AIAC Assignment-11.5

Name: M.Harshith
Ht.no: 2303A51444
Batch. No: 21

Task Description #1 – Stack Implementation

Prompt:

Create a Python Stack class using a list. Include methods: push, pop, peek and is_empty. Add proper docstrings and it should handle edge cases and errors. Take input from user to demonstrate the functionality of the Stack class.

Code:

```
# Create a Python Stack class using a list. I
# nclude methods: push, pop, peek and is_empty. Add proper docstrings and it should handle edge cases
# and errors.
# Take input from user to demonstrate the functionality of the Stack class.
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."""
        self.stack.append(item)

    def pop(self):
        """Remove and return the top item from the stack."""
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.stack.pop()

    def peek(self):
        """Return the top item from the stack without removing it."""
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.stack[-1]

    def is_empty(self):
        """Return True if the stack is empty, otherwise False."""
        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":
```

```python
# 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 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.")
```

Output:

```
Enter an action (push, pop, peek, is_empty, quit): push
Enter an item to push onto the stack: a
'a' has been pushed onto the stack.
Enter an action (push, pop, peek, is_empty, quit): push
Enter an item to push onto the stack: b
'b' has been pushed onto the stack.
Enter an action (push, pop, peek, is_empty, quit): pop
'b' has been popped from the stack.
Enter an action (push, pop, peek, is_empty, quit): peek
The top item is: 'a'.
Enter an action (push, pop, peek, is_empty, quit): is_empty
The stack is not empty.
Enter an action (push, pop, peek, is_empty, quit): quit
Exiting the program.
```

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

Prompt:

Create a Python Queue class using a list. Implement enqueue, dequeue, peek, and size methods. Follow FIFO principle. Add proper docstrings and handle empty queue errors. Take input from user to demonstrate the functionality of the Queue class.

Code:

```
assignment11.py > ...

# Create a Python Queue class using a list.
# Implement enqueue, dequeue, peek, and size methods.
# Follow FIFO principle. Add proper docstrings and
# handle empty queue errors. Take input from asyncio
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."""
        self.queue.append(item)

    def dequeue(self):
        """Remove and return the front item from the queue."""
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.queue.pop(0)

    def peek(self):
        """Return the front item from the queue without removing it."""
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.queue[0]

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

    def is_empty(self):
        """Return True if the queue is empty, otherwise False."""
        return len(self.queue) == 0
# Demonstrate the functionality of the Queue class
if __name__ == "__main__":
```

04.py assignmnet5.5.py assignment08.py 17-02-2026.py ass9.2.py assignment11.py

ignment11.py > ...

```python
            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 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.")
```

Output:

```
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): enqueue
Enter an item to enqueue: 2
'2' has been enqueued.
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): dequeue
'2' has been dequeued.
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): 2
Invalid action. Please try again.
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): peek
peek from empty queue
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): size
The size of the queue is: 0.
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): is_empty
The queue is empty.
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): quit
Exiting the program.
```

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

Prompt:

Create a Python implementation of a Singly Linked List. Define a Node class and a LinkedList class.

Include methods: insert(data) to add at the end and display() to print all elements. Add proper docstrings and basic error handling. Take input from user to demonstrate the functionality of the LinkedList class.

Code:

```
 Create a Python implementation of a Singly Linked List.
 Define a Node class and a LinkedList class.
 Include methods: insert(data) to add at the end and display() to print all elements.
 Add proper docstrings and basic error handling.
 Take input from user to demonstrate the functionality of the LinkedList class.

lass 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
lass 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."""
       new_node = Node(data)
       if not self.head:
           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."""
       if not self.head:
           print("The linked list is empty.")
           return
```

```python
        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("The elements in the linked list are:")
            linked_list.display()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Output:

```
Enter an action (insert, display, quit): display
The elements in the linked list are:
10 -> None
Enter an action (insert, display, quit): quit
Exiting the program.
```

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

Prompt:

Create a Python HashTable class. Implement insert, search, and delete methods. Add proper docstrings and basic error handling. Take input from user to demonstrate the functionality of the HashTable class.

Code:

assignment11.py > ...

```python
# Create a Python HashTable class. Implement insert, search, and delete methods.
# Add proper docstrings and basic error handling.
# Take input from user to demonstrate the functionality of the HashTable class.
# Code:
class HashTable:
    """A simple implementation of a hash table using chaining for collision resolution."""

    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."""
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        self.table[index].append((key, value))  # Insert new key-value pair

    def search(self, key):
        """Search for a value by its key in the hash table."""
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v  # Return the value if the key is found
        raise KeyError(f"Key '{key}' not found in the hash table.")

    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
```

```python
assignment11.py > ...
class HashTable:
    def delete(self, key):
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]  # Remove the key-value pair
                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 insert: ")
            hash_table.insert(key, value)
            print(f"'{key}': '{value}' has been inserted into the hash table.")
        elif action == "search":
            key = input("Enter the key to search for: ")
            try:
                value = hash_table.search(key)
                print(f"The value for key '{key}' is: '{value}'.")
            except KeyError as e:
                print(e)
        elif action == "delete":
            key = input("Enter the key to delete: ")
            try:
                hash_table.delete(key)
                print(f"Key '{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.")
```

Output:

```
Enter an action (insert, display, quit): insert
Enter data to insert into the linked list: 10
'10' has been inserted into the linked list.
Enter an action (insert, display, quit): display
The elements in the linked list are:
10 -> None
Enter an action (insert, display, quit): qiut
Invalid action. Please try again.
Enter an action (insert, display, quit):
```

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

Prompt:

Create a Graph class using an adjacency list (dictionary).

Include methods: add_vertex, add_edge, and display. Add proper docstrings and basic error handling. Take input from user to demonstrate the functionality of the Graph class.

Code:

assignment11.py > ...

```python

# Create a Graph class using an adjacency list (dictionary).
# Include methods: add_vertex, add_edge, and display.
# Add proper docstrings and basic error handling.
# Take input from user to demonstrate the functionality of the Graph class.

class Graph:
    """A simple implementation of a graph using an adjacency list."""

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

    def add_vertex(self, vertex):
        """Add a vertex to the graph."""
        if vertex in self.graph:
            print(f"Vertex '{vertex}' already exists.")
        else:
            self.graph[vertex] = []
            print(f"Vertex '{vertex}' has been added.")

    def add_edge(self, vertex1, vertex2):
        """Add an edge between two vertices in the graph."""
        if vertex1 not in self.graph or vertex2 not in self.graph:
            print("Both vertices must exist in the graph to add an edge.")
            return
        self.graph[vertex1].append(vertex2)
        self.graph[vertex2].append(vertex1)  # For undirected graph
        print(f"Edge between '{vertex1}' and '{vertex2}' has been added.")

    def display(self):
        """Display the adjacency list of the graph."""
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {', '.join(edges)}")
# Demonstrate the functionality of the Graph class
if __name__ == "__main__":
    graph = Graph()
```

```python
    def display(self):
        """Display the adjacency list of the graph."""
        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)
        elif action == "add_edge":
            vertex1 = input("Enter the first vertex: ")
            vertex2 = input("Enter the second vertex: ")
            graph.add_edge(vertex1, vertex2)
        elif action == "display":
            print("The adjacency list of the graph is:")
            graph.display()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Output:

```
ter an action (insert, display, quit): insert
ter data to insert into the linked list: 20
0' has been inserted into the linked list.
ter an action (insert, display, quit): display
e elements in the linked list are:
 -> 20 -> None
ter an action (insert, display, quit): qiut
valid action. Please try again.
ter an action (insert, display, quit): insert
ter data to insert into the linked list: 30
0' has been inserted into the linked list.
ter an action (insert, display, quit): display
e elements in the linked list are:
 -> 20 -> 30 -> None
                                                            Ln 316, Col 55    Spaces: 4
```

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 Prompt:

Create a Python program for Smart Hospital Management System. Implement Emergency Case Handling using a Priority Queue. Patients with higher priority (critical level) should be treated first. Include functions to add patient, treat patient, and display waiting list. Add proper docstrings, and basic error handling. Take input from user to demonstrate the functionality of the Smart Hospital Management System.

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}")
```

```python
# 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.")
```

Output:

```
Enter an action (add_patient, treat_patient, display_waiting_list, quit): add_patient
Enter the patient's name: John
Enter the patient's priority (lower number means higher priority): 2
Patient 'John' with priority 2 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: Alice
Enter the patient's priority (lower number means higher priority): 5
Patient 'Alice' with priority 5 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: Bob
Enter the patient's priority (lower number means higher priority): 3
Patient 'Bob' with priority 3 has been added to the waiting list.
Enter an action (add_patient, treat_patient, display_waiting_list, quit): treat_patient
Patient 'John' has been treated.
Enter an action (add_patient, treat_patient, display_waiting_list, quit): display_waiting_list
Waiting List:
Patient Name: Bob, Priority: 3
Patient Name: Alice, Priority: 5
Enter an action (add_patient, treat_patient, display_waiting_list, quit): quit
Exiting the program.
```

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 Prompt:

Create a Smart Traffic Emergency Vehicle System using Priority Queue. Vehicles have name and priority (1 = highest). Implement add_vehicle(), serve_vehicle(), and display_queue(). Include docstrings and basic error handling. Take input from user to demonstrate the functionality of the Smart Traffic Emergency Vehicle System.

Code:

```python
# Create a Smart Traffic Emergency Vehicle System using Priority Queue.
# Vehicles have name and priority (1 = highest). Implement add_vehicle(), serve_vehicle(),
# and display_queue(). Include docstrings and basic error handling. T
# ake input from user to demonstrate the functionality of the Smart Traffic Emergency Vehicle System.
import heapq
class EmergencyVehicle:
    """A class representing an emergency vehicle with a name and priority."""

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

    def __lt__(self, other):
        """Define less than for comparison based on priority."""
        return self.priority < other.priority
class SmartTrafficSystem:
    """A smart traffic system that manages emergency vehicles using a priority queue."""

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

    def add_vehicle(self, name, priority):
        """Add an emergency vehicle to the priority queue."""
        vehicle = EmergencyVehicle(name, priority)
        heapq.heappush(self.queue, vehicle)
        print(f"Emergency vehicle '{name}' with priority {priority} has been added to the queue.")

    def serve_vehicle(self):
        """Serve the highest priority emergency vehicle from the queue."""
        if not self.queue:
            print("No emergency vehicles in the queue to serve.")
            return
```

```python
assignment11.py > ...
class SmartTrafficSystem:
    def serve_vehicle(self):
            print("No emergency vehicles in the queue to serve.")
            return
        vehicle = heapq.heappop(self.queue)
        print(f"Serving emergency vehicle '{vehicle.name}' with priority {vehicle.priority}.")

    def display_queue(self):
        """Display the current emergency vehicles in the queue."""
        if not self.queue:
            print("The emergency vehicle queue is empty.")
            return
        print("Current emergency vehicles in the queue:")
        for vehicle in sorted(self.queue):
            print(f"'{vehicle.name}' with priority {vehicle.priority}")
# Demonstrate the functionality of the Smart Traffic Emergency Vehicle System
if __name__ == "__main__":
    traffic_system = SmartTrafficSystem()
    while True:
        action = input("Enter an action (add_vehicle, serve_vehicle, display_queue, quit): ").strip().lowe
        if action == "add_vehicle":
            name = input("Enter the name of the emergency vehicle: ")
            try:
                priority = int(input("Enter the priority of the emergency vehicle (1 = highest): "))
                traffic_system.add_vehicle(name, priority)
            except ValueError:
                print("Priority must be an integer. Please try again.")
        elif action == "serve_vehicle":
            traffic_system.serve_vehicle()
        elif action == "display_queue":
            traffic_system.display_queue()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Output:

```
Enter an action (add_vehicle, serve_vehicle, display_queue, quit): add_vehicle
Enter the name of the emergency vehicle: honda
Enter the priority of the emergency vehicle (1 = highest): 5
Emergency vehicle 'honda' with priority 5 has been added to the queue.
Enter an action (add_vehicle, serve_vehicle, display_queue, quit): serve_vehicle
Serving emergency vehicle 'honda' with priority 5.
Enter an action (add_vehicle, serve_vehicle, display_queue, quit): clean
Invalid action. Please try again.
Enter an action (add_vehicle, serve_vehicle, display_queue, quit): quit
Exiting the program.
```

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

Prompt:

Create a Python program implementing an Order Processing System using a Queue. Include enqueue (add order), dequeue (process order), and display methods. Add proper docstrings, and basic error handling. Take input from user to demonstrate the functionality of the Order Processing System.

Code:

```
assignment11.py > ...
#            print("Invalid action. Please try again.")

# Create a Python program implementing an Order Processing System using a Queue.
# Include enqueue (add order), dequeue (process order), and display methods.
# Add proper docstrings, and basic error handling. Take input from user to demonstrate the
# functionality of the Order Processing System.
# Code:
from collections import deque
class OrderProcessingSystem:
    """A simple order processing system using a queue."""

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

    def enqueue(self, order):
        """Add an order to the processing queue."""
        self.orders.append(order)
        print(f"Order '{order}' has been added to the processing queue.")

    def dequeue(self):
        """Process the next order in the queue."""
        if not self.orders:
            print("No orders in the queue to process.")
            return
        order = self.orders.popleft()
        print(f"Processing order '{order}'.")

    def display(self):
        """Display the current orders in the processing queue."""
        if not self.orders:
            print("The order processing queue is empty.")
            return
        print("Current orders in the processing queue:")
        for order in self.orders:
            print(f"'{order}'")
# Demonstrate the functionality of the Order Processing System
```

```python
assignment11.py > ...
class OrderProcessingSystem:
    def display(self):
            print("The order processing queue is empty.")
            return
        print("Current orders in the processing queue:")
        for order in self.orders:
            print(f"'{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 to the processing queue: ")
            order_system.enqueue(order)
        elif action == "dequeue":
            order_system.dequeue()
        elif action == "display":
            order_system.display()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Output:

```
Enter an action (enqueue, dequeue, display, quit): enqueue
Enter the order to add to the processing queue: 10
Order '10' has been added to the processing queue.
Enter an action (enqueue, dequeue, display, quit): dequeue
Processing order '10'.
Enter an action (enqueue, dequeue, display, quit): diasply
Invalid action. Please try again.
Enter an action (enqueue, dequeue, display, quit): display
The order processing queue is empty.
Enter an action (enqueue, dequeue, display, quit): quit
Exiting the program.
```

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.