**QUAID-E-AWAM UNIVERSITY OF ENGINEERING, SCIENCE AND TECHNOLOGY**

**DEPARTMENT OF ARTIFICIAL INTELLIGENCE**

Dated: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Roll No(s): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab 04 – Queues, Deques, and Priority Queues**

**Lab Objectives**

* Understand how queues work in Python using lists.
* Implement a standard queue and a circular queue.
* Learn how to use Deques (double-ended queues) from Python's collections module.
* Implement a priority queue using Python's heapq module.
* Apply queue concepts to simple problems like simulations.

**Background**

A **queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle, much like a waiting line. New elements are added to the rear (enqueue), and elements are removed from the front (dequeue).

A **circular queue** is an enhancement of a standard queue implemented with a fixed-size array. It efficiently reuses space by allowing the rear and front pointers to "wrap around" to the beginning of the array.

A **Deque (Double-Ended Queue)** is a generalisation of a queue that allows elements to be added or removed from *both* the front and the rear.

A **Priority Queue** is a special type of queue where each element is associated with a priority. Elements with higher priority are served before elements with lower priority, regardless of when they were added. Python's heapq module is commonly used to implement this efficiently.

**Lab Tasks**

**Task 1 – Implement Queue Using List**

# Simple Queue Implementation using a List

queue = []

def enqueue(element):

queue.append(element)

print(f"Enqueued: {element}")

def dequeue():

if not queue:

print("Queue Underflow")

else:

# pop(0) removes from the front (FIFO)

print(f"Dequeued: {queue.pop(0)}")

def front():

if queue:

print(f"Front Element: {queue[0]}")

enqueue(10)

enqueue(20)

enqueue(30)

front()

dequeue()

front()

dequeue()

**Task 2 – Implement Circular Queue**

# Circular Queue Implementation

class CircularQueue:

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

self.size = size

self.queue = [None] \* size

self.front = self.rear = -1

def enqueue(self, data):

# Check if queue is full

if (self.rear + 1) % self.size == self.front:

print("Queue is Full")

# Check if first element

elif self.front == -1:

self.front = 0

self.rear = 0

self.queue[self.rear] = data

print(f"Enqueued: {data}")

else:

self.rear = (self.rear + 1) % self.size

self.queue[self.rear] = data

print(f"Enqueued: {data}")

def dequeue(self):

if self.front == -1:

print("Queue is Empty (Underflow)")

# Check if last element

elif self.front == self.rear:

temp = self.queue[self.front]

self.front = -1

self.rear = -1

print(f"Dequeued: {temp}")

else:

temp = self.queue[self.front]

self.front = (self.front + 1) % self.size

print(f"Dequeued: {temp}")

cq = CircularQueue(3)

cq.enqueue(1)

cq.enqueue(2)

cq.enqueue(3)

cq.enqueue(4) # Queue Full

cq.dequeue()

cq.enqueue(4) # Wraps around

cq.dequeue()

cq.dequeue()

cq.dequeue()

cq.dequeue() # Queue Empty

**Task 3 – Implement Deque using collections**

from collections import deque

# Create a deque

d = deque()

# Add to the right (rear)

d.append(10)

print("Append (right):", d)

# Add to the left (front)

d.appendleft(5)

print("Append left (front):", d)

d.append(20)

print("Append (right):", d)

# Remove from the right (rear)

print(f"Popped (right): {d.pop()}")

print("Deque after pop:", d)

# Remove from the left (front)

print(f"Popped left (front): {d.popleft()}")

print("Deque after popleft:", d)

**Task 4 – Implement Priority Queue using heapq**

Python

import heapq

# A priority queue is just a list managed by heapq

pq = []

# heapq implements a min-heap (lowest value has highest priority)

# Push items with (priority, data)

heapq.heappush(pq, (3, 'Task C'))

heapq.heappush(pq, (1, 'Task A'))

heapq.heappush(pq, (2, 'Task B'))

print("Priority Queue (min-heap):", pq)

# Pop the item with the highest priority (lowest number)

print(f"Popped: {heapq.heappop(pq)}")

print(f"Popped: {heapq.heappop(pq)}")

print("Priority Queue after pops:", pq)

**Task 5 – Bank Queue Simulation (Conceptual)**

# A simple simulation using a list-based queue

import random

bank\_queue = []

tellers = 2

simulation\_time = 10

print("--- Bank Simulation Start ---")

for time\_step in range(1, simulation\_time + 1):

print(f"\nTime Step: {time\_step}")

# Customer arrival (random chance)

if random.random() < 0.5: # 50% chance of new customer

customer\_id = f"Cust-{time\_step}"

bank\_queue.append(customer\_id)

print(f"New Arrival: {customer\_id}, Queue: {bank\_queue}")

# Tellers serving customers

for i in range(tellers):

if bank\_queue: # Check if queue is not empty

served = bank\_queue.pop(0)

print(f"Teller {i+1} serving: {served}")

else:

print(f"Teller {i+1} is idle")

print("\n--- Simulation End ---")

print(f"Customers remaining in queue: {bank\_queue}")

**Questions**

1. Why is a queue called a **FIFO** structure?
2. What is the main advantage of a circular queue over a standard queue implemented with a fixed-size list?
3. How can a deque from the collections module be used to implement a stack (LIFO)?
4. In Task 4, heapq implements a min-heap. How would you use it to implement a max-heap (where the highest number has the highest priority)?
5. What is the "producer-consumer problem," and how does a queue help solve it?