None:  
 elements.append(str(current.\_element))  
 current = current.\_next  
 return "[" + ", ".join(elements) + "]"

**LinkedDeque Class:**

from DoublyLinkedBase import \_DoublyLinkedBase  
class LinkedDeque(\_DoublyLinkedBase):#note the use of inheritance  
 *'''Double ended queue implementation based on a doubly linked list.'''* def first(self):  
 *'''Return but do not remove the element at the front of the deque'''* if self.is\_empty():  
 raise Exception("Deque is empty!")  
 return self.\_header.\_next.\_element #real item just after header  
 def last(self):  
 *'''Return but do not remove the element at the back of the deque'''* if self.is\_empty():  
 raise Exception("Deque is empty!")  
 return self.\_trailer.\_prev.\_element #real item just before trailer  
 def insert\_first(self, e):  
 *'''Add an element to the fron of the deque.'''* self.\_insert\_between(e, self.\_header, self.\_header.\_next)#after header  
 def insert\_last(self, e):  
 *'''Add an element to the back of the deque'''* self.\_insert\_between(e, self.\_trailer.\_prev, self.\_trailer)#before trailer  
 def delete\_first(self):  
 *'''Remove and return the element from the front of the deque.'''* '''Raise Exception if the deque is empty.'''  
 if self.is\_empty():  
 raise Exception("Deque is empty!")  
 return self.\_delete\_node(self.\_header.\_next)#use inherited method  
 def delete\_last(self):  
 *'''Remove and return the element from the back of the deque.'''* '''Raise Exception if the deque is empty.'''  
 if self.is\_empty():  
 raise Exception("Deque is empty!")  
 return self.\_delete\_node(self.\_trailer.\_prev)#use inherited method  
 def \_\_str\_\_(self):  
 *"""Return a string representation of the deque's contents."""* elements = []  
 current = self.\_header.\_next  
 while current is not self.\_trailer:  
 elements.append(str(current.\_element))  
 current = current.\_next  
 return "[" + ", ".join(elements) + "]"

**LinkedStack Class:**

class LinkedStack:  
 class Node:  
 def \_\_init\_\_(self, data=None, next\_node=None):  
 self.data = data  
 self.next = next\_node  
  
 def \_\_init\_\_(self):  
 self.top = None  
 self.size = 0  
  
 def push(self, item):  
 new\_node = self.Node(item, self.top)  
 self.top = new\_node  
 self.size += 1  
  
 def pop(self):  
 if self.is\_empty():  
 raise IndexError("pop from empty stack")  
 data = self.top.data  
 self.top = self.top.next  
 self.size -= 1  
 return data  
  
 def peek(self):  
 if self.is\_empty():  
 raise IndexError("peek from empty stack")  
 return self.top.data  
  
 def is\_empty(self):  
 return self.size == 0  
  
 def \_\_str\_\_(self):  
 items = []  
 current = self.top  
 while current:  
 items.append(str(current.data))  
 current = current.next  
 return " -> ".join(items)