NAMES: NDAHIRIWE OLIVIER
REG:224015960
BIT: Year two(2)
Work of Data structure

Part I: Stacks - The LIFO Principle and Its Applications

A **Stack** is an abstract data type that follows the **LIFO (Last-In, First-Out)** principle, meaning the last item added to the structure is the first one to be removed. The two primary operations are **Push** (adding an item to the top) and **Pop** (removing the top item).

1. Stack Mechanism and LIFO Examples

The LIFO behavior is fundamental to how a stack operates and is clearly demonstrated in real-world scenarios:

- **Payment Detail Reversal (Q1):** In the MTN MoMo app, pressing the 'back' button removes the most recently entered payment detail step. The steps entered follow the sequence: *_182#_ → *1 → *_1_ → *Receipt number → *_Amount number_ → *pin#. The $*$pin# (Last-In) is the first to be removed (First-Out), confirming LIFO.
- **Undo Functionality (Q2, Q3):** The 'undo' action in applications like UR Canvas removes the last step taken. This directly corresponds to a **Pop** operation on a history stack, which retrieves and removes the most recent action. Every successful transaction state is **Pushed** onto the stack, and a mistake is corrected by **Popping** the last action.
- **Task Management (Q5):** The sequence of tasks (CBE notes, Math revision, Debate, Group assignment) demonstrates the stack operation. After pushing all items, a **Pop()** removes "Debate" (the top item). The final task at the top of the stack after all operations is **"Group assignment"**.

2. Correction, Validation, and Backtracking

The stack's LIFO property makes it ideal for tasks that require reversing a sequence or validating nested structures:

- **Correcting Mistakes (Q6):** When a student 'undoes' three recent actions, three **Pop** operations are performed. Because of LIFO, the three most recently added answers (the ones at the top) are removed, leaving the earlier entered answers at the bottom of the stack.

- **Form Balancing/Validation (Q4):** Stacks are used to ensure forms, brackets, or code tags are correctly nested (balanced). Every opening item is **Pushed** onto the stack. When a closing item is encountered, the stack is **Popped** to check if the retrieved opening item is the correct match. An empty stack upon finding a closing item, or a mismatch, indicates an **unbalanced** structure.
- **Backtracking/Form Retracing (Q7):** The stack enables the retracing of a multi-step form. As a passenger moves to a new step, the current state is **Pushed**. Pressing "back" triggers a **Pop** operation, which retrieves the previous step's state, allowing the form to revert.
- **Proverb Reversal (Q8):** The proverb "Umwana ni umutware" is reversed by pushing each word onto the stack: ["Umwana", "ni", "umutware"]. Since the words are then popped in LIFO order ("umutware", "ni", "Umwana"), the order of the entire string is reversed, resulting in **"umutware ni Umwana"**.

3. Stack vs. Queue for Search and Navigation

The choice between a stack and a queue depends on the required exploration pattern:

- **Depth-First Search (DFS) (Q9):** A stack is better suited for a **deep search** (DFS). When a search hits a new path (e.g., a student checking a new aisle), the current location is **Pushed** (saved). When a dead end is reached, a **Pop** operation enables backtracking to the **most recently** saved location to try a different route. A Queue is used for Breadth-First Search (BFS), which explores all immediate neighbors first.
- **Transaction Navigation (Q10):** Stacks can implement a "Recently Viewed Transaction Details" feature. The transaction ID is **Pushed** when viewed, and a "Go Back to Previous Transaction" button executes a **Pop**, immediately returning the user to the details of the transaction viewed **just before** the current one.

Part II: Queues - The FIFO Principle and Its Applications

A **Queue** is an abstract data type that follows the **FIFO (First-In, First-Out)** principle, meaning the first item added to the structure is the first one to be removed. The two primary operations are **Enqueue** (adding an item to the rear/back) and **Dequeue** (removing an item from the front).

1. Queue Mechanism and FIFO Examples

The FIFO nature of a queue ensures strict sequential processing based on arrival time:

- **Restaurant Service (Q1, Q3):** This is a classic real-life queue. The **First** customer to arrive (First-In) is the **First** one to be served (First-Out), directly illustrating the **FIFO** principle. People **Enqueue** at the rear and are served (Dequeue) from the front.
- **YouTube Playlist (Q2):** Playing the next video is analogous to a **Dequeue** operation. The current video (at the front) is removed from the 'to-be-played' state, and the next video takes its place at the front, ready to be "dequeued" (played).

- **Personnel Order (Q5):** The sequence of operations demonstrates queue dynamics. After Enqueueing Alice, Eric, Chantal, Dequeueing removes Alice. The final Enqueue of Jean places him at the rear. The person at the **front** of the queue now is **Eric**.

2. Fairness, Service, and System Modeling

The queue structure is essential for providing fair, orderly, and efficient service:

- **Fairness and Predictability (Q4, Q6):** Queues improve customer service by ensuring **fairness**. By strictly processing requests in the order of arrival (FIFO), they guarantee that every customer is treated equally, preventing favoritism. This structured system allows for better estimation of wait times and efficient management of service flow. For handling applications, a queue ensures the **first application submitted is the first one processed**, establishing an objective and equitable system.
- **Modeling Complex Processes (Q8):** A multi-stage process like a restaurant can be modeled using two queues:
    1. **Order Queue:** Customer orders are **Enqueued** here and are **Dequeued** by the kitchen for processing (FIFO).
    2. **Ready Queue (Holding Area):** Completed orders are **Enqueued** here. Customers are called to pick up their food by **Dequeueing** the ready items. This separation manages workflow and collection efficiently.
- **Ride-Hailing Matching (Q10):** A system can be built using two queues: a **Driver Queue** and a **Passenger Queue**. Available drivers and requesting passengers **Enqueue** into their respective lines. The matching system then repeatedly tries to **Dequeue** one item from the front of each queue. This ensures that the **longest-waiting driver** is matched with the **longest-waiting passenger**, promoting fairness for both parties.

3. Types of Queues in Rwandan Context

Queues can be implemented with various structural optimizations depending on the need:

Queue Type	
Linear Queue	People at a wedding’s food table
Circular Queue	Buses looping at vehicle station like in Nyabugogo,kinyinya etc
Deque (Double-Ended Queue)	Boarding a bus from front/rear
Priority Queue	Hospital Emergency Room