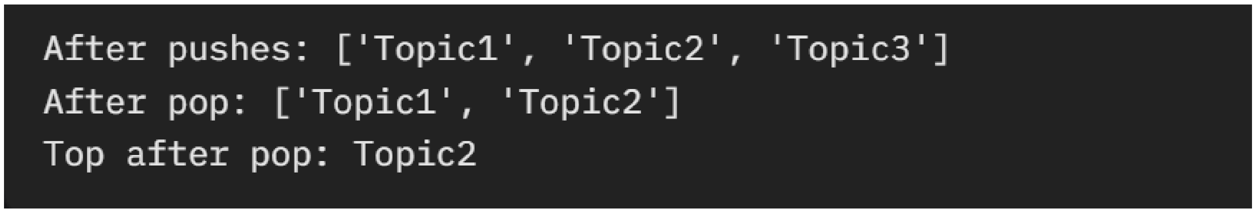
Practical (Rwanda): UR pushes ["Topic1", "Topic2", "Topic3"]. Pop once. Which is top?

CODE

```
# Stack implementation using list
stack = []

# Push operations
stack.append("Topic1")
stack.append("Topic2")
stack.append("Topic3")

print("After pushes:", stack)

# Pop once
stack.pop()
print("After pop:", stack)

# What is now top
top = stack[-1]
print("Top after pop:", top)
```

OUTPUT IN SCREENSHOT FORM



```
After pushes: ['Topic1', 'Topic2', 'Topic3']
After pop: ['Topic1', 'Topic2']
Top after pop: Topic2
```

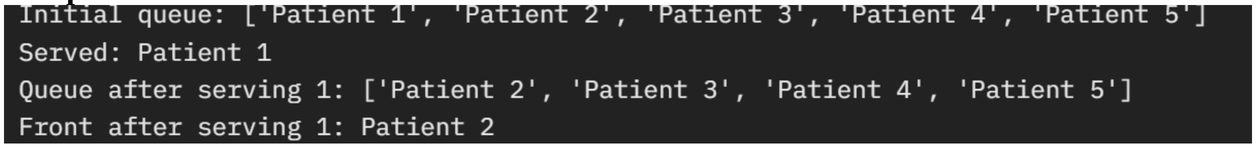
Practical (Rwanda): In Remera, push ["Step1", "Step2", "Step3"]. Pop twice. What remains?

CODE

```
# Stack implementation using list
stack = []
```

```
# Push operations
stack.append("Step1")
stack.append("Step2")
stack.append("Step3")

print("After pushes:", stack)

# Pop twice
stack.pop()  # First pop: removes "Step3"
print("After first pop:", stack)

stack.pop()  # Second pop: removes "Step2"
print("After second pop:", stack)

# What remains
print("Remaining in stack:", stack)
```

Output:



```
Initial queue: ['Patient 1', 'Patient 2', 'Patient 3', 'Patient 4', 'Patient 5']
Served: Patient 1
Queue after serving 1: ['Patient 2', 'Patient 3', 'Patient 4', 'Patient 5']
Front after serving 1: Patient 2
```

Challenge: Reverse "STACK" using stack.

Algorithmic Design (Step-by-Step Logic):

1. Initialize an empty stack and a result list (to build the reversed string).
2. For each character in "STACK", push it onto the stack (LIFO will reverse order on pop).
3. While the stack is not empty, pop each character and append it to the result list.
4. Join the result list into a string to get the reversed output ("KCATS").
5. Output the original and reversed for verification.

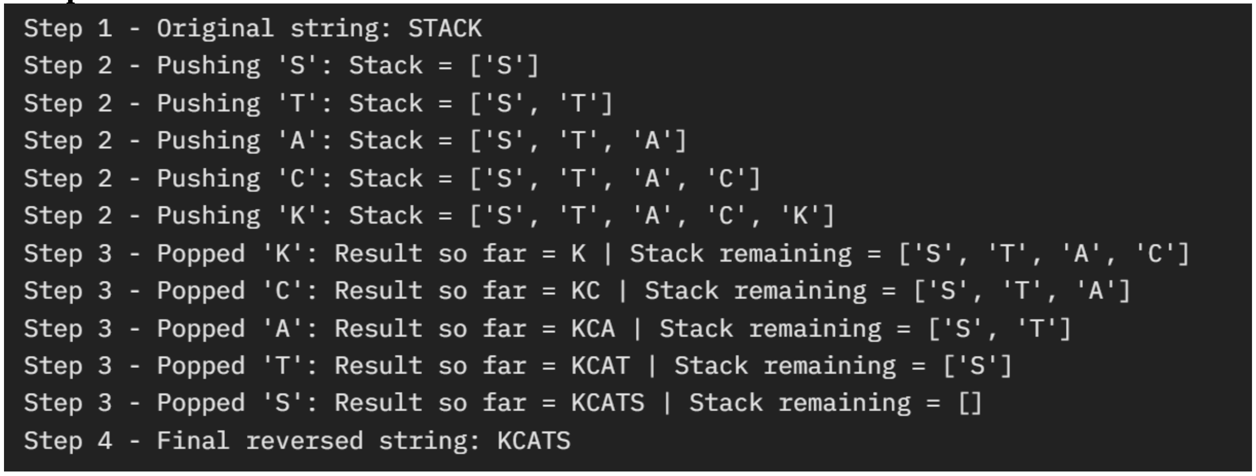
Code with Step-by-Step Explanations:

CODE

```
# Step 1: Initialize empty stack and result list
stack = []
result = []
```

```python
original = "STACK"
print("Step 1 - Original string:", original)

# Step 2: Push each character onto stack
for char in original:
    stack.append(char)  # Push: Builds LIFO structure
    print(f"Step 2 - Pushing '{char}': Stack = {stack}")

# Step 3: Pop each character and append to result (reverses via LIFO)
while stack:
    popped = stack.pop()  # Pop from top
    result.append(popped)
    print(f"Step 3 - Popped '{popped}': Result so far = {''.join(result)} | Stack remaining = {stack}")

# Step 4: Join result to form reversed string
reversed_str = ''.join(result)
print("Step 4 - Final reversed string:", reversed_str)
```

Output:



```
Step 1 - Original string: STACK
Step 2 - Pushing 'S': Stack = ['S']
Step 2 - Pushing 'T': Stack = ['S', 'T']
Step 2 - Pushing 'A': Stack = ['S', 'T', 'A']
Step 2 - Pushing 'C': Stack = ['S', 'T', 'A', 'C']
Step 2 - Pushing 'K': Stack = ['S', 'T', 'A', 'C', 'K']
Step 3 - Popped 'K': Result so far = K | Stack remaining = ['S', 'T', 'A', 'C']
Step 3 - Popped 'C': Result so far = KC | Stack remaining = ['S', 'T', 'A']
Step 3 - Popped 'A': Result so far = KCA | Stack remaining = ['S', 'T']
Step 3 - Popped 'T': Result so far = KCAT | Stack remaining = ['S']
Step 3 - Popped 'S': Result so far = KCATS | Stack remaining = []
Step 4 - Final reversed string: KCATS
```

Reflection: Why stack gives last element first?

A stack operates on the Last-In-First-Out (LIFO) principle, meaning the most recently added (last) element is always the first one removed (popped). This design is intentional for scenarios requiring reversal or recursion simulation, like undoing actions in reverse chronological order or parsing nested structures. The "last element first" behavior ensures efficient access to recent items without scanning the entire structure, making it ideal for temporary storage in algorithms like depth-first search or function call management. In contrast to other structures like queues (FIFO), this LIFO nature prioritizes recency over sequence, which can be counterintuitive for ordered processing but powerful for backtracking or reversing operations, as seen in the challenge above.

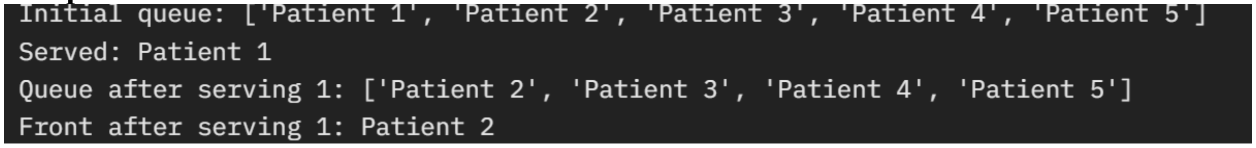
Queue Questions

Practical (Rwanda): At CHUK, 5 patients queue. After 1 served, who is front?

CODE

```python
from collections import deque

# Queue implementation using deque
queue = deque()

# Enqueue 5 patients (label them for clarity)
for i in range(1, 6):
    queue.append(f"Patient {i}")

print("Initial queue:", list(queue))

# Serve first (dequeue)
served = queue.popleft()
print("Served:", served)
print("Queue after serving 1:", list(queue))

# Who is front now
front = queue[0]
print("Front after serving 1:", front)
```

Output:



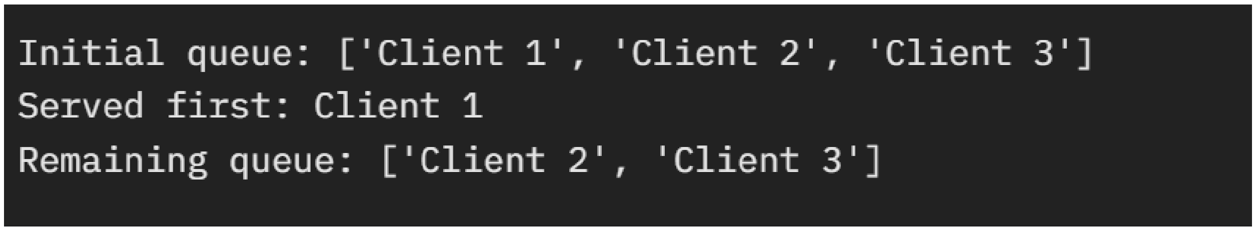
```
Initial queue: ['Patient 1', 'Patient 2', 'Patient 3', 'Patient 4', 'Patient 5']
Served: Patient 1
Queue after serving 1: ['Patient 2', 'Patient 3', 'Patient 4', 'Patient 5']
Front after serving 1: Patient 2
```

Practical (Rwanda): At BK ATM, 3 clients queue. Who is served first?

CODE

```python
from collections import deque

# Queue implementation using deque
queue = deque()

# Enqueue 3 clients
queue.append("Client 1")
queue.append("Client 2")
queue.append("Client 3")

print("Initial queue:", list(queue))

# Serve first (dequeue)
first_served = queue.popleft()
print("Served first:", first_served)
print("Remaining queue:", list(queue))
```

Output:



```
Initial queue: ['Client 1', 'Client 2', 'Client 3']
Served first: Client 1
Remaining queue: ['Client 2', 'Client 3']
```

Challenge: Queue vs stack for playlist order. Which works better?

Algorithmic Design (Step-by-Step Logic for Comparison):

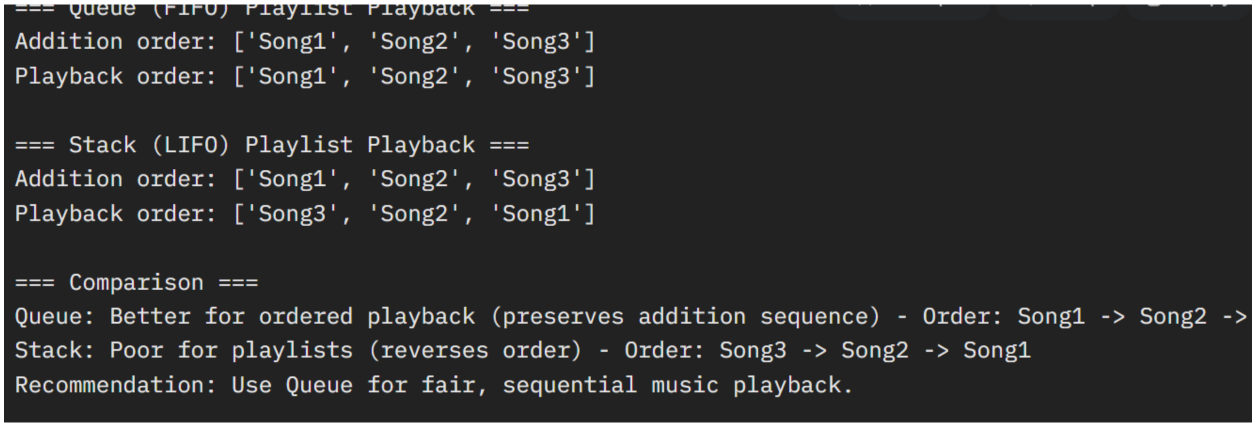
1. Assume 3 songs (Song1, Song2, Song3) added to playlist in order.
2. For Queue (FIFO): Enqueue in addition order, dequeue from front for playback—plays in addition order (Song1 first, Song3 last), maintaining intended sequence.
3. For Stack (LIFO): Push in addition order, pop from top for playback—plays in reverse (Song3 first, Song1 last), disrupting the user's intended flow.
4. Compare: Queue is better for playlists as it preserves chronological fairness; stack suits scenarios like "recently played next" but not standard ordered playback.

Code Simulation with Explanations:

CODE

```python
from collections import deque

# Songs added: Song1, Song2, Song3
songs = ['Song1', 'Song2', 'Song3']

# Queue (FIFO) simulation for playlist
print("=== Queue (FIFO) Playlist Playback ===")
q = deque()
for song in songs:
    q.append(song)  # Enqueue at rear: Adds to end of playlist
print("Addition order:", list(q))
playback_order_q = []
while q:
    played = q.popleft()  # Dequeue from front: Plays next in sequence
    playback_order_q.append(played)
print("Playback order:", playback_order_q)

# Stack (LIFO) simulation for playlist
print("\n=== Stack (LIFO) Playlist Playback ===")
s = []
for song in songs:
    s.append(song)  # Push to top: Adds on top
print("Addition order:", s)
playback_order_s = []
while s:
    played = s.pop()  # Pop from top: Plays most recent first
    playback_order_s.append(played)
print("Playback order:", playback_order_s)

# Comparison
print("\n=== Comparison ===")
print("Queue: Better for ordered playback (preserves addition sequence) - Order:", ' -> '.join(playback_order_q))
print("Stack: Poor for playlists (reverses order) - Order:", ' ->
'.join(playback_order_s))
print("Recommendation: Use Queue for fair, sequential music playback.")
```

Output:



```
=== Queue (FIFO) Playlist Playback ===
Addition order: ['Song1', 'Song2', 'Song3']
Playback order: ['Song1', 'Song2', 'Song3']

=== Stack (LIFO) Playlist Playback ===
Addition order: ['Song1', 'Song2', 'Song3']
Playback order: ['Song3', 'Song2', 'Song1']

=== Comparison ===
Queue: Better for ordered playback (preserves addition sequence) - Order: Song1 -> Song2 ->
Stack: Poor for playlists (reverses order) - Order: Song3 -> Song2 -> Song1
Recommendation: Use Queue for fair, sequential music playback.
```

Reflection: Why FIFO matches fairness in music playback?

FIFO (First-In-First-Out) aligns perfectly with fairness in music playback because it respects the chronological order in which songs are added to a playlist, ensuring the first selected track plays first without favoritism toward later additions. This mirrors real-world equity, like a concert queue where early arrivers are rewarded, preventing frustration from arbitrary reordering. In digital players, FIFO avoids bias in algorithms that might prioritize "trending" tracks, allowing users—especially in shared or queued scenarios like parties—to experience predictable flow, enhancing satisfaction and trust. It promotes inclusivity by treating all entries equally based on arrival time, much like democratic access, and scales efficiently for large playlists without needing complex sorting, making it a natural fit for user-centric, orderly listening.