Lab: Queues — Fixed-Size vs Circular (2 hours)

Instructor: Azeem Aslam

Format: Task Brief → Code → ✅ Final Result

Learning Objectives

- Implement a teaching Queue using a Python list (conceptual).
- Implement a fixed-size array-backed queue and observe wasted slots after dequeues.
- Implement a circular queue using modulo wrap-around: `rear = (rear + 1) % N`, `front = (front + 1) % N`.
- Trace `front`, `rear`, and `count` to understand empty/full states.
- Apply queues to a tiny real-world simulation (printer jobs).

> Tip: Run cells top-to-bottom. If something goes wrong, restart kernel and run all.

Task: FIFO Warm-Up (List)

Time: ~10 min

Goal: Warm-up and verify FIFO behavior conceptually.

What you'll do

1. Use a plain Python list to **append** items and **pop(0)** to mimic queue behavior.
2. Print the order of removal to confirm **First-In, First-Out (FIFO)**.

You should learn

- Why `pop(0)` is O(n) and only good for teaching.
- Visual feel of FIFO before implementing classes.

In [ ]:
```
# Show results after each step for the user's FIFO warm-up
q = []                      # start empty
print("Start:", q)

q.append('A')               # enqueue A
print("After enqueue A:", q)

q.append('B')               # enqueue B
print("After enqueue B:", q)

q.append('C')               # enqueue C
```

```python
print("After enqueue C:", q)

out1 = q.pop(0)          # dequeue -> 'A'
print("After dequeue ->", out1, "| Queue:", q)

out2 = q.pop(0)          # dequeue -> 'B'
print("After dequeue ->", out2, "| Queue:", q)

out3 = q.pop(0)          # dequeue -> 'C'
print("After dequeue ->", out3, "| Queue:", q)

print('Removed order:', out1, out2, out3)
```

✅ Final Result

You should see `Removed order: A B C`, which confirms FIFO behavior.

Task: Teaching Queue (list-backed)

Time: ~20 min

Goal: Implement a **teaching Queue** backed by a Python list with clear methods.

What you'll do

1. Implement methods: `enqueue`, `dequeue`, `front`, `is_empty`.
2. Test the queue with 2–3 operations and print outputs.

You should learn

- Encapsulation of queue operations in a class.
- Why this version is **not efficient** (because of `pop(0)`), but great for learning.

In [ ]:

```python
# Teaching Queue (list-backed) with step-by-step outputs

class Queue:
    def __init__(self):
        self.items = []                     # internal list to hold elements
        print("Init ->", self.items)

    def is_empty(self):
        return len(self.items) == 0         # empty if length == 0

    def enqueue(self, x):
        self.items.append(x)                # append at end (rear)
        print(f"enqueue({x}) ->", self.items)

    def dequeue(self):
        if self.is_empty():
            print("dequeue() -> Underflow (None) |", self.items)
            return None                     # signal underflow
        val = self.items.pop(0)             # remove & return element at index 0 (fr
        print(f"dequeue() -> {val} |", self.items)
        return val
```

```python
    def front(self):
        if self.is_empty():
            print("front() -> None |", self.items)
            return None
        print("front() ->", self.items[0], "|", self.items)
        return self.items[0]            # peek front without removing

# quick test with step-by-step tracing
q = Queue()
q.enqueue(10)
q.enqueue(20)
_ = q.dequeue()     # expect 10
_ = q.front()       # expect 20
print('is_empty ->', q.is_empty())  # expect False
```

✅ Final Result

The output should show the first removed value is `10`, current front is `20`, and `empty?` is `False`.

Task: Fixed-Size Array Queue — See the Waste

Time: ~25 min

Goal: Implement a **fixed-size array queue** and observe **wasted slots** after dequeues.

What you'll do

1. Implement `ArrayQueue(size)` with `enqueue`, `dequeue`, `front_val`, `is_empty`, `is_full`.
2. Enqueue a few items, dequeue one or two, then inspect `front`, `rear`, `count`.
3. Print the internal array slice that is **logically active** to see the waste.

You should learn

- How `front` moves forward and left-side slots become **unusable** without shifting.
- Why this motivates the **circular** version.

In [7]:

```python
# ---- Fixed-size Array Queue ----
class ArrayQueue:
    def __init__(self, size):
        self.size = size                    # total capacity
        self.a = [None] * size              # fixed-size storage
        self.front = 0                      # index of current front
        self.rear = -1                      # index of last filled position
        self.count = 0                      # number of current elements

    def is_empty(self):
        return self.count == 0

    def is_full(self):
        return self.count == self.size
```

```python
    def enqueue(self, x):
        if self.is_full():
            print("Overflow")
            return False
        self.rear += 1                  # move right
        self.a[self.rear] = x           # place new element
        self.count += 1
        return True

    def dequeue(self):
        if self.is_empty():
            print("Underflow")
            return None
        val = self.a[self.front]        # read front
        self.front += 1                 # move right
        self.count -= 1
        return val

    def front_val(self):
        if self.is_empty():
            return None
        return self.a[self.front]

# demo: show wasteb
aq = ArrayQueue(5)
aq.enqueue(10); aq.enqueue(20); aq.enqueue(30)
print('dequeue ->', aq.dequeue())       # remove 10
print('dequeue ->', aq.dequeue())       # remove 20
print('front   ->', aq.front_val())     # expect 30
print('state   ->', 'count=', aq.count, 'front=', aq.front, 'rear=', aq.rear)
print('active  ->', aq.a[aq.front:aq.rear+1])  # logical active window
```

```
dequeue -> 10
dequeue -> 20
front   -> 30
state   -> count= 1 front= 2 rear= 2
active  -> [30]
```

✅ Final Result

You should see `front` advanced and `active` containing only the logical items (e.g., `[30]`) while left-side slots are now wasted.

Task: Circular Queue — Wrap-Around with %

Time: ~35 min

Goal: Implement a **circular queue** to reuse freed slots using modulo wrap-around.

What you'll do

1. Implement `CircularQueue(size)` with wrap-around in `enqueue` and `dequeue` :
   - `self.rear  = (self.rear  + 1) % self.size`

  - `self.front = (self.front + 1) % self.size`
2. Maintain a `count` to distinguish **empty vs full**.
3. Add a helper `to_list()` that returns elements in logical order for easy checking.
4. Show **wrap-around** by enqueuing, dequeuing, and enqueuing again.

You should learn

- How modulo arithmetic enables index wrap-around.
- Why circular queues fix the wasted-slot problem.

In [9]:
```
# ---- Circular Queue with modulo wrap-around ----
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.a = [None] * size
        self.front = 0
        self.rear = -1
        self.count = 0

    def is_empty(self):
        return self.count == 0

    def is_full(self):
        return self.count == self.size

    def enqueue(self, x):
        if self.is_full():
            print("Overflow")
            return False
        self.rear = (self.rear + 1) % self.size
        self.a[self.rear] = x
        self.count += 1
        return True

    def dequeue(self):
        if self.is_empty():
            print("Underflow")
            return None
        val = self.a[self.front]
        self.front = (self.front + 1) % self.size
        self.count -= 1
        return val

    def front_val(self):
        if self.is_empty():
            return None
        return self.a[self.front]

    def to_list(self):
        # return logical order of elements from front, length = count
        res = []
        idx = self.front
        for _ in range(self.count):
            res.append(self.a[idx])
            idx = (idx + 1) % self.size
        return res

# demo: wrap-around behavior
```

```python
cq = CircularQueue(5)
for x in [10, 20, 30, 40]:
    cq.enqueue(x)
print('start   ->', cq.to_list())        # [10,20,30,40]

print('dequeue ->', cq.dequeue())        # remove 10
print('dequeue ->', cq.dequeue())        # remove 20
print('mid     ->', cq.to_list())        # [30,40]

cq.enqueue(50); cq.enqueue(60)           # should wrap when needed
print('after   ->', cq.to_list())        # [30,40,50,60]
print('front   ->', cq.front_val())
print('state   ->', 'front=', cq.front, 'rear=', cq.rear, 'count=', cq.count)
```

```
start   -> [10, 20, 30, 40]
dequeue -> 10
dequeue -> 20
mid     -> [30, 40]
after   -> [30, 40, 50, 60]
front   -> 30
state   -> front= 2 rear= 0 count= 4
```

✅ Final Result

You should see wrap-around in action. After two dequeues and two enqueues, `to_list()` shows reused slots (e.g., `[30, 40, 50, 60]`) and indices reflect wrapping.

✅ Final Result

You should see a full queue near the end. If `enq(25)` fails (`False`), that's correct when the queue is already full. Assertions should pass silently.

Task: Mini-Project: Printer Jobs (FCFS)

Time: ~10 min

Goal: Mini-project — **Printer Job Simulation** using a queue.

What you'll do

1. Simulate incoming print jobs `J1..Jn` (strings) and **enqueue** them.
2. Process jobs in arrival order by **dequeueing**.
3. Print the service order.

You should learn

- How queues naturally model real-world **first-come, first-served** systems.

In [12]:
```python
# ---- Printer Job Simulation ----
def run_printer_sim(jobs, capacity=8):
    q = CircularQueue(capacity)
    print("Incoming jobs:", jobs)
```

```
    for j in jobs:
        if not q.enqueue(j):
            print("Queue full, job dropped:", j)   # simple handling
    serviced = []
    while not q.is_empty():
        serviced.append(q.dequeue())
    print("Serviced order:", serviced)
    return serviced

# demo
jobs = ["J1","J2","J3","J4","J5"]
serviced = run_printer_sim(jobs, capacity=4)
```

```
Incoming jobs: ['J1', 'J2', 'J3', 'J4', 'J5']
Overflow
Queue full, job dropped: J5
Serviced order: ['J1', 'J2', 'J3', 'J4']
```

✅ Final Result

You should see all jobs printed in the same arrival order until capacity is reached; excess jobs are dropped in this simple demo. Try changing `capacity` and `jobs` to experiment.

Wrap-Up & Viva (Quick Oral Check)

- What is FIFO? Where do you see it in daily life?
- Why does a fixed-size array queue waste space after several dequeues?
- How does the circular queue fix that? State the key formulas.
- How do we know the queue is **full** vs **empty** in circular queues?
- What breaks if we **don't** maintain `count` ?

Next steps: Add `clear()` , `size()` , and `print_queue()` utilities; try resizing logic to double capacity while preserving order.