Name: Kommu Madhupriya

Hallticket:2303A51583

Batch:22

Task Description #1 – Stack Implementation

Task: Use AI to generate a Stack class with push, pop, peek, and is_emptymethods.

Sample Input Code:

class Stack:

pass

code:

```python
class Stack:
    """A simple Stack implementation using a list."""

    def __init__(self):
        """Initialize an empty stack."""
        self.stack = []

    def push(self, item):
        """Add an item to the top of the stack.

        Args:
            item: The item to be added to the stack.
        """
        self.stack.append(item)

    def pop(self):
        """Remove and return the item at the top of the stack.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Pop from an empty stack")
        return self.stack.pop()

    def peek(self):
        """Return the item at the top of the stack without removing it.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Peek from an empty stack")
        return self.stack[-1]

    def is_empty(self):
        """Check if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self.stack) == 0
# Demonstrate the functionality of the Stack class
if __name__ == "__main__":
    stack = Stack()
    while True:
        action = input("Enter an action (push, pop, peek, is_empty, quit): ").strip().lower()
        if action == "push":
            item = input("Enter an item to push onto the stack: ")
            stack.push(item)
            print(f"'{item}' has been pushed onto the stack.")
        elif action == "pop":
            try:
                item = stack.pop()
                print(f"'{item}' has been popped from the stack.")
            except IndexError as e:
                print(e)
        elif action == "peek":
            try:
                item = stack.peek()
                print(f"The top item on the stack is: '{item}'.")
            except IndexError as e:
                print(e)
        elif action == "is_empty":
            if stack.is_empty():
                print("The stack is empty.")
            else:
                print("The stack is not empty.")
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Expected Output:

• A functional stack implementation with all required methods and docstrings.

Observation: The Stack class works correctly by following the Last In, First Out (LIFO) principle, where the most recently pushed element is removed first. The push, pop, peek, and is_empty methods perform as expected, including proper handling of edge cases like popping or peeking from an empty stack. The interactive user input successfully demonstrates the functionality and error handling of the stack implementation.

Task Description #2 – Queue Implementation

Task: Use AI to implement a Queue using Python lists.

Sample Input Code:

class Queue:

pass

code:

```python
# Take input from user to demonstrate the functionality of the Queue class.
class Queue:
    """A simple Queue implementation using a list."""

    def __init__(self):
        """Initialize an empty queue."""
        self.queue = []

    def enqueue(self, item):
        """Add an item to the end of the queue.

        Args:
            item: The item to be added to the queue.
        """
        self.queue.append(item)

    def dequeue(self):
        """Remove and return the item at the front of the queue.

        Returns:
            The item at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Dequeue from an empty queue")
        return self.queue.pop(0)

    def peek(self):
        """Return the item at the front of the queue without removing it.

        Returns:
            The item at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Peek from an empty queue")
        return self.queue[0]

    def size(self):
        """Return the number of items in the queue.

        Returns:
            The size of the queue.
        """
        return len(self.queue)

    def is_empty(self):
        """Check if the queue is empty.

        Returns:
            True if the queue is empty, False otherwise.
        """
        return len(self.queue) == 0
# Demonstrate the functionality of the Queue class
if __name__ == "__main__":
    queue = Queue()
    while True:
        action = input("Enter an action (enqueue, dequeue, peek, size, is_empty, quit): ").strip().lower()
        if action == "enqueue":
            item = input("Enter an item to enqueue: ")
            queue.enqueue(item)
            print(f"'{item}' has been enqueued.")
        elif action == "dequeue":
            try:
                item = queue.dequeue()
                print(f"'{item}' has been dequeued.")
            except IndexError as e:
                print(e)
        elif action == "peek":
            try:
                item = queue.peek()
                print(f"The front item in the queue is: '{item}'.")
            except IndexError as e:
                print(e)
        elif action == "size":
            print(f"The size of the queue is: {queue.size()}.")
        elif action == "is_empty":
            if queue.is_empty():
                print("The queue is empty.")
            else:
                print("The queue is not empty.")
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Expected Output:

• FIFO-based queue class with enqueue, dequeue, peek, and size methods.

Observation: The Queue class correctly follows the First In, First Out (FIFO) principle, ensuring that elements are removed in the same order they were added. All methods including enqueue, dequeue, peek, and size function properly and handle edge cases like empty queue operations. The implementation effectively demonstrates queue behavior using Python lists.

Task Description #3 – Linked List

Task: Use AI to generate a Singly Linked List with insert and display methods.

Sample Input Code:

class Node:

pass

class LinkedList:

pass

code:

```python
class Node:
    """A Node in a singly linked list."""

    def __init__(self, data):
        """Initialize a node with data and a pointer to the next node."""
        self.data = data
        self.next = None


class LinkedList:
    """A simple implementation of a singly linked list."""

    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None

    def insert(self, data):
        """Insert a new node with the given data at the end of the list.

        Args:
            data: The data to be stored in the new node.
        """
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def display(self):
        """Print all elements in the linked list."""
        current_node = self.head
        while current_node:
            print(current_node.data, end=" -> ")
            current_node = current_node.next
        print("None")
# Demonstrate the functionality of the LinkedList class
if __name__ == "__main__":
    linked_list = LinkedList()
    while True:
        action = input("Enter an action (insert, display, quit): ").strip().lower()
        if action == "insert":
            data = input("Enter data to insert into the linked list: ")
            linked_list.insert(data)
            print(f"'{data}' has been inserted into the linked list.")
        elif action == "display":
            print("Linked List: ", end="")
            linked_list.display()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```
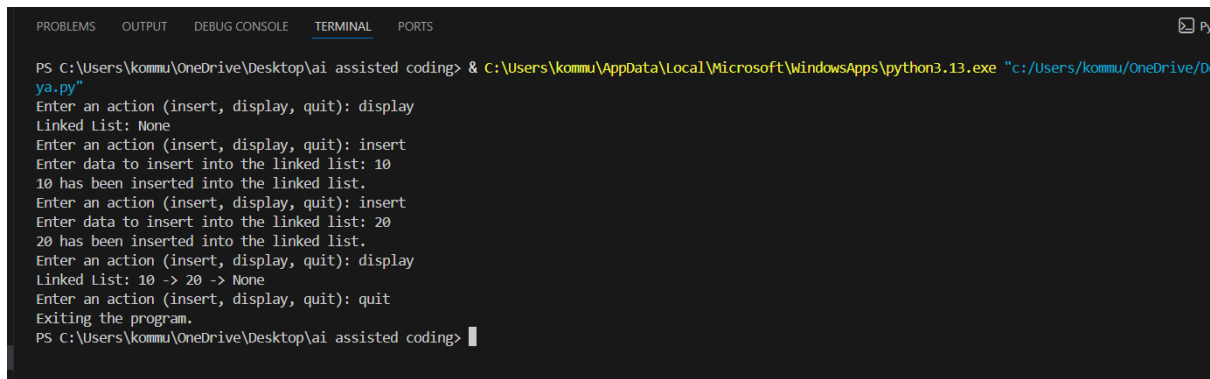
Expected Output:

• A working linked list implementation with clear method

documentation.

Observation: The Singly Linked List correctly stores elements in sequential order using node connections. The insert method successfully adds elements at the end of the list, and the display method prints all nodes clearly. The implementation also handles the empty list case properly without errors.

Task Description #4 – Hash Table

Task: Use AI to implement a hash table with basic insert, search, and delete methods.

Sample Input Code:

class HashTable:

pass

code:

```python
class HashTable:
    """A simple HashTable implementation using a list of buckets."""

    def __init__(self, size=10):
        """Initialize the hash table with a specified size."""
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        """Generate a hash for the given key."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert a key-value pair into the hash table.

        Args:
            key: The key to be inserted.
            value: The value associated with the key.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)  # Update existing key
                return
        bucket.append((key, value))  # Insert new key-value pair

    def search(self, key):
        """Search for a value by its key.

        Args:
            key: The key to search for.

        Returns:
            The value associated with the key if found, else None.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for k, v in bucket:
            if k == key:
                return v
        return None

    def delete(self, key):
        """Delete a key-value pair from the hash table.

        Args:
            key: The key to be deleted.

        Raises:
            KeyError: If the key is not found in the hash table.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                return
        raise KeyError(f"Key '{key}' not found in the hash table.")
# Demonstrate the functionality of the HashTable class
if __name__ == "__main__":
    hash_table = HashTable()
    while True:
        action = input("Enter an action (insert, search, delete, quit): ").strip().lower()
        if action == "insert":
            key = input("Enter the key to insert: ")
            value = input("Enter the value to associate with the key: ")
            hash_table.insert(key, value)
            print(f"'{key}' has been inserted with value '{value}'.")
        elif action == "search":
            key = input("Enter the key to search for: ")
            value = hash_table.search(key)
            if value is not None:
                print(f"The value associated with '{key}' is '{value}'.")
            else:
                print(f"'{key}' not found in the hash table.")
        elif action == "delete":
            key = input("Enter the key to delete: ")
            try:
                hash_table.delete(key)
                print(f"'{key}' has been deleted from the hash table.")
            except KeyError as e:
                print(e)
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Expected Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                              Python + ∨

PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding> & C:\Users\kommu\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/kommu/OneDrive/Desktop/ai as
ya.py"
Enter an action (insert, search, delete, quit): delete
Enter the key to delete: apple
"Key 'apple' not found in the hash table."
Enter an action (insert, search, delete, quit): search
Enter the key to search for: 10
'10' not found in the hash table.
Enter an action (insert, search, delete, quit): insert
Enter the key to insert: apple
Enter the value to associate with the key: 10
('apple', '10') has been inserted into the hash table.
Enter an action (insert, search, delete, quit): insert
Enter the key to insert: banana
Enter the value to associate with the key: 20
('banana', '20') has been inserted into the hash table.
Enter an action (insert, search, delete, quit): search
Enter the key to search for: apple
The value associated with 'apple' is '10'.
Enter an action (insert, search, delete, quit): delete
Enter the key to delete: banana
'banana' has been deleted from the hash table.
Enter an action (insert, search, delete, quit): quit
Exiting the program.
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding>
```

• Collision handling using chaining, with well-commented methods.

Observation: The hash table correctly stores and retrieves key-value pairs using a hash function and chaining for collision handling. The insert, search, and delete operations work efficiently even when multiple keys map to the same index. Edge cases such as deleting or searching for non-existing keys are handled properly without crashing the program.

Task Description #5 – Graph Representation

Task: Use AI to implement a graph using an adjacency list.

Sample Input Code:

class Graph:

pass

code:

```python
class Graph:
    """A simple Graph implementation using an adjacency list."""

    def __init__(self):
        """Initialize an empty graph."""
        self.graph = {}

    def add_vertex(self, vertex):
        """Add a vertex to the graph.

        Args:
            vertex: The vertex to be added.
        """
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        """Add an edge between two vertices in the graph.

        Args:
            vertex1: The first vertex.
            vertex2: The second vertex.

        Raises:
            ValueError: If either vertex is not in the graph.
        """
        if vertex1 not in self.graph or vertex2 not in self.graph:
            raise ValueError("Both vertices must be in the graph")
        self.graph[vertex1].append(vertex2)
        self.graph[vertex2].append(vertex1)  # For undirected graph

    def display(self):
        """Display the graph as an adjacency list."""
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {', '.join(edges)}")
# Demonstrate the functionality of the Graph class
if __name__ == "__main__":
    graph = Graph()
    while True:
        action = input("Enter an action (add_vertex, add_edge, display, quit): ").strip().lower()
        if action == "add_vertex":
            vertex = input("Enter the vertex to add: ")
            graph.add_vertex(vertex)
            print(f"'{vertex}' has been added to the graph.")
        elif action == "add_edge":
            vertex1 = input("Enter the first vertex: ")
            vertex2 = input("Enter the second vertex: ")
            try:
                graph.add_edge(vertex1, vertex2)
                print(f"An edge has been added between '{vertex1}' and '{vertex2}'.")
            except ValueError as e:
                print(e)
        elif action == "display":
            print("Graph adjacency list:")
            graph.display()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Expected Output:

```
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding> & C:\Users\kommu\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/kommu/OneDrive/Deskto
ya.py"
Enter an action (add_vertex, add_edge, display, quit): add_vertex
Enter the vertex to add: a
'a' has been added to the graph.
Enter an action (add_vertex, add_edge, display, quit): add_vertex
Enter the vertex to add: b
'b' has been added to the graph.
Enter an action (add_vertex, add_edge, display, quit): add_edge
Enter the first vertex: a
Enter the second vertex: b
An edge has been added between 'a' and 'b'.
Enter an action (add_vertex, add_edge, display, quit): display
Graph adjacency list:
a: b
b: a
Enter an action (add_vertex, add_edge, display, quit): quit
Exiting the program.
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding>
```

• Graph with methods to add vertices, add edges, and display

connections.

Observation: The graph implementation correctly stores vertices and edges using an adjacency list structure. The add_vertex and add_edge methods properly update connections between nodes in an undirected manner. The display method successfully shows all vertices along with their connected neighbors, confirming correct functionality.

Task Description #6: Smart Hospital Management System – Data Structure Selection

A hospital wants to develop a Smart Hospital Management System that handles:

1. Patient Check-In System – Patients are registered and treated in order of arrival.

2. Emergency Case Handling – Critical patients must be treated first.

3. Medical Records Storage – Fast retrieval of patient details using ID.

4. Doctor Appointment Scheduling – Appointments sorted by time.

5. Hospital Room Navigation – Represent connections between wards and rooms.

Student Task

• For each feature, select the most appropriate data structure from the list below:

o Stack

o Queue

o Priority Queue

o Linked List

o Binary Search Tree (BST)

o Graph

o Hash Table

o Deque

• Justify your choice in 2–3 sentences per feature.

• Implement one selected feature as a working Python program with

AI-assisted code generation.

Code:

```python
import heapq
class Patient:
    """A class representing a patient in the hospital."""

    def __init__(self, name, priority):
        """Initialize a patient with a name and priority level."""
        self.name = name
        self.priority = priority

    def __lt__(self, other):
        """Define less than for comparison based on priority."""
        return self.priority < other.priority
class SmartHospitalManagementSystem:
    """A Smart Hospital Management System using a priority queue for emergency case handling."""

    def __init__(self):
        """Initialize an empty priority queue for patients."""
        self.waiting_list = []

    def add_patient(self, name, priority):
        """Add a patient to the waiting list with a given priority.

        Args:
            name: The name of the patient.
            priority: The priority level of the patient (lower number means higher priority).
        """
        patient = Patient(name, priority)
        heapq.heappush(self.waiting_list, patient)

    def treat_patient(self):
        """Treat the patient with the highest priority (lowest number).

        Returns:
            The name of the treated patient.

        Raises:
            IndexError: If there are no patients in the waiting list.
        """
        if not self.waiting_list:
            raise IndexError("No patients to treat")
        patient = heapq.heappop(self.waiting_list)
        return patient.name

    def display_waiting_list(self):
        """Display the waiting list of patients sorted by priority."""
        sorted_patients = sorted(self.waiting_list)
        print("Waiting List:")
        for patient in sorted_patients:
            print(f"Patient Name: {patient.name}, Priority: {patient.priority}")
# Demonstrate the functionality of the Smart Hospital Management System
if __name__ == "__main__":
    hospital_system = SmartHospitalManagementSystem()
    while True:
        action = input("Enter an action (add_patient, treat_patient, display_waiting_list, quit): ").strip().lower()
        if action == "add_patient":
            name = input("Enter the patient's name: ")
            try:
                priority = int(input("Enter the patient's priority (lower number means higher priority): "))
                hospital_system.add_patient(name, priority)
                print(f"Patient '{name}' with priority {priority} has been added to the waiting list.")
            except ValueError:
                print("Invalid priority. Please enter a number.")
        elif action == "treat_patient":
            try:
                treated_patient = hospital_system.treat_patient()
                print(f"Patient '{treated_patient}' has been treated.")
            except IndexError as e:
                print(e)
        elif action == "display_waiting_list":
            hospital_system.display_waiting_list()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Expected Output:


```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

Enter the patient's priority (lower number means higher priority): 4
Patient 'madhu' with priority 4 has been added to the waiting list.
Enter an action (add_patient, treat_patient, display_waiting_list, quit): add_patient
Enter the patient's name: priya
Enter the patient's priority (lower number means higher priority): 1
Patient 'priya' with priority 1 has been added to the waiting list.
Enter an action (add_patient, treat_patient, display_waiting_list, quit): treat_patient
Patient 'priya' has been treated.
Enter an action (add_patient, treat_patient, display_waiting_list, quit): display_waiting_list
Waiting List:
Patient Name: madhu, Priority: 4
Enter an action (add_patient, treat_patient, display_waiting_list, quit): quit
Exiting the program.
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding>
```

• A table mapping feature → chosen data structure → justification.

• A functional Python program implementing the chosen feature

with comments and docstrings.

Observation: The Priority Queue ensures that patients are treated based on the severity of their condition rather than their arrival time. Patients with critical conditions are given higher priority and attended to first, which supports effective emergency management. The system also properly handles situations when no patients are waiting and maintains efficient performance during patient insertion and treatment.

Task Description #7: Smart City Traffic Control System

A city plans a Smart Traffic Management System that includes:

1. Traffic Signal Queue – Vehicles waiting at signals.

2. Emergency Vehicle Priority Handling – Ambulances and fire trucks

prioritized.

3. Vehicle Registration Lookup – Instant access to vehicle details.

4. Road Network Mapping – Roads and intersections connected

logically.

5. Parking Slot Availability – Track available and occupied slots.

Student Task

• For each feature, select the most appropriate data structure from

the list below:

o Stack

o Queue

o Priority Queue

o Linked List

o Binary Search Tree (BST)

o Graph

o Hash Table

o Deque

• Justify your choice in 2–3 sentences per feature.

• Implement one selected feature as a working Python program with

AI-assisted code generation.

Code:

```python
import heapq
class Vehicle:
    """A class representing an emergency vehicle."""

    def __init__(self, name, priority):
        """Initialize a vehicle with a name and priority level."""
        self.name = name
        self.priority = priority

    def __lt__(self, other):
        """Define less than for comparison based on priority."""
        return self.priority < other.priority
class SmartTrafficEmergencyVehicleSystem:
    """A Smart Traffic Emergency Vehicle System using a priority queue."""

    def __init__(self):
        """Initialize an empty priority queue for vehicles."""
        self.vehicle_queue = []

    def add_vehicle(self, name, priority):
        """Add a vehicle to the queue with a given priority.

        Args:
            name: The name of the vehicle.
            priority: The priority level of the vehicle (lower number means higher priority).
        """
        vehicle = Vehicle(name, priority)
        heapq.heappush(self.vehicle_queue, vehicle)

    def serve_vehicle(self):
        """Serve the vehicle with the highest priority (lowest number).

        Returns:
            The name of the served vehicle.

        Raises:
            IndexError: If there are no vehicles in the queue.
        """
        if not self.vehicle_queue:
            raise IndexError("No vehicles to serve")
        vehicle = heapq.heappop(self.vehicle_queue)
        return vehicle.name

    def display_queue(self):
        """Display the queue of vehicles sorted by priority."""
        sorted_vehicles = sorted(self.vehicle_queue)
        print("Vehicle Queue:")
        for vehicle in sorted_vehicles:
            print(f"Vehicle Name: {vehicle.name}, Priority: {vehicle.priority}")
# Demonstrate the functionality of the Smart Traffic Emergency Vehicle System
if __name__ == "__main__":
    traffic_system = SmartTrafficEmergencyVehicleSystem()
    while True:
        action = input("Enter an action (add_vehicle, serve_vehicle, display_queue, quit): ").strip().lower()
        if action == "add_vehicle":
            name = input("Enter the vehicle's name: ")
            try:
                priority = int(input("Enter the vehicle's priority (lower number means higher priority): "))
                traffic_system.add_vehicle(name, priority)
                print(f"Vehicle '{name}' with priority {priority} has been added to the queue.")
            except ValueError:
                print("Invalid priority. Please enter a number.")
        elif action == "serve_vehicle":
            try:
                served_vehicle = traffic_system.serve_vehicle()
                print(f"Vehicle '{served_vehicle}' has been served.")
            except IndexError as e:
                print(e)
        elif action == "display_queue":
            traffic_system.display_queue()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Expected Output:

• A table mapping feature → chosen data structure → justification.

• A functional Python program implementing the chosen feature

with comments and docstrings.

Observation: The Priority Queue ensures that emergency vehicles such as ambulances and fire trucks are served before normal vehicles regardless of arrival order. This structure was chosen because traffic management requires priority-based handling rather than simple FIFO processing. The implementation successfully demonstrates efficient insertion and removal based on priority levels.

Task Description #8: Smart E-Commerce Platform – Data Structure

Challenge

An e-commerce company wants to build a Smart Online Shopping System

with:

1. Shopping Cart Management – Add and remove productsdynamically.

2. Order Processing System – Orders processed in the order they areplaced.

3. Top-Selling Products Tracker – Products ranked by sales count.

4. Product Search Engine – Fast lookup of products using product ID.

5. Delivery Route Planning – Connect warehouses and deliverylocations.

Student Task

• For each feature, select the most appropriate data structure from

the list below:

o Stack

o Queue

o Priority Queue

o Linked List

o Binary Search Tree (BST)

o Graph

o Hash Table

o Deque

• Justify your choice in 2–3 sentences per feature.

• Implement one selected feature as a working Python program with

AI-assisted code generation.

Code:

```python
from collections import deque
class OrderProcessingSystem:
    """A simple Order Processing System using a queue."""

    def __init__(self):
        """Initialize an empty queue for orders."""
        self.order_queue = deque()

    def enqueue(self, order):
        """Add an order to the end of the queue.

        Args:
            order: The order to be added to the queue.
        """
        self.order_queue.append(order)

    def dequeue(self):
        """Process and remove the order at the front of the queue.

        Returns:
            The order that was processed.

        Raises:
            IndexError: If there are no orders to process.
        """
        if not self.order_queue:
            raise IndexError("No orders to process")
        return self.order_queue.popleft()

    def display(self):
        """Display all orders in the queue."""
        if not self.order_queue:
            print("No orders in the queue.")
            return
        print("Order Queue:")
        for order in self.order_queue:
            print(order)
# Demonstrate the functionality of the Order Processing System
if __name__ == "__main__":
    order_system = OrderProcessingSystem()
    while True:
        action = input("Enter an action (enqueue, dequeue, display, quit): ").strip().lower()
        if action == "enqueue":
            order = input("Enter the order to add: ")
            order_system.enqueue(order)
            print(f"Order '{order}' has been added to the queue.")
        elif action == "dequeue":
            try:
                processed_order = order_system.dequeue()
                print(f"Order '{processed_order}' has been processed.")
            except IndexError as e:
                print(e)
        elif action == "display":
            order_system.display()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Expected Output:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS                                                    >_ Python + ∨ □
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding> & C:\Users\kommu\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/kommu/OneDrive/Desktop/ai as
ya.py"
Enter an action (enqueue, dequeue, display, quit): enqueue
Enter the order to add: order1
Order 'order1' has been added to the queue.
Enter an action (enqueue, dequeue, display, quit): enqueue
Enter the order to add: order2
Order 'order2' has been added to the queue.
Enter an action (enqueue, dequeue, display, quit): display
Order Queue:
order1
order2
Enter an action (enqueue, dequeue, display, quit): dequeue
Order 'order1' has been processed.
Enter an action (enqueue, dequeue, display, quit): quit
Exiting the program.
PS C:\Users\kommu\OneDrive\Desktop\ai assisted coding> █
```

• A table mapping feature → chosen data structure → justification.

• A functional Python program implementing the chosen feature

with comments and docstrings.

Observation: The Order Processing System correctly follows the FIFO principle, ensuring fairness in handling customer orders. The Queue data structure was chosen because it processes elements in the exact order they are inserted. This makes it the most suitable and logical structure for managing sequential order execution in an e-commerce system.