Assignment-11.5

Name : Akash

Ht.No : 2303A51422

Batch : 21

Task Description #1 – Stack Implementation

Prompt:

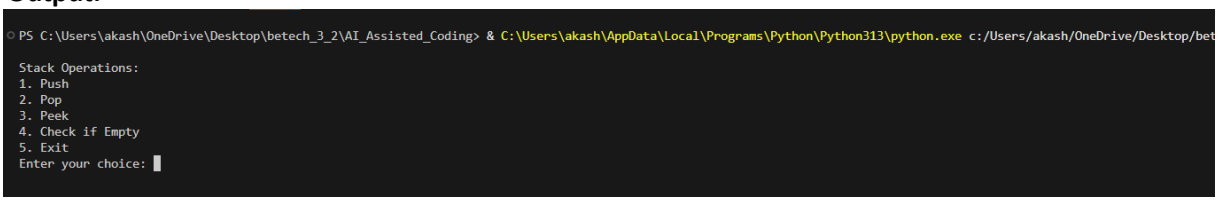
Create a Python Stack class using a list. Include methods: push, pop, peek and is_empty. Add proper docstrings and it should handle edge cases and errors. Take input from user to demonstrate the functionality of the Stack class.

Code:

```
'''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.'''
# use this prompt and give the code shortly
class Stack:
    """A simple Stack class implemented 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 item at the top of the stack. Raises an error 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. Raises an error if the stack is empty."""
        if self.is_empty():
            raise IndexError("Peek from an empty stack")
        return self.stack[-1]

    def is_empty(self):
        """Return True if the stack is empty, False otherwise."""
        return len(self.stack) == 0
def main():
    stack = Stack()
    while True:
        print("\nStack Operations:")
        print("1. Push")
        print("2. Pop")
        print("3. Peek")
        print("4. Check if Empty")
        print("5. Exit")
        choice = input("Enter your choice: ")

```

```
        if choice == '1':
            item = input("Enter the item to push: ")
            stack.push(item)
            print(f"'{item}' has been pushed to the stack.")
        elif choice == '2':
            try:
                popped_item = stack.pop()
                print(f"'{popped_item}' has been popped from the stack.")
            except IndexError as e:
                print(e)
        elif choice == '3':
            try:
                top_item = stack.peek()
                print(f"The top item is: '{top_item}'")
            except IndexError as e:
                print(e)
        elif choice == '4':
            if stack.is_empty():
                print("The stack is empty.")
            else:
                print("The stack is not empty.")
        elif choice == '5':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please try again.")
if __name__ == "__main__":
    main()
```

Output:


PS C:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding> & C:\Users\akash\AppData\Local\Programs\Python\Python313\python.exe c:/Users/akash/OneDrive/Desktop/bet
Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice:

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:

Lab_08.py > ...

```python
'''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.'''
class Node:
    """A class representing a node in a singly linked list."""
    def __init__(self, data):
        """
        Initializes a new node with the given data and sets the next pointer to None.

        :param data: The data to be stored in the node.
        """
        self.data = data
        self.next = None
class LinkedList:
    """A class representing a singly linked list."""
    def __init__(self):
        """Initializes an empty linked list."""
        self.head = None

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

        :param 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):
        """Prints all elements in the linked list."""
        current_node = self.head
        while current_node:
            print(current_node.data, end=' ')
            current_node = current_node.next
        print()  # for a new line after printing all elements
if __name__ == "__main__":
    linked_list = LinkedList()
    while True:
        user_input = input("Enter a value to insert into the linked list (or 'exit' to stop): ")
        if user_input.lower() == 'exit':
            break
        linked_list.insert(user_input)
    print("Elements in the linked list:")
    linked_list.display()
```

Output:

```
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding> & C:\Users\akash\AppData\Local\Programs\Pyt
hon\Python313\python.exe c:/Users/akash/OneDrive/Desktop/betech_3_2/AI_Assisted_Coding/Lab_08.py
Enter a value to insert into the linked list (or 'exit' to stop): 4
Enter a value to insert into the linked list (or 'exit' to stop): 1
Enter a value to insert into the linked list (or 'exit' to stop):
```

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:

```python
'''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.'''
class HashTable:
    def __init__(self, size=10):
        """Initialize the hash table with a specified size."""
        self.size = size
        self.table = [None] * self.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)
        if self.table[index] is None:
            self.table[index] = [(key, value)]
        else:
            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))  # Add new key-value pair

    def search(self, key):
        """Search for a value by its key in the hash table."""
        index = self._hash(key)
        if self.table[index] is not None:
            for k, v in self.table[index]:
                if k == key:
                    return v
        return None  # Key not found

    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        index = self._hash(key)
        if self.table[index] is not None:
            for i, (k, v) in enumerate(self.table[index]):
                if k == key:
                    del self.table[index][i]  # Remove the key-value pair
                    return True
        return False  # Key not found
```

```python
        return False  # Key not found
# Demonstration of HashTable functionality
if __name__ == "__main__":
    hash_table = HashTable()

    while True:
        print("\nHash Table Operations:")
        print("1. Insert")
        print("2. Search")
        print("3. Delete")
        print("4. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            key = input("Enter key to insert: ")
            value = input("Enter value to insert: ")
            hash_table.insert(key, value)
            print(f"Inserted ({key}, {value}) into the hash table.")

        elif choice == '2':
            key = input("Enter key to search: ")
            value = hash_table.search(key)
            if value is not None:
                print(f"Value found for key '{key}': {value}")
            else:
                print(f"Key '{key}' not found in the hash table.")

        elif choice == '3':
            key = input("Enter key to delete: ")
            if hash_table.delete(key):
                print(f"Key '{key}' deleted from the hash table.")
            else:
                print(f"Key '{key}' not found in the hash table.")

        elif choice == '4':
            print("Exiting...")
            break

        else:
            print("Invalid choice. Please try again.")

```

Output:

```
Enter your choice: C:\Users\akash\AppData\Local\Programs\Python\Python313\python.exe c:/Users/akash/OneDrive/
Desktop/betech_3_2/AI_Assisted_Coding/Lab_08.py
Invalid choice. Please try again.

Hash Table Operations:
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 2
Enter key to search: 3
Key '3' not found in the hash table.

Hash Table Operations:
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice:
```

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:

```python
'''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'''
class Graph:
    def __init__(self):
        """Initialize an empty graph using an adjacency list."""
        self.graph = {}

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

        Args:
            vertex: The vertex to be added.

        Raises:
            ValueError: If the vertex already exists in the graph.
        """
        if vertex in self.graph:
            raise ValueError(f"Vertex '{vertex}' already exists.")
        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 does not exist in the graph.
        """
        if vertex1 not in self.graph or vertex2 not in self.graph:
            raise ValueError("Both vertices must exist in the graph.")
        self.graph[vertex1].append(vertex2)
        self.graph[vertex2].append(vertex1)

    def display(self):
        """Display the graph as an adjacency list."""
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {edges}")
def main():
    graph = Graph()
```

```
    while True:
        print("\nMenu:")
        print("1. Add Vertex")
        print("2. Add Edge")
        print("3. Display Graph")
        print("4. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':
            vertex = input("Enter vertex to add: ")
            try:
                graph.add_vertex(vertex)
                print(f"Vertex '{vertex}' added successfully.")
            except ValueError as e:
                print(e)

        elif choice == '2':
            vertex1 = input("Enter the first vertex: ")
            vertex2 = input("Enter the second vertex: ")
            try:
                graph.add_edge(vertex1, vertex2)
                print(f"Edge between '{vertex1}' and '{vertex2}' added successfully.")
            except ValueError as e:
                print(e)

        elif choice == '3':
            print("Graph representation (adjacency list):")
            graph.display()

        elif choice == '4':
            print("Exiting the program.")
            break

        else:
            print("Invalid choice. Please try again.")
if __name__ == "__main__":
    main()

```

Output:

```
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding> & C:\Users\akash\AppData\Local\Programs\Python\Python313\python.exe c:/Users/akash/OneDrive/Desktop/betech_3_2/AI_Assisted_Coding/Lab_08.py

Menu:
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice: 1
Enter vertex to add: 2
Vertex '2' added successfully.

Menu:
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice:
```

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:

```python
'''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.
'''
import heapq
class Patient:
    def __init__(self, name, priority):
        self.name = name
        self.priority = priority

    def __lt__(self, other):
        return self.priority < other.priority
class SmartHospitalManagementSystem:
    def __init__(self):
        self.waiting_list = []

    def add_patient(self, name, priority):
        """Add a patient to the waiting list with a given name and priority."""
        patient = Patient(name, priority)
        heapq.heappush(self.waiting_list, patient)

    def treat_patient(self):
        """Treat the patient with the highest priority (lowest priority number)."""
        if not self.waiting_list:
            print("No patients in the waiting list.")
            return None
        return heapq.heappop(self.waiting_list)

    def display_waiting_list(self):
        """Display the current waiting list of patients."""
        if not self.waiting_list:
            print("No patients in the waiting list.")
            return
        print("Waiting List:")
        for patient in sorted(self.waiting_list):
            print(f"Patient Name: {patient.name}, Priority: {patient.priority}")
```

```python
def main():
    hospital = SmartHospitalManagementSystem()
    while True:
        print("\nSmart Hospital Management System")
        print("1. Add Patient")
        print("2. Treat Patient")
        print("3. Display Waiting List")
        print("4. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            name = input("Enter patient name: ")
            try:
                priority = int(input("Enter patient priority (lower number means higher priority): "))
                hospital.add_patient(name, priority)
                print(f"Patient {name} added with priority {priority}.")
            except ValueError:
                print("Invalid input for priority. Please enter an integer.")
        elif choice == '2':
            treated_patient = hospital.treat_patient()
            if treated_patient:
                print(f"Treated patient: {treated_patient.name} with priority {treated_patient.priority}.")
        elif choice == '3':
            hospital.display_waiting_list()
        elif choice == '4':
            print("Exiting the system.")
            break
        else:
            print("Invalid choice. Please try again.")
if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding> & C:\Users\akash\AppData\Local\Programs\Python\Python
313\python.exe c:/Users/akash/OneDrive/Desktop/betech_3_2/AI_Assisted_Coding/Lab_08.py

Smart Hospital Management System
1. Add Patient
2. Treat Patient
3. Display Waiting List
4. Exit
Enter your choice: 1
Enter patient name: tejuuu
Enter patient priority (lower number means higher priority): 2
Patient tejuuu added with priority 2.

Smart Hospital Management System
1. Add Patient
2. Treat Patient
3. Display Waiting List
4. Exit
Enter your choice:
```

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:

```
'''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.
'''
import heapq
class SmartTrafficEmergencyVehicleSystem:
    def __init__(self):
        """Initialize an empty priority queue for vehicles."""
        self.vehicle_queue = []

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

        Parameters:
        name (str): The name of the vehicle.
        priority (int): The priority of the vehicle (1 = highest).

        Raises:
        ValueError: If priority is not a positive integer.
        """
        if priority < 1:
            raise ValueError("Priority must be a positive integer.")
        heapq.heappush(self.vehicle_queue, (priority, name))

    def serve_vehicle(self):
        """
        Serve the highest priority vehicle from the queue.

        Returns:
        str: The name of the served vehicle.

        Raises:
        IndexError: If there are no vehicles to serve.
        """
        if not self.vehicle_queue:
            raise IndexError("No vehicles to serve.")
        return heapq.heappop(self.vehicle_queue)[1]

```

```
    def display_queue(self):
        """Display the current queue of vehicles."""
        if not self.vehicle_queue:
            print("The queue is empty.")
            return
        print("Current Vehicle Queue:")
        for priority, name in sorted(self.vehicle_queue):
            print(f"Vehicle: {name}, Priority: {priority}")
def main():
    system = SmartTrafficEmergencyVehicleSystem()
    while True:
        print("\n1. Add Vehicle")
        print("2. Serve Vehicle")
        print("3. Display Queue")
        print("4. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            name = input("Enter vehicle name: ")
            try:
                priority = int(input("Enter vehicle priority (1 = highest): "))
                system.add_vehicle(name, priority)
                print(f"Vehicle '{name}' added with priority {priority}.")
            except ValueError as e:
                print(e)
        elif choice == '2':
            try:
                served_vehicle = system.serve_vehicle()
                print(f"Served vehicle: {served_vehicle}")
            except IndexError as e:
                print(e)
        elif choice == '3':
            system.display_queue()
        elif choice == '4':
            print("Exiting the system.")
            break
        else:
            print("Invalid choice. Please try again.")
if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding> C:\Users\akash\AppData\Local\Programs\Python\Python313\python.exe c:/Users/akash/OneDrive/Desktop/betech_3_2/AI_Assisted_Coding/Lab_08.py

1. Add Vehicle
2. Serve Vehicle
3. Display Queue
4. Exit
Enter your choice: 1
Enter vehicle name: akshaya
Enter vehicle priority (1 = highest): 2
Vehicle 'akshaya' added with priority 2.

1. Add Vehicle
2. Serve Vehicle
3. Display Queue
4. Exit
Enter your choice:
```

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
'''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.'''
class OrderProcessingSystem:
    """A class to represent an order processing system using a queue data structure."""

    def __init__(self):
        """Initialize the order processing system with an empty queue."""
        self.orders = []

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

        Args:
            order (str): The order to be added to the queue.
        """
        self.orders.append(order)
        print(f"Order '{order}' has been added to the queue.")

    def dequeue(self):
        """Process the next order in the queue.

        Returns:
            str: The order that was processed, or a message if the queue is empty.
        """
        if not self.orders:
            return "No orders to process."
        processed_order = self.orders.pop(0)
        return f"Order '{processed_order}' has been processed."

    def display(self):
        """Display all orders currently in the queue."""
        if not self.orders:
            print("No orders in the queue.")
        else:
            print("Current orders in the queue:")
            for index, order in enumerate(self.orders, start=1):
                print(f"{index}. {order}")
def main():
    """Main function to demonstrate the functionality of the Order Processing System."""
    system = OrderProcessingSystem()

```

```
    while True:
        print("\nOrder Processing System")
        print("1. Add Order")
        print("2. Process Order")
        print("3. Display Orders")
        print("4. Exit")

        choice = input("Enter your choice (1-4): ")

        if choice == '1':
            order = input("Enter the order to add: ")
            system.enqueue(order)
        elif choice == '2':
            result = system.dequeue()
            print(result)
        elif choice == '3':
            system.display()
        elif choice == '4':
            print("Exiting the Order Processing System.")
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 4.")
if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding> & C:\Users\akash\AppData\Local\Programs\Pyt
hon\Python313\python.exe c:/Users/akash/OneDrive/Desktop/betech_3_2/AI_Assisted_Coding/Lab_08.py
1. Add Order
2. Process Order
3. Display Orders
3. Display Orders
4. Exit
Enter your choice (1-4): 1
Enter the order to add: 2
Order '2' has been added to the queue.

Order Processing System
1. Add Order
2. Process Order
3. Display Orders

Order Processing System
1. Add Order
2. Process Order
3. Display Orders
4. Exit
```

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:

```python
'''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.'''
class OrderProcessingSystem:
    """A class to represent an order processing system using a queue data structure."""

    def __init__(self):
        """Initialize the order processing system with an empty queue."""
        self.orders = []

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

        Args:
            order (str): The order to be added to the queue.
        """
        self.orders.append(order)
        print(f"Order '{order}' has been added to the queue.")

    def dequeue(self):
        """Process the next order in the queue.

        Returns:
            str: The order that was processed, or a message if the queue is empty.
        """
        if not self.orders:
            return "No orders to process."
        processed_order = self.orders.pop(0)
        return f"Order '{processed_order}' has been processed."

    def display(self):
        """Display all orders currently in the queue."""
        if not self.orders:
            print("No orders in the queue.")
        else:
            print("Current orders in the queue:")
            for index, order in enumerate(self.orders, start=1):
                print(f"{index}. {order}")
def main():
    """Main function to demonstrate the functionality of the Order Processing System."""
    system = OrderProcessingSystem()

```

```python
    while True:
        print("\nOrder Processing System")
        print("1. Add Order")
        print("2. Process Order")
        print("3. Display Orders")
        print("4. Exit")

        choice = input("Enter your choice (1-4): ")

        if choice == '1':
            order = input("Enter the order to add: ")
            system.enqueue(order)
        elif choice == '2':
            result = system.dequeue()
            print(result)
        elif choice == '3':
            system.display()
        elif choice == '4':
            print("Exiting the Order Processing System.")
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 4.")
if __name__ == "__main__":
    main()

```

Output:

```
PS C:\Users\akash\OneDrive\Desktop\betech_3_2\AI_Assisted_Coding> & C:\Users\akash\AppData\Local\Programs\Python\Python313\python.exe c:/Users/akash/OneDrive/Desktop/betech_3_2/AI_Assisted_Coding/Lab_08.py
1. Add Order
2. Process Order
3. Display Orders
3. Display Orders
4. Exit
Enter your choice (1-4): 1
Enter the order to add: 2
Order '2' has been added to the queue.

Order Processing System
1. Add Order
2. Process Order
3. Display Orders

Order Processing System
1. Add Order
2. Process Order
3. Display Orders
4. Exit
```

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.