Version 2024.1

Assignment 3

Data Structures and Analysis

Chandra Gummaluru *University of Toronto*

Part 3.1: Implementing a Queue Class

In this problem, you will implement a **Queue** class using heaps. Below you can find a skeleton implementation.

```python

class Queue:
    """
    A priority-based queue implemented using a binary min-heap.

    The queue internally stores QueueNode objects in an array-based
    binary heap. Lower priority values correspond to objects that
    are removed earlier.
    """

    class QueueNode:
        """
        A container storing an object and its associated priority.

        Args:
        obj (object): The object to store in the queue.
        pri (int): The priority associated with the object. Lower values
            indicate higher priority.

        Returns:
        None
        """
        def __init__(self, obj: object, pri: int):
            self.obj = obj
            self.pri = pri

    """
    Initialize the Queue data structure.

    Args:
```

```
    cap (int): The initial capacity of the queue.

    Returns:
    None
    """
    def __init__(self, cap: int):
        self.cap = cap

    def add(self, obj: object, pri: int) -> None:
        """
        Add a new object to the queue with the given priority.

        A new QueueNode is created and appended to the internal heap
        array. The method then restores the min-heap property by
        performing a heap-up operation.

        Args:
        obj (object): The object to insert.
        pri (int): The priority associated with the object. Lower values
            indicate higher priority.
        """
        pass

    def pop(self) -> object:
        """
        Remove and return the object with the largest priority value.

        The root of the heap is removed. The last element in the heap is
        moved to the root position, and a heap-down operation is
        performed to restore the heap property.

        Args:
        None

        Returns:
        object: The object stored in the QueueNode with the largest
            priority value (or None if the Queue is empty).
        """
        pass
```

Complete the `Queue` class provided in `a3_submission.py` using an array-based max-heap. Do NOT import any additional libraries.

Part 3.2: Packet Scheduling with Retransmissions

A radio tower needs to transmit packets that it receives in an unreliable network. Each packet contains several pieces of metadata that allow the tower to track transmissions, detect loss, handle acknowledgments, and schedule retries. Specifically, every packet includes:

- `sender`: The ID or name of the device that originally generated the packet.
- `recipient`: The intended destination device for the packet.
- `packet_id`: A unique identifier that allows the system to track the packet and match it to any acknowledgment messages.
- `packet_type`: The category of the packet. Valid types include `text`, `audio`, `video`, `picture`, and `ack`.
- `send_time`: The time at which the packet was sent by the sender.
- `ack_time_tolerance`: The maximum amount of time the radio tower should wait for an acknowledgement before treating the packet as lost or requiring retransmission.

Below is a simple Python class representing a packet:

```python
class Packet:
    """
    Represents a data packet transmitted through a radio tower network.

    Args:
    sender (str): The device that created and sent the packet.
    recipient (str): The intended destination of the packet.
    packet_id (int): A unique identifier used to track this packet.
    packet_type (str): The type of packet.
    send_time (float): The time at which the packet was sent.
    ack_time_tolerance (float): Maximum allowed time to wait for an
        acknowledgement packet.

    Returns:
    None
    """
    def __init__(self, sender: str, recipient: str, packet_id: int,
                 packet_type: str, send_time: float, ack_time_tolerance:
                     float):
        self.sender = sender
        self.recipient = recipient
        self.packet_id = packet_id
        self.packet_type = packet_type
        self.send_time = send_time
        self.ack_time_tolerance = ack_time_tolerance
```

Design an efficient system that the tower can use to process packets according to the following rules:

- All received packets must be processed in the exact order in which they arrive.
- Packets that are meant for this tower can be read and then discarded (in this case, just discarded).
- Any acknowledgements received (i.e., packets whose `packet_type` is `ack`) should be recorded and associated with the corresponding packet they acknowledge. The tower must keep track of which packets have been acknowledged and which are still awaiting acknowledgement.
- Packets that are to be forwarding must be given a priority value before being placed into a forwarding queue. The priority is determined by the formula:

  $$\texttt{priority} = \texttt{ack\_time\_tolerance} + \texttt{processing\_time},$$

  where processing_time depends on `packet_type` as specified below. Lower values indicate higher priority.
- Each packet type requires a fixed processing-time cost:
  - `text`: 1 time-step
  - `picture`: 2 time-steps
  - `audio`: 3 time-steps
  - `video`: 4 time-steps
  - `ack`: 1 time-step
- The tower maintains the current global time-step, which increases by 1 after each cycle of operations.
- During each global time-step, the tower must perform the following actions in order:
  1. **Process Received Packets.** The tower processes received packets (up to 10).
  2. **Check for Expired Acknowledgements.** For every packet previously sent but still awaiting acknowledgement, the tower checks whether:

     $$\texttt{current\_time\_step} - \texttt{send\_time} > \texttt{ack\_time\_tolerance}.$$

     Any packets that exceed their tolerance must be marked for retransmission, even if the acknowledgement arrived on time but could not be processed in time.

3. **Send One Packet.** At the end of each time-step, the tower transmits exactly one packet from the forwarding structure, chosen according to priority (lower priority value is sent first).

```python
class Tower:
    def __init__(self):
        # Initialize any internal state here
        self.time = 0
        pass

    def process(self, new_packets):
        """
        Called once per time step.

        Parameters:
            new_packets: a list of packets that arrived at this time step

        Returns:
            read_packets: packets that were read/received this step
            sent_packets: packets that were sent out this step
            acked_packets: packets that were acknowledged this step
        """

        # Step 1: advance time
        self.time += 1

        # Step 2: record or process newly arrived packets
        read_packets = []

        # Step 3: logic determining which packets get acknowledged
        acked_packets = []

        # Step 4: logic determining which packets get sent this step
        sent_packets = []

        return read_packets, sent_packets, acked_packets
```

Complete the `process` function within the `Tower` class provided in `a3_submission.py`. You should make use of your `Queue` class from Part 3.1.

Tips

Here are a few tips to help you for this assignment:

- Do **NOT** change any of the pre-existing code within `a2_submission.py`. The auto-grader will assume that the starter code remains unchanged.
- When writing tests, it's often best to test functions in isolation. If one function depends on another, you don't want failures in the inner function to cause false failures in the outer function. In such cases, you can mock (i.e., replace temporarily) the inner function so you only test the logic of the outer function.

  ```python
  def bar(x: int) -> int:
      # Imagine this is a complicated function (e.g., database call)
      return x * 2

  def foo(x: int) -> int:
      # foo depends on bar
      return bar(x) + 1
  ```

  Normally, `foo(3)` calls `bar(3)` which returns `6`, so the result is `7`.

  But when testing `foo`, we may want to ignore the real logic of `bar` and mock it.

  ```python
  # Save the original bar function
  original_bar = bar

  # Define a fake version of bar for testing
  def mock_bar(x: int) -> int:
      print("mock_bar called with", x)
      return 100 # always return 100, no matter what

  # Replace bar with the mock
  bar = mock_bar

  # Now test foo in isolation
  print(foo(3)) # This will print "mock_bar called with 3" and return
      101

  # Restore the original bar after testing
  bar = original_bar

  print(foo(3)) # Back to normal: 7
  ```