UNIVERSITY OF RWANDA

HUYE COMPUS

MODULE:Data Structure and Algorithm

LECTURER:RUKUNDO Prince

Names:DUFITIMANA chance

Reg No:224004922

Part I – STACK

Q1: How does this show the LIFO nature of stacks?

- In MTN MoMo, when you fill details step by step, the *last step you entered* (like amount) is the *first one removed* when you press **back**.
- This reflects **LIFO (Last In, First Out)** because the *most recent step* is undone first.

Q2: Why is this action similar to popping from a stack?

- In UR Canvas, pressing **back** removes the last page or module you visited.
- Just like **pop()** removes the **top item** from a stack, the app removes the most recent navigation step.

Q3: How could a stack enable the undo function when correcting mistakes?

- Each action (typing, transaction, or entry) is **pushed** onto a stack.
- If you make a mistake, the system simply **pops** the most recent action (undo), restoring the previous state.

Q4: How can stacks ensure forms are correctly balanced?

- When filling Irembo forms:
  - Each **opening field (e.g., start of section)** is *pushed* onto the stack.
  - Each **closing field (end of section)** must correctly match the **top item** on the stack.
- If at the end the stack is empty, the form is correctly filled; otherwise, it is unbalance
-
- **Q5: Which task is next (top of stack)?**
  Steps:

1. Push("CBE notes") → Stack = [CBE notes]
2. Push("Math revision") → Stack = [CBE notes, Math revision]
3. Push("Debate") → Stack = [CBE notes, Math revision, Debate]
4. Pop() → removes "Debate" → Stack = [CBE notes, Math revision]
5. Push("Group assignment") → Stack = [CBE notes, Math revision, Group assignment]

✅ **Top of stack = "Group assignment"**

Q6: Which answers remain in the stack after undoing?

- If a student undoes **3 recent actions**, that means the system performs **3 pops**.
- After removing those 3 most recent entries, only the **earlier answers** remain in the stack.
- Example: If the stack was [Q1, Q2, Q3, Q4, Q5], undoing 3 pops leaves [Q1, Q2].

✅ **Remaining = the first (earlier) actions that were not undone.**

Part I – STACK (Advanced Thinking)

Q7: How does a stack enable this retracing process?

- In RwandAir booking, each completed step (passenger info, flight choice, payment, etc.) is **pushed** onto a stack.
- When the passenger presses **back**, the system **pops** the last step, showing the previous one.
  ✅ This works because a stack naturally supports **backtracking** by removing the latest step first.

Q8: Show how a stack algorithm reverses the proverb "Umwana ni umutware."

Steps:

1. **Push** each word:
   - Push("Umwana"), Push("ni"), Push("umutware")
   - Stack = [Umwana, ni, umutware]
2. **Pop** words one by one:
   - Pop → "umutware"
   - Pop → "ni"
   - Pop → "Umwana"
3. Output in popped order:
   ✅ "umutware ni Umwana"

Q9: Why does a stack suit this case better than a queue?

- In **DFS (Depth-First Search)**, you go as deep as possible before backtracking.
- A **stack** is ideal because it remembers the **most recent node/branch** visited and returns there first when backtracking.
- A **queue**, on the other hand, is better for **BFS (Breadth-First Search)** since it processes level by level.

✅ DFS needs *last visited, first backtracked* → which is exactly how a stack works.

Q10: Suggest a feature using stacks for transaction navigation.

- In BK Mobile app, when checking **transaction history**:
  - Each movement forward (viewing a newer transaction) is **pushed** onto the stack.
  - Pressing **back** pops the last viewed transaction, showing the previous one.

Suggested feature: **"Undo last navigation"** → letting users quickly return to the previous transaction details by popping from the stack.

Part II – QUEUE

Q1: How does this show FIFO behavior?

- In a Kigali restaurant, customers are served in the **same order they arrived**.
- The **first customer in line** is the **first to be served** → this is exactly **FIFO (First In, First Out)**.

Q2: Why is this like a dequeue operation?

- In a YouTube playlist, the **first video in the queue** starts playing.
- After it finishes, it is **removed (dequeue)**, and the **next video** automatically plays.
  This matches **dequeue()**, which removes items from the **front of the queue**.

Q3: How is this a real-life queue?

- At RRA offices, people line up to pay taxes:
  - New arrivals join at the **rear (enqueue)**.
  - The tax officer serves from the **front (dequeue)**.
    This is a perfect example of **queue structure in daily life**.

Q4: How do queues improve customer service?

- By serving people **in the order they arrived**, queues prevent cheating or jumping ahead.
- They make service **fair, organized, and efficient**.
- Staff can handle requests smoothly without confusion.

Queues ensure **fairness + order → better customer experience**.

Part II – QUEUE (C & D)

Q5: Sequence (Equity Bank)

Operations:

- Enqueue("Alice") → Queue = [Alice]
- Enqueue("Eric") → Queue = [Alice, Eric]
- Enqueue("Chantal") → Queue = [Alice, Eric, Chantal]
- Dequeue() → removes **Alice** → Queue = [Eric, Chantal]
- Enqueue("Jean") → Queue = [Eric, Chantal, Jean]

Answer: Eric is at the front now.

Q6: FIFO message handling (RSSB pensions): how a queue ensures fairnessA queue serves in **arrival order**:

- whoever arrived first is served first → prevents cutting in line.
- Implementation details that preserve fairness: assign each application a **timestamp** or sequence number on arrival; always dequeue the smallest timestamp first.
- Additional fairness safeguards: single shared queue for all servers (avoids server-specific queue-jumping), and timeout/retry rules so stalled items don't block the whole queue.

D. Advanced Thinking

Q7: Mapping queue types to Rwandan life examples

- **Linear queue (single-ended FIFO)** — *People at a wedding buffet*: join at rear, leave at front; once served, slot is gone. Simple FIFO with fixed start/end.
- **Circular queue (ring buffer)** — *Buses looping at Nyabugogo*: buses arrive and re-enter the loop; the "slot" of a bus can be reused after it completes a loop. Useful when resources are reused and capacity is fixed.
- **Deque (double-ended queue)** — *Boarding a bus from front/rear*: passengers may enter or exit from either end depending on the bus layout or priority boarding (front for disabled, rear for others). Deque supports enqueue/dequeue at both ends.

Each maps to how items enter/leave and whether capacity/resources are reused or both ends are allowed.

Q8: Restaurant orders called when ready: how queues model this
Model with two queues or a single order queue plus a ready-notify system:

1. **Order Queue (kitchen)** — customers' orders are enqueued as they place them. Kitchen dequeues next order to prepare (FIFO or priority-based for rush items).
2. **Ready Queue / Notification** — when an order is finished it is placed in a "ready" list or a notification is sent to the customer (dequeue from ready queue when picked up).
3. Variations:
   - Use **ticket numbers** (enqueue number) → display the next ready number.
   - Use **multiple queues** for different stations (grill, drinks) and a final assembly queue.
   - Support **callbacks/notifications** so customer doesn't block a physical spot, but fairness remains via ticket numbers.

This prevents confusion, ensures first-come-first-served for preparation, and lets customers know exactly when to collect.

Q9: Why CHUK emergencies = priority queue, not normal queue

- In a normal queue all items treated equally (FIFO). In hospital triage, **some cases are far more urgent**.
- A **priority queue** associates a priority (e.g., severity score) with each patient; dequeue returns the highest-priority patient first.
- This allows life-saving deviation from FIFO while still structured — e.g., a critical emergency jumps ahead of routine cases.

Q10: Fair matching in a moto/e-bike taxi app (drivers ↔ students)

Goal: match fairly and efficiently. Suggested approach combines queues + priority rules:

Basic FIFO match (simple fairness)

- Maintain two queues: `DriversQueue` (available drivers) and `RidersQueue` (waiting students). When a rider arrives, enqueue them; when a driver signals availability, if `RidersQueue` non-empty → dequeue earliest rider and assign driver (or vice-versa).
- This ensures "first requester" fairness.

Practical improvements (real-world fairness & efficiency):

1. **Geographic bucketing:** maintain separate FIFO queues per zone/sector so matches are local (reduces idle time).
2. **Distance-aware priority:** within a zone, favor riders closest to drivers but ensure long-wait riders gradually get priority (aging).
   - Example: score = base_priority + wait_time_weight × waiting_seconds − distance_penalty × distance_km. Dequeue highest score.
3. **Round-robin/driver fairness:** rotate assignments so drivers receive fairly distributed rides (avoid always sending rides to the same drivers).
4. **Timeouts & requeueing:** if rider/driver ignores match, re-enqueue with bumped priority to avoid starvation.
5. **Batching for efficiency:** if multiple riders heading same direction appear, allow driver to accept pooled ride (with consent).
6. **Transparent queue position:** show riders their queue position/estimated wait to reduce cancellations.

Algorithm sketch (zone-based):

- On rider request: enqueue rider in `ZoneQueue[z]` with `timestamp` and `location`.
- On driver available: look in `ZoneQueue[z]` → if empty, optionally search neighbouring zones; otherwise compute match score (aging + distance) and pick best rider; dequeue and assign.
- Periodically increase waiting riders' priority so long-waiters are matched first if other factors equal.
- This balances **fairness (FIFO/aging)** and **practical efficiency (distance, drive distribution)**.