Richard Hayes Crowley

07/09/2021

CSC\_242\_Lab\_010

**SOURCE CODE:**

**PriorityLinkedQueue.py**

class Node(object):

'''represents singly linked node'''

def \_\_init\_\_(*self*, *data*, *next*=None) -> None:

super().\_\_init\_\_()

*self*.data = *data*

*self*.next = *next*

class Comparable(object):

''' Comparable class for priority queue'''

def \_\_init\_\_(*self*, *data*, *priority*=1) -> None:

super().\_\_init\_\_()

*self*.data = *data*

*self*.priority = *priority*

def \_\_str\_\_(*self*):

*return* str(*self*.data)

def \_\_eq\_\_(*self*, *other*: object) -> bool:

*if* *self* is *other*:

*return* True

*if* type(*self*) != type(*other*):

*return* False

*return* *self*.priority == *other*.priority

def \_\_lt\_\_(*self*, *other*):

*return* *self*.priority < *other*.priority

def \_\_le\_\_(*self*, *other*):

*return* *self*.priority <= *other*.priority

def setData(*self*, *data*):

*self*.data = *data*

def getData(*self*):

*return* *self*.data

def getPriority(*self*):

*return* *self*.priority

class PriorityLinkedQueue(object):

'''a linked-based priority queue implementation'''

*# special methods*

def \_\_init\_\_(*self*, *sourceCollection*=None) -> None:

super().\_\_init\_\_()

*self*.size = 0

*if* *sourceCollection*:

*for* item *in* *sourceCollection*:

*self*.enqueue(item)

def \_\_iter\_\_(*self*):

def visitNodes(*node*):

*if* *node* != None:

visitNodes(*node*.next)

tempList.append(*node*.data)

tempList = list()

visitNodes(*self*.front)

*return* iter(tempList)

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

*return* *self*.size

def \_\_str\_\_(*self*):

*return* "[" + f"{', '.join(map(str, *self*))}" + "]"

def \_\_add\_\_(*self*, *other*):

result = PriorityLinkedQueue(*self*)

*for* item *in* *other*:

result.enqueue(item)

*return* result

def \_\_eq\_\_(*self*, *other*):

*if* *self* is *other*:

*return* True

*if* type(*self*) != type(*other*) or len(*self*) != len(*other*):

*return* False

otherIter = iter(*other*)

*for* item *in* *self*:

*if* item != next(otherIter):

*return* False

*return* True

*# accessors*

def isEmpty(*self*):

*return* len(*self*) == 0

def peek(*self*):

*if* *self*.isEmpty():

*raise* KeyError("The queue is empty.")

*return* str(*self*.front.data)

def count(*self*, *item*):

'''returns # of instances of item in self'''

count = 0

*for* el *in* *self*:

*if* el is *item*:

count += 1

*elif* el == *item*:

count += 1

*return* count

*# mutators*

def enqueue(*self*, *newItem*):

'''inserts new item after items of greater or equal priority or ahead of items of lesser priority.

A has greater priority than B if A < B

Assumes newItem is a comparable (see Comparable class)

'''

*if* *self*.isEmpty() or *newItem* >= *self*.rear.data:

*# new item goes to the rear of the queue, if its priority is less than or equal to the item current in the rear, else if the queue is empty, it goes in the front*

newNode = Node(*newItem*, None)

*if* *self*.isEmpty():

*self*.front = newNode

*else*:

*self*.rear.next = newNode

*self*.rear = newNode

*self*.size += 1

*else*:

*# search for position where the new item has less priority*

probe = *self*.front

*# step through nodes until newItem is less than or equal to probe.data*

*while* *newItem* >= probe.data:

trailer = probe

probe = probe.next

*# instantiate new Node for new item to go into the queue*

newNode = Node(*newItem*, probe)

*if* probe == *self*.front:

*# new item goes to front*

*self*.front = newNode

*else*:

*# new item goes between two nodes*

trailer.next = newNode

*self*.size += 1

def dequeue(*self*):

*if* *self*.isEmpty():

*raise* KeyError("The queue is empty.")

oldItem = *self*.front.data

*self*.front = *self*.front.next

*if* *self*.front is None:

*self*.rear = None

*self*.size -= 1

*return* oldItem

def clear(*self*):

*self*.size = 0

*self*.front = None

**Cookout.py**

*from* priority\_linked\_queue *import* Comparable, PriorityLinkedQueue

*from* simple\_term\_menu *import* TerminalMenu

*from* tabulate *import* tabulate

*from* random *import* choice

*# priority is given in this queue to items that take very little time to make, e.g., fries and milkshakes, as opposed to cheeseburgers and cookout trays*

*# the queue is constructed by passing in a number of these items and then popping them out in order of priority and time spent in the queue*

menuDict = {

"fries": 1,

"milkshake": 1,

"cheeseburger": 2,

"hotdog": 2,

"chilidog": 3,

"chicken tenders": 3,

"cookout tray": 4,

"party tray": 5

}

menuTable = tabulate([

[key, value] *for* key, value *in* menuDict.items()], *headers*=["Menu Item", "Priority (ordinal speed)"], *tablefmt*='github', *numalign*="left")

def main():

print("~\*~\*~\*~ Welcome to Cookout Queue Simulator! ~\*~\*~\*~\*~")

print("Menu (quicker items get first priority in queue):\n")

print(menuTable)

*while* True:

customer\_queue = None

entry = TerminalMenu(["Generate queue", "Exit"],

*title*="\nWhat would you like to do?").show()

*if* entry == 0:

number\_of\_customers = int(input(

"\nHow many customers in line?: "))

print(

"\nGenerating random orders and inserting them into appropriate position in queue...")

sourceCollection = []

*for* i *in* range(number\_of\_customers):

randomChoice = choice(

list(menuDict.items()))

comparable = Comparable(

f"Customer #{i+1}: {randomChoice[0]}", randomChoice[1])

sourceCollection.append(comparable)

customer\_queue = PriorityLinkedQueue(sourceCollection)

print(

f"\nPrioritized order queue for {len(customer\_queue)} customer(s) created. The bottom most customers are first in line, as their orders are fastest to make!")

*for* i *in* customer\_queue:

print(i.getData())

*while* True:

*if* customer\_queue.isEmpty():

print("\nThe queue is empty!")

*break*

run\_queue = TerminalMenu(

["View Queue", "Peek", "Serve an order", "Simulate Dequeue", "Nothing"], *title*="\nWhat would you like to do with this queue?").show()

*if* run\_queue == 0:

print("")

*for* i *in* customer\_queue:

print(i.getData())

*elif* run\_queue == 1:

print(

f"\nFirst up in the queue is: {customer\_queue.peek()}")

*elif* run\_queue == 2:

print(f"\nOrder up!: {customer\_queue.dequeue()}")

*elif* run\_queue == 3:

print("\nDequeing each order until the queue is empty...\n")

*while* not customer\_queue.isEmpty():

print(f"\nOrder up!: {customer\_queue.dequeue()}")

*else*:

*break*

*else*:

print("\nGoodbye!")

exit()

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

main()

**OUTPUT:**

See attached video!