Editorials : Queue Assignment

Question : Waiting line

```
# Read the number of operations
N = int(input())

# Initialize an empty queue using a list
queue = []

# Process each of the N operations
for _ in range(N):
    # Read the operation as a list of strings
    operation = input().split()

    if operation[0] == 'E':
        # Enqueue operation: append the integer to the queue
        x = int(operation[1])
        queue.append(x)
        # Print the new size of the queue
        print(len(queue))

    elif operation[0] == 'D':
        # Dequeue operation
        if queue:
            # If the queue is not empty, remove and get the front element
            removed = queue.pop(0)
            # Print the removed element and the new size of the queue
            print(removed, len(queue))
        else:
            # If the queue is empty, print -1 and size (0)
            print(-1, 0)
```

Explanation

1. **Reading the Number of Operations**

   ```
   N = int(input())
   ```

   - The first line of input contains an integer `N`, which represents the total number of operations.

2. **Initializing the Queue**

```
queue = []
```

   - We use a Python list named `queue` to simulate queue behavior (FIFO: First In, First Out).

3. **Processing Each Operation**

```
for _ in range(N):
    operation = input().split()
    ...
```

   - We read each operation line by line.
   - `operation` becomes a list of strings. For example:
     - `["E", "5"]` could represent an enqueue operation with the value 5.
     - `["D"]` could represent a dequeue operation.

4. **Enqueue Operation (`E`)**

```
if operation[0] == 'E':
    x = int(operation[1])
    queue.append(x)
    print(len(queue))
```

   - If the first element in `operation` is `"E"`, this indicates an **enqueue**.
   - We convert the second element (the number to be enqueued) to an integer and append it to the end of `queue`.
   - We then print the **new size** of the queue (i.e., `len(queue)`).

5. **Dequeue Operation (`D`)**

```
elif operation[0] == 'D':
    if queue:
        removed = queue.pop(0)
        print(removed, len(queue))
    else:
        print(-1, 0)
```

   - If the first element is `"D"`, this indicates a **dequeue**.
   - We check if the queue is **not empty**:

        - If it has at least one element, we remove the front element using `pop(0)` (this removes and returns the element at index 0).
        - We print the **removed element** and the **new size** of the queue.
    - If the queue is empty, we print `-1` (to indicate that no element could be dequeued) and `0` (the size of the queue).

How It Works Overall

- You have `N` operations in total.
- Each operation is either:
    i. **`E x`**: Enqueue `x` into the queue. The code appends `x` to the list and prints the updated size.
    ii. **`D`**: Dequeue from the queue. The code removes the first element of the list (if any) and prints the removed element along with the updated size.
- If a dequeue is attempted on an empty queue, the code prints `-1 0`.
- By the end of all operations, each enqueue and dequeue request has been processed, with appropriate outputs after each operation.

Question : Implement Queue using Stack

```python
class Queue:
    def __init__(self):
        self.s1 = []  # Stack for enqueue
        self.s2 = []  # Stack for dequeue and peek

    def enqueue(self, value):
        # Add element to s1
        self.s1.append(value)

    def dequeue(self):
        # Pop and return top element
        # If s2 is not empty, pop from s2
        if self.s2:
            return self.s2.pop()
        # If s2 is empty, transfer elements from s1 to s2
        while self.s1:
            self.s2.append(self.s1.pop())
        # Then pop from s2
        return self.s2.pop() if self.s2 else None

    def peek(self):
```

```
        # Return the element at the front of the queue without removing it
        # If s2 is not empty, the front is on top of s2
        if self.s2:
            return self.s2[-1]
        # If s2 is empty, move all elements from s1 to s2
        while self.s1:
            self.s2.append(self.s1.pop())
        # Now the front is on top of s2
        return self.s2[-1] if self.s2 else None

    def isEmpty(self):
        # Returns True if both stacks are empty
        return not self.s1 and not self.s2
```

Explanation

1. Two Stacks: `s1` and `s2`

- `s1` is used for enqueue operations.
- `s2` is used for dequeue (and peek) operations.

2. `enqueue(value)`

```
def enqueue(self, value):
    self.s1.append(value)
```

- To enqueue an item (i.e., add it to the queue), we push it onto the top of `s1`.
- This operation is **O(1)**.

3. `dequeue()`

```
def dequeue(self):
    if self.s2:
        return self.s2.pop()
    while self.s1:
        self.s2.append(self.s1.pop())
    return self.s2.pop() if self.s2 else None
```

- If `s2` is **not empty**, we can simply pop from `s2`. The top of `s2` corresponds to the **front** of the queue.

- If `s2` is **empty**, we transfer all elements from `s1` to `s2` by popping from `s1` and pushing onto `s2`.
    - This reversal makes the top of `s2` the front of the queue.
- Finally, we pop from `s2`.
- If both `s1` and `s2` end up empty, there is nothing to dequeue, so we return `None`.
- Amortized time complexity is **O(1)**: each element is moved from `s1` to `s2` at most once.

4. `peek()`

```python
def peek(self):
    if self.s2:
        return self.s2[-1]
    while self.s1:
        self.s2.append(self.s1.pop())
    return self.s2[-1] if self.s2 else None
```

- Similar logic to `dequeue()`.
- If `s2` is non-empty, its top element (`s2[-1]`) is the front of the queue.
- If `s2` is empty, we move all elements from `s1` to `s2`, then the top of `s2` is the front.
- We **return** that front element without popping it (so it remains in the queue).

5. `isEmpty()`

```python
def isEmpty(self):
    return not self.s1 and not self.s2
```

- Returns **True** if both stacks are empty; otherwise, **False**.

How It Works Overall

- **Enqueue**: Always push to `s1`.
- **Dequeue** / **Peek**:
    i. If `s2` is not empty, pop/peek from `s2`.
    ii. Otherwise, move all elements from `s1` to `s2` (thus reversing the order). Now the top of `s2` is the front of the queue.
- This approach achieves amortized **O(1)** time for each enqueue or dequeue, because each element is transferred from `s1` to `s2` at most once.

Question : New Year Celebration

```
cold = []
ice = []
for i in range(int(input())):
    x = list(map(int, input().split()))
    if len(x) == 2:
        if x[0] == 1:
            ice.append(x[1])
        else:
            cold.append(x[1])
    else:
        if x[0] == 3:
            if len(ice) == 0:
                print(-1)
            else:
                print(ice[0])
        elif x[0] == 4:
            if len(cold) == 0:
                print(-1)
            else:
                print(cold[-1])
        else:
            if len(ice) == 0:
                print(-1)
            else:
                cold.append(ice.pop(0))
```

Explanation

1. **List Initialization**

   ```
   cold = []
   ice = []
   ```

   - We have two lists:
     - `ice` to represent a **queue** (FIFO structure) for ice-cream orders.
     - `cold` to represent a **stack** (LIFO structure) for cold drinks.

2. **Reading the Number of Queries**

```
for i in range(int(input())):
```

- The code first reads an integer which tells us how many queries (operations) we need to process.

3. **Reading Each Query**

```
x = list(map(int, input().split()))
```

- Each query is read as a list of integers.
- Depending on the length of `x`, we can have different types of operations:
  - If `len(x) == 2`, it means the query is either `1 X` or `2 X`.
  - Otherwise, if `len(x) == 1`, it means the query is one of `3`, `4`, or `5`.

4. **Handling `len(x) == 2`**

```
if len(x) == 2:
    if x[0] == 1:
        ice.append(x[1])
    else:
        cold.append(x[1])
```

- **Query `1 X`**: Add item `X` to the `ice` list (queue).
  - This simulates enqueueing `X` to the queue.
- **Query `2 X`**: Add item `X` to the `cold` list (stack).
  - This simulates pushing `X` onto the stack.

5. **Handling `len(x) == 1`**

- Here, `x[0]` can be `3`, `4`, or `5`.

a. Query `3`:

```
if x[0] == 3:
    if len(ice) == 0:
        print(-1)
    else:
        print(ice[0])
```

- If the queue `ice` is empty, print `-1`.
- Otherwise, print the front element of the queue (which is `ice[0]`).

b. Query `4`:

```
elif x[0] == 4:
    if len(cold) == 0:
        print(-1)
    else:
        print(cold[-1])
```

- If the stack `cold` is empty, print `-1`.
- Otherwise, print the top element of the stack (which is `cold[-1]`).

c. Query `5` (implicitly the `else` case):

```
else:
    if len(ice) == 0:
        print(-1)
    else:
        cold.append(ice.pop(0))
```

- If the queue `ice` is empty, print `-1`.
- Otherwise, remove the front item from the queue (`ice.pop(0)`) and **push** it onto the `cold` stack (`cold.append(...)`).

How It Works Overall

- We maintain two data structures:
  - **`ice`** (list used as a queue): Items are added to the end (`ice.append(...)`) and removed from the front (`ice.pop(0)`).
  - **`cold`** (list used as a stack): Items are added to the end (`cold.append(...)`) and accessed/removed from the end (`cold[-1]`).
- **Operations**:
  i. **`1 X`**: Enqueue `X` to `ice`.
  ii. **`2 X`**: Push `X` onto `cold`.
  iii. **`3`**: Print the front of `ice` (or `-1` if empty).
  iv. **`4`**: Print the top of `cold` (or `-1` if empty).
  v. **`5`**: Pop from the front of `ice` and push onto `cold` (or `-1` if `ice` is empty).

- This setup simulates two counters: one for ice-cream (`ice` queue) and one for cold drinks (`cold` stack). Queries allow adding to each counter, peeking at them, or transferring from the ice queue to the cold stack.

Question : Implement Stack using Queue

```
class Stack:
    def __init__(self):
        self.Q1 = []
        self.Q2 = []

    def push(self, value):
        # Enqueue the new value into Q2
        self.Q2.append(value)

        # Move all elements from Q1 to Q2
        while self.Q1:
            self.Q2.append(self.Q1.pop(0))

        # Swap the two queues so Q1 has all elements again,
        # with the newest at the front
        self.Q1, self.Q2 = self.Q2, self.Q1

    def pop(self):
        if self.isEmpty():
            print(-1)  # or handle empty case as needed
        else:
            # Dequeue from Q1 (the front is the "top" of our stack)
            popped_val = self.Q1.pop(0)
            print(popped_val)

    def top(self):
        if self.isEmpty():
            print(-1)  # or handle empty case as needed
        else:
            # The front of Q1 is the "top" of the stack
            print(self.Q1[0])

    def isEmpty(self):
        return len(self.Q1) == 0
```

Explanation

1. **Data Members**

    ```
    self.Q1 = []
    self.Q2 = []
    ```

    - We use **two queues** (implemented as lists) to simulate stack operations:
        - `Q1` will always hold all the elements after each push operation, with the top element at the **front**.
        - `Q2` is used temporarily to help reorder the elements when we push a new one.

2. **Push Operation**

    ```
    def push(self, value):
        self.Q2.append(value)

        while self.Q1:
            self.Q2.append(self.Q1.pop(0))

        self.Q1, self.Q2 = self.Q2, self.Q1
    ```

    - We **enqueue** the new `value` into `Q2`.
    - We then **transfer** all elements from `Q1` to `Q2`. This ensures the newly pushed element ends up at the **front** of `Q2` (since it was added first, and everything else is appended after it).
    - Finally, we **swap** `Q1` and `Q2`.
        - After the swap, `Q1` holds all the elements with the **newly pushed** element at index `0` (the front of `Q1`).
        - `Q2` becomes empty again.

    Result: In `Q1`, the front (index `0`) acts as the top of the stack.

3. **Pop Operation**

    ```
    def pop(self):
        if self.isEmpty():
            print(-1)
        else:
            popped_val = self.Q1.pop(0)
            print(popped_val)
    ```

    - If `Q1` is empty, we print `-1` to indicate an empty stack.
    - Otherwise, we **pop from the front** of `Q1` (index `0`), which represents the top of the stack.

- We print the popped value.

4. **Top Operation**

```
def top(self):
    if self.isEmpty():
        print(-1)
    else:
        print(self.Q1[0])
```

- If `Q1` is empty, we print `-1`.
- Otherwise, the element at the front of `Q1` (`Q1[0]`) is the top of the stack, so we print that value without removing it.

5. **Checking if the Stack is Empty**

```
def isEmpty(self):
    return len(self.Q1) == 0
```

- Returns **True** if there are no elements in `Q1`, meaning the stack is empty.

How It Works Overall

- **Push**:
    i. Insert the new item into `Q2`.
    ii. Move everything from `Q1` to `Q2`.
    iii. Swap `Q1` and `Q2`.
    - This places the newly pushed element at the front of `Q1`, effectively making it the top of the stack.

- **Pop**:
    - Remove and return the front element of `Q1`, which corresponds to the top of the stack.

- **Top**:
    - Return (or print) the front element of `Q1` without removing it.

- **isEmpty**:
    - Checks if `Q1` has no elements, which means the stack is empty.

By doing so, **each push** operation runs in O(n) time (due to moving all elements from `Q1` to `Q2`), and **each pop** or **top** operation runs in O(1) time. This satisfies the idea of implementing a stack using two queues, with the new element always placed at the "front" of the queue.

Question : People in Queue

```
K, Q = map(int, input().split())
q = []

while Q > 0:
    t = list(map(int, input().split()))
    if t[0] == 1:
        x = t[1]
        # Check if the queue is not full (has fewer than K elements)
        if len(q) < K:
            q.append(x)
            print(x)
        else:
            print("-1")
    else:
        # t[0] == 2
        # Dequeue operation
        if len(q) > 0:
            z = q[0]
            print(q[0])
            q.remove(z)
        else:
            print("-1")
    Q -= 1
```

Explanation

1. **Reading the Input**

   ```
   K, Q = map(int, input().split())
   ```

   - `K`: The **capacity** of the queue (the maximum number of people that can be in the queue at once).
   - `Q`: The **number of queries** or operations to process.

2. **Initializing the Queue**

```
q = []
```

   - We use a Python list `q` to represent the queue.
   - We will append to the **end** when someone enters the queue (enqueue) and remove from the **front** when someone leaves (dequeue).

3. **Processing Each Query**

```
while Q > 0:
    t = list(map(int, input().split()))
    ...
    Q -= 1
```

   - We loop `Q` times to read and execute each operation.
   - Each query is read as a list of integers `t`.
   - After processing each query, we decrement `Q` by 1.

4. **Enqueue Operation (t[0] == 1)**

```
if t[0] == 1:
    x = t[1]
    if len(q) < K:
        q.append(x)
        print(x)
    else:
        print("-1")
```

   - If `t[0]` equals `1`, it means "**a person with identity x wants to enter the queue.**"
     - We extract the identity `x` from `t[1]`.
     - We check if the queue is **not full** (`len(q) < K`).
       - If there is space, we append `x` to `q` and **print** the identity of the person who entered.
       - If the queue is already at capacity (`len(q) == K`), we print `"-1"` to indicate the person could not join.

5. **Dequeue Operation (t[0] == 2)**

```
else:
    if len(q) > 0:
        z = q[0]
```

```
        print(q[0])
        q.remove(z)
    else:
        print("-1")
```

- If `t[0]` equals `2`, it means **“the person at the front of the queue leaves.”**
  - If the queue is **not empty** (`len(q) > 0`):
    - We store `z = q[0]`, which is the identity at the front.
    - We **print** that identity.
    - We then remove it from the queue using `q.remove(z)` (alternatively, we could use `pop(0)`).
  - If the queue is empty, we print `"-1"` to indicate no one can leave.

How It Works Overall

- **Capacity**: The queue can hold at most `K` people at a time.
- **Queries**:
  i. **`1 x`**: Attempt to add person with identity `x` to the queue.
     - Print `x` if successful, otherwise print `-1` if the queue is full.
  ii. **`2`**: Attempt to remove the person at the front of the queue.
     - Print that person’s identity if the queue is not empty, otherwise print `-1` if the queue is empty.
- After reading and processing all `Q` queries, the program ends.