# NAMES:BYIRINGIRO Francois

# Reg:224013905

## 1) Core idea — LIFO vs FIFO (short reminder)

- **Stack (LIFO)**: last thing pushed is the first popped. Perfect for *undo* because you undo the most recent action first. Push/pop are O(1).
- **Queue (FIFO)**: first thing enqueued is the first dequeued. Perfect for *serving people* or ordering work fairly. Enqueue/dequeue are O(1) with the right structure.

---

## 2) MoMo Pay — stack deep dive (undo behavior)

Scenario: push `["Enter Amount", "Enter PIN", "Confirm Payment"]` then pop once.

Mechanics:

- UI sequence = a stack of *steps* or *commands*. When user presses “Back” or “Cancel last step”, app performs a `pop()`.
- After one pop the removed element is `"Confirm Payment"`. The top becomes `"Enter PIN"`.

Why stacks make sense:

- Users expect the **last thing they did** to be the first thing undone.
- Implementation is simple and fast: a stack in-memory per session or a stack of *command objects* that can be executed/rolled-back.

Important practical notes for payment flows:

- **Security**: never persist raw PINs in logs or long-lived storage. PINs should be handled in ephemeral memory and cleared immediately after use.
- **Irreversible actions**: a *confirmed* payment is normally irreversible by a simple UI undo. The UI stack can remove the confirmation screen but the backend transaction must be handled via secure reversal/refund flows — i.e., undo in UI ≠ undo in payment ledger.
- **Atomicity**: state transitions shown in the stack must map to transactional changes only at safe points (e.g., when `Confirm Payment` triggers a payment, that call is an atomic backend operation).

---

## 3) UR student scenario (stack of study items) — undo/redo and command patterns

Scenario: push `["Lecture Notes", "Practice Questions", "Mock Exam"]`, then pop twice → left is `"Lecture Notes"`.

Implementation options:

- **Simple stack of values** (good for UI navigation/history).
- **Command pattern**: store command objects that know how to `do()` and `undo()` — this supports complex undo (e.g., "revert last saved answer" across different data sources).
- **Undo/Redo**: use two stacks:
  - `undo_stack` — pushes every executed command.
  - `redo_stack` — when you `undo()`, pop from `undo_stack` and push onto `redo_stack`. When you `redo()`, pop from `redo_stack` and reapply then push onto `undo_stack`.
  - Any new user action clears `redo_stack`.

Pseudo-workflow (simple):

```
push "Lecture Notes"
push "Practice Questions"
push "Mock Exam"
pop() -> removes "Mock Exam"
pop() -> removes "Practice Questions"
top() -> "Lecture Notes"
```

Why command objects?

- They encapsulate operation + inverse and are safer when side effects involve databases or remote APIs.

---

## 4) Reversing a string using a stack — exact algorithm + complexity

String: `"RWANDA"`

Step-by-step:

1. Initialize empty stack.
2. Push characters in order: push('R'), push('W'), push('A'), push('N'), push('D'), push('A').
3. Pop all characters to build result: pop() -> 'A', then 'D', 'N', 'A', 'W', 'R'.
4. Result: `"ADNAWR"` (that's `RWANDA` reversed).

Complexity:

- Time: O(n) for n characters (push n times, pop n times).
- Space: O(n) extra for stack.

Minimal Python-like snippet:

```
def reverse_string(s):
    stack = []
    for ch in s:
        stack.append(ch)
    res = []
    while stack:
        res.append(stack.pop())
    return ''.join(res)

print(reverse_string("RWANDA"))  # -> ADNAWR
```

---

## 5) Queues: Airtel / RRA scenarios — fairness and richer queue design

Scenario: Airtel 5 customers enqueued [1,2,3,4,5]. After 3 served → front is customer 4.

Why FIFO fits service desks:

- Customers expect first-come-first-served. Queue preserves arrival order.

- For fairness and predictability, FIFO is the standard. If service has priorities (e.g., emergencies), use a *priority queue* or multiple queues.

Real-life queue complexities:

- **Multiple servers**: if there are c tellers the system becomes M/M/c in queueing theory; expected wait time needs arrival/service rates.
- **Starvation prevention**: priority queues can starve low-priority requests unless aging or fairness is used.
- **Ticketing systems**: give customers a ticket number (logical FIFO) so they can wait remotely.

Simple queue representation (Python-like):

```
from collections import deque
q = deque([1,2,3,4,5])
# 3 served:
for _ in range(3):
    q.popleft()
# front:
print(q[0])  # -> 4
```

---

# 6) Implementation techniques & algorithms (practical engineering)

Data structure choices:

- **Stack**:
  - Use an array/list where `push = append`, `pop = pop()` (end). O(1) amortized.
  - For persistent or immutable stacks, use functional persistent linked lists (sharing tails).
- **Queue**:
  - Use `collections.deque` in Python for O(1) append and popleft.
  - Implement queue with *two stacks* to get amortized O(1) enqueue and dequeue:
    - `in_stack` for pushes; `out_stack` for pops. When `out_stack` empty, move all from `in_stack` to `out_stack`.

Two-stacks-as-queue pseudocode:

```
enqueue(x): in_stack.push(x)
dequeue():
  if out_stack empty:
    while in_stack not empty:
      out_stack.push(in_stack.pop())
  return out_stack.pop()
```

Undo/Redo with two stacks (pseudo):

```
undo_stack = []
redo_stack = []

def do(command):
    command.do()
    undo_stack.append(command)
    redo_stack.clear()

def undo():
    if undo_stack:
        cmd = undo_stack.pop()
        cmd.undo()
        redo_stack.append(cmd)

def redo():
    if redo_stack:
        cmd = redo_stack.pop()
        cmd.do()
        undo_stack.append(cmd)
```

Edge behavior:

- Popping empty stack → underflow error; always check emptiness.
- Queues can grow; set limits or backpressure in production to avoid OOM.

---

## 7) Concurrency, distribution, and durability

Concurrency issues:

- Multiple threads/processes may act on a queue/stack. Protect with locks or use atomic operations.
- In distributed systems, use message brokers (durable queues) to guarantee delivery and ordering (with caveats: exact ordering is harder at scale).

Durability:

- For critical operations (payments), you don't rely only on in-memory stacks. Use persistent logs/transactions:
  - Append-only log for operations (event sourcing).
  - Snapshots + logs for replay/undo in complex systems.

Ordering at scale:

- Single-threaded UI stacks are easy. Distributed ordering and fairness often require sequence IDs or ticket servers.

## 8) UX / Product design concerns

Undo for actions vs. transactions:

- **UI-level undo** (e.g., going back between screens) is fine for stacks.
- **Business-level undo** (e.g., reversing a money transfer) needs formal inversion (refund, reversal) and audit trail.

Guidelines:

- Limit undo depth or show history so users know what they can undo.
- Use confirmations for irreversible actions (e.g., “Confirm payment — this will be processed”).
- Provide a **redo** option if users commonly toggle undo/redo.

Performance & visibility:

- Keep stack operations local to the session for speed.
- Persist only committed states to server; keep UI navigation stack ephemeral.

Security (especially for MoMo / PINs):

- Never log or persist sensitive inputs like PINs.
- Use secure input fields and erase buffers after use.
- For audit, store *events* (e.g., payment-of-X) but not raw secrets.

## 9) Testing & edge cases to try

- Undo all items until stack empty — assert proper underflow handling.
- Do undo then do a new operation — verify redo stack cleared.
- Simulate concurrent enqueues and dequeues — verify ordering and no races.
- For queues with priority, simulate starving lower-priority entries and add aging.

## 10) Practical recommendations (short list)

1. Use a **stack** (LIFO) for navigation and undo in UI; use a **command pattern** if actions have side effects.

2. For payments: use UI stack for screens but treat payment confirmation as a backend transaction with its own lifecycle (logs, refunds).
3. For serving customers: use **queues** or ticketing system; if prioritization is required, use priority queues with aging to avoid starvation.
4. In code use `deque` for queues and list append/pop for stacks in high-level languages.
5. For undo/redo, use two stacks and clear the redo stack when a new action occurs.

---

## TL;DR (very short)

- Stack = undo (LIFO). Queue = serve in arrival order (FIFO).
- `RWANDA → ADNAWR` via pushing each char and popping all.
- Implement undo/redo with two stacks. Use `deque` for efficient queues.
- For payment systems, treat UI undo separately from ledger operations (security & audit required).

---

If you want, I can now:

- 
    1. **Produce runnable Python code** showing: the MoMo stack example, reversing `"RWANDA"`, an undo/redo demo, and the Airtel queue example — with printed outputs formatted like "screenshot-style".
- 
    2. Or I can design a short **class diagram / flowchart** (text) of how a real app maps UI stacks to backend transaction logs.