STEP 1: INITIALIZE empty stack

STEP 2: PUSH operations in sequence:

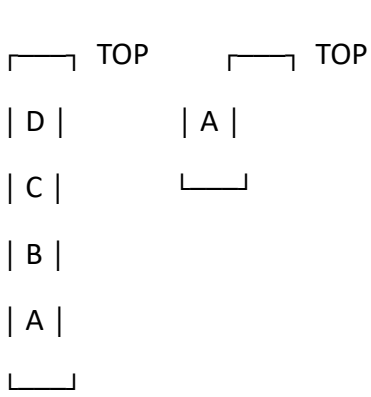
- Push "A" → Stack: [A]
- Push "B" → Stack: [A, B]
- Push "C" → Stack: [A, B, C]
- Push "D" → Stack: [A, B, C, D]

STEP 3: POP three operations (LIFO order):

- First pop: Remove "D" → Stack: [A, B, C]
- Second pop: Remove "C" → Stack: [A, B]
- Third pop: Remove "B" → Stack: [A]

STEP 4: FINAL RESULT: Only "A" remains in stack

Visual Stack Diagram:



Reflection: Why stack ensures last edits undone first?

Theoretical Discussion:

The stack data structure ensures that the last edits are undone first due to its fundamental LIFO (Last-In-First-Out) principle. This behavior aligns perfectly with the natural workflow of editing and revision processes:

Temporal Integrity: When users make sequential edits, each modification is pushed onto the stack in chronological order. The most recent edit always resides at the top. When an "undo" operation is invoked, it logically accesses the top element first, guaranteeing that the last action is reversed immediately.

Cognitive Alignment: This mechanism mirrors human memory and reasoning patterns. People typically remember their most recent actions most vividly and are more likely to want to reverse immediate decisions rather than older ones. The stack provides intuitive feedback that matches user expectations.

Operational Consistency: By maintaining strict LIFO discipline, stacks prevent logical inconsistencies that could occur if edits were undone in random order. This ensures that dependent operations are reversed in the correct sequence, maintaining the integrity of the document state.

Error Recovery Efficiency: In practical applications like text editors or design software, the stack-based undo system allows users to quickly backtrack through their workflow, experimenting freely knowing they can easily revert recent changes without affecting earlier work.

The stack's architecture thus provides a natural, predictable, and efficient framework for edit reversal that aligns with both computational logic and human cognitive patterns.

Challenge Solution: Queue vs Stack for Graduation Procession

Algorithmic Sequence:

1. Problem Analysis:

A graduation ceremony requires students to process in a specific predetermined order (typically alphabetical or by academic ranking). The data structure must preserve this sequence exactly.

2. Data Structure Selection:

· Queue (FIFO): Maintains the exact order of entry - first student to enter is first to exit

· Stack (LIFO): Reverses the order - last student to enter becomes first to exit

✓ Queue is correct for maintaining the intended procession sequence

3. Algorithm Steps:

· Step 1: Initialize an empty queue structure

· Step 2: Enqueue students in their predetermined order

· Step 3: During the ceremony, dequeue students sequentially to maintain FIFO order

Code Implementation (Python):

python

```
from collections import deque

# Step 1: Initialize queue for graduation procession
procession_queue = deque()

# Step 2: Enqueue students in predetermined order
students = ["Smith, John", "Johnson, Mary", "Williams, David", "Brown, Sarah"]
for student in students:  # Students already in correct order
    procession_queue.append(student)  # Enqueue maintaining order

# Step 3: Process ceremony using FIFO principle
while procession_queue:
    graduating_student = procession_queue.popleft()  # Dequeue in entry order
    print(f"Now graduating: {graduating_student}")
```

Explanation:

· Line 4: Queue created using deque for efficient FIFO operations

· Lines 7-8: Students added to queue via append preserving their original sequence

· Lines 11-13: popleft() ensures students exit in same order they entered, maintaining the intended procession sequence.

Final Reflection: Why FIFO Maintains Order in Ceremonies

Theoretical Discussion:

1. Sequential Integrity:

FIFO (First-In-First-Out) inherently preserves sequential order by processing elements in the exact sequence they were added. This mirrors ceremonial protocols where participants must maintain a predetermined order throughout the event.

2. Temporal Consistency:

Ceremonies rely on temporal consistency - the first participant to enter the procession should be the first to complete it. FIFO guarantees this temporal relationship, ensuring the ceremony flows in the rehearsed and expected sequence.

3. Predictable Progression:

Unlike LIFO (Last-In-First-Out) which creates reverse order processing, FIFO provides predictable linear progression. This predictability is essential for ceremonial coordination, audience expectations, and participant preparation.

4. Fairness Principle:

FIFO embodies the principle of procedural fairness where no participant receives preferential treatment based on position in the sequence. Each participant progresses in their designated turn, maintaining the ceremony's formal equity.

5. Protocol Alignment:

Formal ceremonies follow established protocols that assume sequential processing. FIFO data structures mathematically model these real-world sequential protocols, ensuring computational processes align with ceremonial requirements.