AI Assisted Coding
Assignment-11.5

Name: Mohammed Rahil Raza
Ht.No: 2303A51433
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:

```python
class MyStack:
    """
    Stack implementation using list.
    Works on LIFO (Last In First Out).
    """

    def __init__(self):
        self.data = []

    def add(self, value):
        # Push element to top
        self.data.insert(len(self.data), value)

    def remove(self):
        # Pop element from top
        if not self.isEmpty():
            top_value = self.data[-1]
            del self.data[-1]
            return top_value
        else:
            raise Exception("Stack Underflow")

    def top(self):
        # Peek top element
        if not self.isEmpty():
            return self.data[-1]
        else:
            raise Exception("Stack is empty")

    def isEmpty(self):
        return len(self.data) == 0


# Demo
s = MyStack()
s.add(100)
s.add(200)

print("Top element:", s.top())       # 200
print("Removed:", s.remove())        # 200
print("Is Empty?", s.isEmpty())      # False
print("Removed:", s.remove())        # 100
print("Is Empty?", s.isEmpty())      # True
```

Output:

```
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python
Top of stack: 2
Popped item: 2
Is stack empty? False
Popped item: 1
Is stack empty? True
```

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:

```python
class MyQueue:
    """
    Implementation of a Queue using Python list.
    Follows FIFO (First In First Out).
    """

    def __init__(self):
        self.data = []

    def add(self, value):
        # Insert element at rear
        self.data.insert(len(self.data), value)

    def remove(self):
        # Remove element from front
        if len(self.data) == 0:
            raise Exception("Queue is empty")
        front_value = self.data[0]
        del self.data[0]
        return front_value

    def front(self):
        # View front element
        if len(self.data) == 0:
            raise Exception("Queue is empty")
        return self.data[0]

    def length(self):
        return len(self.data)

    def empty(self):
        return len(self.data) == 0


# Demo
q = MyQueue()
q.add(10)
q.add(20)

print("Front:", q.front())        # 10
print("Removed:", q.remove())     # 10
print("Size:", q.length())        # 1
print("Removed:", q.remove())     # 20
print("Is Empty?", q.empty())          # True
```

Output:

```
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python
Front of queue: 1
Dequeued item: 1
Queue size: 1
Front of queue: 1
Dequeued item: 1
Queue size: 1
Dequeued item: 2
Is queue empty? True
```

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:

asdasdsa.py > ...

```python
class ListNode:
    def __init__(self, value):
        self.value = value
        self.link = None


class MyLinkedList:
    def __init__(self):
        self.start = None

    def add_element(self, value):
        new_node = ListNode(value)

        # If list is empty
        if self.start is None:
            self.start = new_node
            return

        # Traverse to last node
        temp = self.start
        while temp.link is not None:
            temp = temp.link

        temp.link = new_node

    def show(self):
        temp = self.start
        elements = []

        while temp:
            elements.append(str(temp.value))
            temp = temp.link

        print(" -> ".join(elements))


# Demo
ll = MyLinkedList()
ll.add_element(5)
ll.add_element(10)
ll.add_element(15)

print("Elements in list:")
ll.show()   # 5 -> 10 -> 15
```

Output:

```
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python
Linked List contents:
1 2 3
```

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
class MyHashMap:
    """
    Simple Hash Map using separate chaining.
    Stores key-value pairs inside buckets (lists).
    """

    def __init__(self, capacity=10):
        self.capacity = capacity
        self.buckets = [[] for _ in range(capacity)]

    def _get_index(self, key):
        # Generate index using built-in hash
        return abs(hash(key)) % self.capacity

    def put(self, key, value):
        index = self._get_index(key)
        bucket = self.buckets[index]

        # Update if key already exists
        for i in range(len(bucket)):
            k, v = bucket[i]
            if k == key:
                bucket[i] = (key, value)
                return

        # Otherwise insert new key-value pair
        bucket.append((key, value))

    def get(self, key):
        index = self._get_index(key)
        bucket = self.buckets[index]

        for k, v in bucket:
            if k == key:
                return v
        return None

    def remove(self, key):
        index = self._get_index(key)
        bucket = self.buckets[index]

        for i in range(len(bucket)):
            if bucket[i][0] == key:
                del bucket[i]
                return True
        return False


# Demo
h = MyHashMap()

h.put("name", "Rahil")
h.put("age", 21)

print("Name:", h.get("name"))
print("Age:", h.get("age"))

h.remove("age")
print("Age after delete:", h.get("age"))
```

```python
# Example usage
hash_table = HashTable()
hash_table.insert("name", "Alice")
hash_table.insert("age", 30)
print("Name:", hash_table.search("name"))  # Output: Alice
print("Age:", hash_table.search("age"))  # Output: 30
hash_table.delete("name")
print("Name after deletion:", hash_table.search("name"))  # Output: None
```

Output:

```
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python
Name: Alice
Age: 30
Name after deletion: None
```

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
class Graph:
    """
    A Graph is a data structure that consists of a set of vertices and a set of edges connecting those vertices.
    This implementation uses an adjacency list to represent the graph, allowing for efficient storage and traversal.
    """
    def __init__(self):
        self.graph = {}
    def add_vertex(self, vertex):
        """Add a vertex to the graph."""
        if vertex not in self.graph:
            self.graph[vertex] = []
    def add_edge(self, vertex1, vertex2):
        """Add an edge between two vertices in the graph."""
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertex1)  # For undirected graph
    def display(self):
        """Display the connections in the graph."""
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {', '.join(edges)}")
# Example usage
graph = Graph()
graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")
graph.add_edge("A", "B")
graph.add_edge("A", "C")
print("Graph connections:")
graph.display()  # Output: A: B, C; B: A; C:
```

Output:

```
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python
Graph connections:
A: B, C
B: A
C: A
```

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:

```
import heapq
class SmartHospitalManagementSystem:
    """
    A class to handle emergency cases in a hospital using a priority queue.
    Critical patients are treated first based on their severity level.
    """
    def __init__(self):
        self.emergency_queue = []
    def add_patient(self, patient_name, severity):
        """Add a patient to the emergency queue with their severity level."""
        heapq.heappush(self.emergency_queue, (-severity, patient_name))  # Negate severity for max-heap behavior
    def treat_patient(self):
        """Treat the most critical patient in the queue."""
        if not self.emergency_queue:
            print("No patients in the emergency queue.")
            return None
        severity, patient_name = heapq.heappop(self.emergency_queue)
        print(f"Treating patient: {patient_name} with severity level: {-severity}")
        return patient_name
    def display_waiting_patients(self):
        """Display the patients currently waiting in the emergency queue."""
        print("Patients in the emergency queue:")
        for severity, patient_name in sorted(self.emergency_queue, reverse=True):
            print(f"{patient_name} (Severity: {-severity})")

# demonstrate the functionality of the SmartHospitalManagementSystem class with error handling
if __name__ == "__main__":
    handler = SmartHospitalManagementSystem()
    handler.add_patient("Alice", 5)
    handler.add_patient("Bob", 8)
    handler.add_patient("Charlie", 3)
    handler.display_waiting_patients()
    handler.treat_patient()
    handler.treat_patient()
    handler.treat_patient()
    handler.treat_patient()  # Attempt to treat when queue is empty
```

Output:

```
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python
Patients in the emergency queue:
Charlie (Severity: 3)
Alice (Severity: 5)
Bob (Severity: 8)
Treating patient: Bob with severity level: 8
Treating patient: Alice with severity level: 5
Treating patient: Charlie with severity level: 3
No patients in the emergency queue.
```

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:

```
import heapq
class SmartTrafficManagementSystem:
    """
    A class to manage traffic in a smart city using a priority queue for emergency vehicles.
    Emergency vehicles are given priority over regular traffic based on their urgency level.
    """
    def __init__(self):
        self.traffic_queue = []
    def add_vehicle(self, vehicle_type, urgency):
        """Add a vehicle to the traffic queue with its urgency level."""
        heapq.heappush(self.traffic_queue, (-urgency, vehicle_type))  # Negate urgency for max-heap behavior
    def manage_traffic(self):
        """Manage traffic by allowing the most urgent vehicle to pass first."""
        if not self.traffic_queue:
            print("No vehicles in the traffic queue.")
            return None
        urgency, vehicle_type = heapq.heappop(self.traffic_queue)
        print(f"Allowing {vehicle_type} to pass with urgency level: {-urgency}")
        return vehicle_type
    def display_waiting_vehicles(self):
        """Display the vehicles currently waiting in the traffic queue."""
        print("Vehicles in the traffic queue:")
        for urgency, vehicle_type in sorted(self.traffic_queue, reverse=True):
            print(f"{vehicle_type} (Urgency: {-urgency})")

# demonstrate the functionality of the SmartTrafficManagementSystem class with error handling
if __name__ == "__main__":
    traffic_manager = SmartTrafficManagementSystem()
    traffic_manager.add_vehicle("Car", 1)
    traffic_manager.add_vehicle("Ambulance", 5)
    traffic_manager.add_vehicle("Fire Truck", 4)
    traffic_manager.display_waiting_vehicles()
    traffic_manager.manage_traffic()
    traffic_manager.manage_traffic()
    traffic_manager.manage_traffic()
    traffic_manager.manage_traffic()  # Attempt to manage when queue is empty
```

Output:

```
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python
Vehicles in the traffic queue:
Car (Urgency: 1)
Fire Truck (Urgency: 4)
Ambulance (Urgency: 5)
Allowing Ambulance to pass with urgency level: 5
Allowing Fire Truck to pass with urgency level: 4
Allowing Car to pass with urgency level: 1
No vehicles in the traffic queue.
```

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:

```
from collections import deque
class orderProcessingSystem:
    """
    A class to manage order processing in an e-commerce platform using a queue.
    Orders are processed in the order they are placed (FIFO).
    """
    def __init__(self):
        self.order_queue = deque()
    def place_order(self, order_id):
        """Place a new order into the processing queue."""
        self.order_queue.append(order_id)
        print(f"Order {order_id} placed.")
    def process_order(self):
        """Process the next order in the queue."""
        if not self.order_queue:
            print("No orders to process.")
            return None
        order_id = self.order_queue.popleft()
        print(f"Processing order {order_id}.")
        return order_id
    def display_pending_orders(self):
        """Display all pending orders in the queue."""
        print("Pending orders:")
        for order_id in self.order_queue:
            print(order_id)
# demonstrate the functionality of the orderProcessingSystem class with error handling
if __name__ == "__main__":
    order_system = orderProcessingSystem()
    order_system.place_order("Order001")
    order_system.place_order("Order002")
    order_system.place_order("Order003")
    order_system.display_pending_orders()
    order_system.process_order()
    order_system.process_order()
    order_system.process_order()
    order_system.process_order()  # Attempt to process when queue is empty
```

Output:

```
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python
Order Order001 placed.
Order Order002 placed.
Order Order003 placed.
Pending orders:
Order001
Order002
Order003
Processing order Order001.
Processing order Order002.
Processing order Order003.
No orders to process.
```

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 execution.