"""

Thomas Morris

Assignment 3

October 17th, 2019

Instructions:

Implement the circular double-ended queue (deque)

which allows insertion and deletion from either front or rear end.

Note:

This program was implemented with using information from pgs. 243-249

in Data Structures and Algorithms by Goodrich, Tamassia, and Goldwasser

"""

class MyCircularDeque:

#constructor to set the size of deque to be k

DEFAULT\_CAPACITY = 0

def \_\_init\_\_(self, k):

#Create an empty deque

MyCircularDeque.DEFAULT\_CAPACITY = k

self.\_data = [None]\*MyCircularDeque.DEFAULT\_CAPACITY

self.\_size = 0

self.\_front = 0

def \_\_len\_\_(self):

#Return the number of elements in the deque

return self.\_size

def isEmpty(self):

#Return True if the deque is empty

return self.\_size == 0

def isFull(self):

return self.\_size == MyCircularDeque.DEFAULT\_CAPACITY

def getFront(self):

#Return but do not remove the first element at the front of the deque

if self.isEmpty():

return -1

return self.\_data[self.\_front]

def getRear(self):

#Return the last element but do not remove it

if self.isEmpty():

return -1

back = (self.\_front+self.\_size-1)%len(self.\_data)

return self.\_data[back]

def deleteFront(self):

#Removes the first item in the deque, returns true if successful

if self.isEmpty():

return False

self.\_data[self.\_front] = None

self.\_front = (self.\_front+1)%len(self.\_data)

self.\_size -= 1

return True

def deleteLast(self):

if self.isEmpty():

return False

self.\_data[self.\_front] = None

self.\_front = (self.\_front+1)%len(self.\_data)

self.\_size -= 1

return True

def insertLast(self, e):

#Adds an element to the back of the deque, returns True if successful

if self.isFull():

return False

last = (self.\_front+self.\_size)%len(self.\_data)

self.\_data[last] = e

self.\_size += 1

return True

def insertFront(self, e):

#Adds an element to the front of deque, returns True if successful

if self.isFull():

return False

self.\_front = (self.\_front-1)%len(self.\_data)

self.\_data[self.\_front] = e

self.\_size += 1

return True

if \_\_name\_\_ == "\_\_main\_\_":

circularDeque = MyCircularDeque(3)

print(circularDeque.insertLast(1))

print(circularDeque.insertLast(2))

print(circularDeque.insertFront(3))

print(circularDeque.insertFront(4))

print(circularDeque.getRear())

print(circularDeque.isFull())

print(circularDeque.deleteLast())

print(circularDeque.insertFront(4))

print(circularDeque.getFront())

"""

Thomas Morris

Assignment 3

October 17th, 2019

Instructions:

Implement the program which merges two sorted singly-linked lists

and return it as a new list. The new list should be made by splicing

together the nodes of the first two lists.

You should use your own class and include a testing.

Note:

Pages 256-266 of Data Structures and Alogrithms were referenced

"""

class Node:

def \_\_init\_\_(self, element):

self.\_element = element

self.\_enext = None

class LinkedList:

#initialize the head

def \_\_init\_\_(self):

self.\_head = None

self.\_size = 0

#add an element to the front of the list

def add\_first(self, e):

newest = Node(e)

newest.\_enext = self.\_head

self.\_head = newest

self.\_size += 1

#print out the list

def printList(self):

node = self.\_head

while node:

print(node.\_element, "-> ", end='')

node = node.\_enext

print("∅")

#merge the two lists provided together using recursion

def merge(self, l1, l2):

#make a empty node

l3 = None

#stop recursing when either l1 or l2 head is null

#and then return the rest of the other list head

if l1 == None:

return l2

if l2 == None:

return l1

#compare the elements at each head to place them at the right spot

#then make the recursive call to merge the list with the next element

if l1.\_element <= l2.\_element:

l3 = l1

l3.\_enext = LinkedList().merge(l1.\_enext, l2)

else:

l3 = l2

l3.\_enext = LinkedList().merge(l1, l2.\_enext)

#return the new list that was created by the recursive calls

return l3

if \_\_name\_\_ == '\_\_main\_\_':

l1 = LinkedList()

l1.add\_first(4)

l1.add\_first(2)

l1.add\_first(1)

l1.printList()

l2 = LinkedList()

l2.add\_first(4)

l2.add\_first(3)

l2.add\_first(1)

l2.printList()

l3 = LinkedList()

l3.\_head = l3.merge(l1.\_head,l2.\_head)

l3.printList()

# -\*- coding: utf-8 -\*-

"""

Thomas Morris

Assignment 3

October 17, 2019

Implement a queue using linked lists. You should use your own class with the methods

(enqueue(object), dequeue(), first(), len(), is\_empty(), search()) and include a testing.

Note that search(object) returns True (or False) to check if an object is in the Queue

Note:

Pages 264-266 in Data Structures and Algorithms were referenced

"""

class \_Node:

def \_\_init\_\_(self, element, enext):

self.\_element = element

self.\_next = enext

class LinkedQueue:

def \_\_init\_\_(self):

self.\_head = None

self.\_tail = None

self.\_size = 0

def len(self):

return self.\_size

def is\_empty(self):

return self.\_size == 0

def first(self):

#return but do not remove first element

if self.is\_empty():

raise Exception('Queue is empty')

return self.\_head.\_element

def dequeue(self):

#remove first element from queue

if self.is\_empty():

raise Exception('Queue is empty')

element = self.\_head.\_element

self.\_head = self.\_head.\_next

self.\_size -= 1

if self.is\_empty():

self.\_tail = None

return element

def enqueue(self, e):

#add element to back of queue

newest = \_Node(e, None)

if self.is\_empty():

self.\_head = newest

else:

self.\_tail.\_next = newest

self.\_tail = newest

self.\_size += 1

def search(self, key):

#search the entire queue for the key element provided

pos = self.\_head

while pos:

if pos.\_element == key:

return True

pos = pos.\_next

return False

if \_\_name\_\_ == '\_\_main\_\_':

l1 = LinkedQueue()

l1.enqueue(5)

l1.enqueue(73)

l1.enqueue(9)

l1.enqueue(8)

l1.enqueue(16)

l1.enqueue(57)

l1.enqueue(95)

print(l1.search(5))

l1.dequeue()

print(l1.search(5))

print(l1.first())

print(l1.len())

print(l1.is\_empty())

print(l1.search(73))