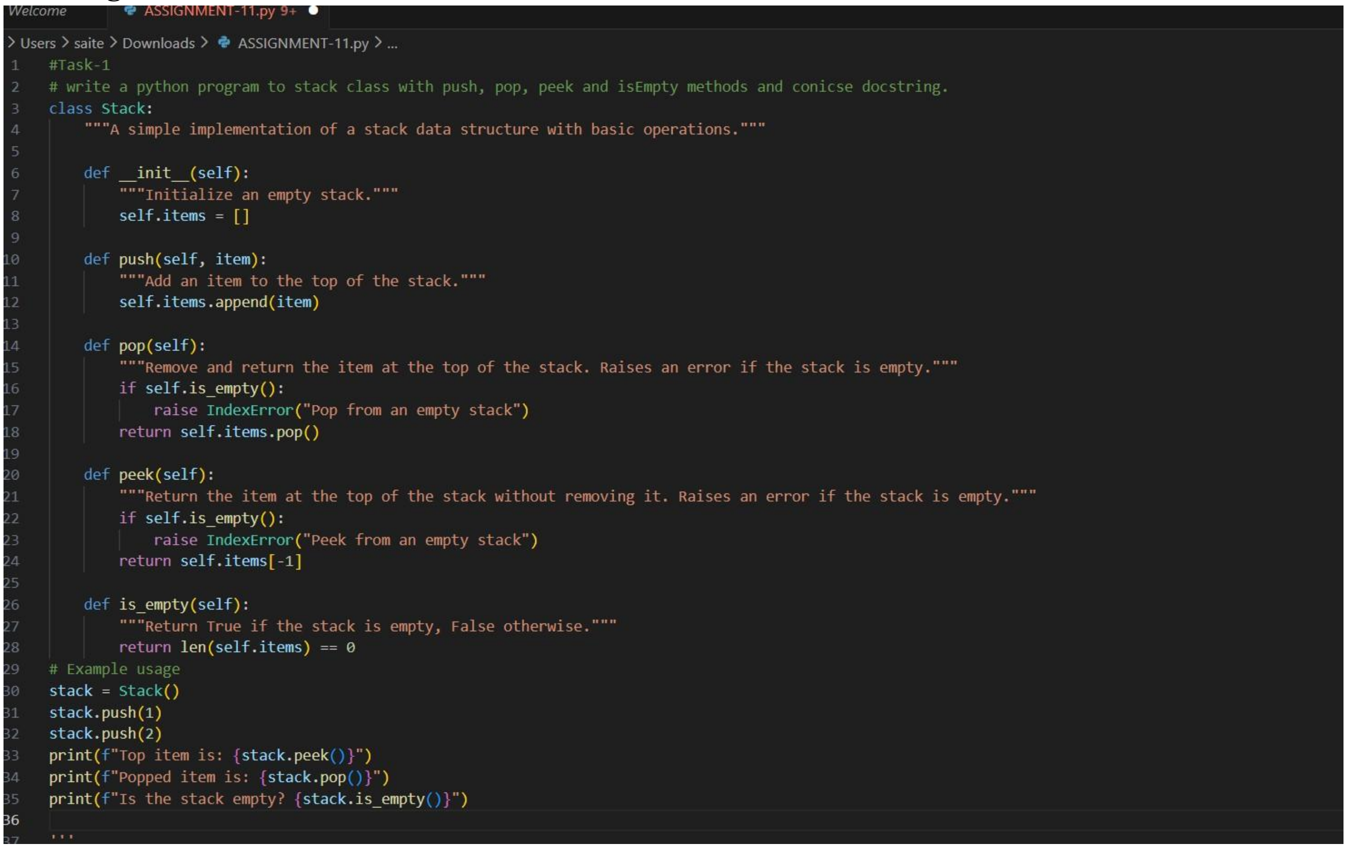
SR UNIVERSITY

School of Computer Science and Artificial Intelligence

Lab Assignment # 11

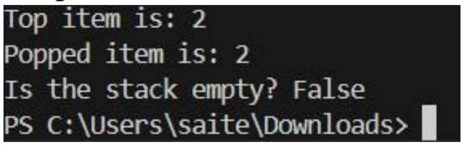
Name of Student : R SHIVA SAI REDDY
Enrollment No. : 2303A51542
Batch No. : 22

Task Description #1 – Stack Implementation
Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.
Sample Input Code: class
Stack:
pass
Expected Output:
• A functional stack implementation with all required methods and docstrings. Code:



```python
#Task-1
# write a python program to stack class with push, pop, peek and isEmpty methods and conicse docstring.
class Stack:
    """A simple implementation of a stack data structure with basic operations."""

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

    def push(self, item):
        """Add an item to the top of the stack."""
        self.items.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.items.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.items[-1]

    def is_empty(self):
        """Return True if the stack is empty, False otherwise."""
        return len(self.items) == 0
# Example usage
stack = Stack()
stack.push(1)
stack.push(2)
print(f"Top item is: {stack.peek()}")
print(f"Popped item is: {stack.pop()}")
print(f"Is the stack empty? {stack.is_empty()}")

'''
```

Output:



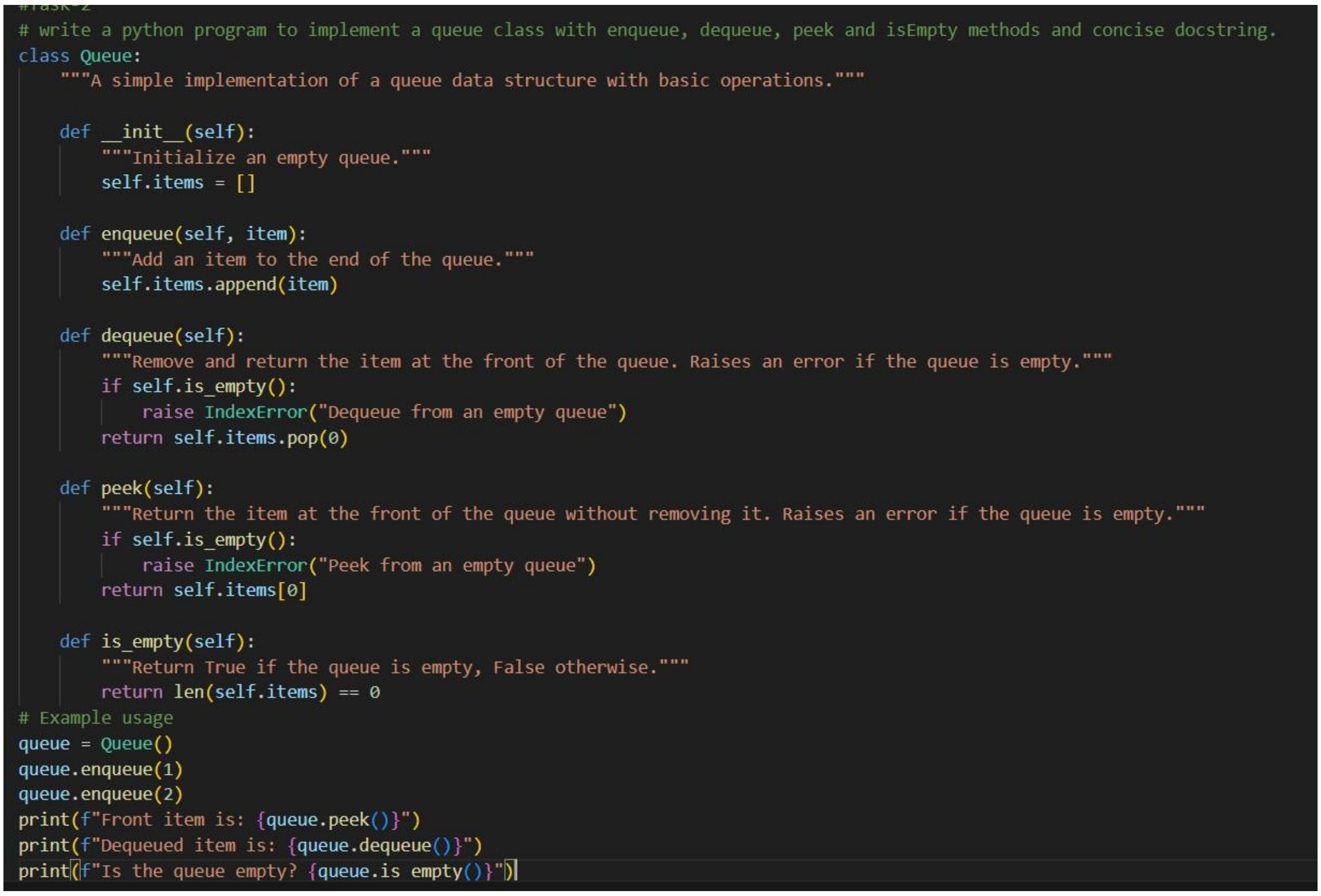
```
Top item is: 2
Popped item is: 2
Is the stack empty? False
PS C:\Users\saite\Downloads>
```

Justification:

The Stack class provides a clear and concise implementation of a stack data structure, with methods for pushing, popping, peeking, and checking if the stack is empty.
Each method includes a docstring that explains its purpose and behavior, making the code easy to understand and maintain.

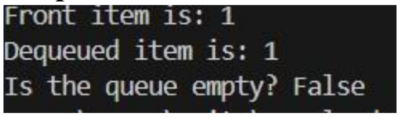
task Description #2 – Queue Implementation
Task: Use AI to implement a Queue using Python lists.
Sample Input Code: class
Queue:
pass
Expected Output:
• FIFO-based queue class with enqueue, dequeue, peek, and size methods.

Code:



```python
# write a python program to implement a queue class with enqueue, dequeue, peek and isEmpty methods and concise docstring.
class Queue:
    """A simple implementation of a queue data structure with basic operations."""

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

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

    def dequeue(self):
        """Remove and return the item at the front of the queue. Raises an error if the queue is empty."""
        if self.is_empty():
            raise IndexError("Dequeue from an empty queue")
        return self.items.pop(0)

    def peek(self):
        """Return the item at the front of the queue without removing it. Raises an error if the queue is empty."""
        if self.is_empty():
            raise IndexError("Peek from an empty queue")
        return self.items[0]

    def is_empty(self):
        """Return True if the queue is empty, False otherwise."""
        return len(self.items) == 0
# Example usage
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
print(f"Front item is: {queue.peek()}")
print(f"Dequeued item is: {queue.dequeue()}")
print(f"Is the queue empty? {queue.is_empty()}")
```

Output:



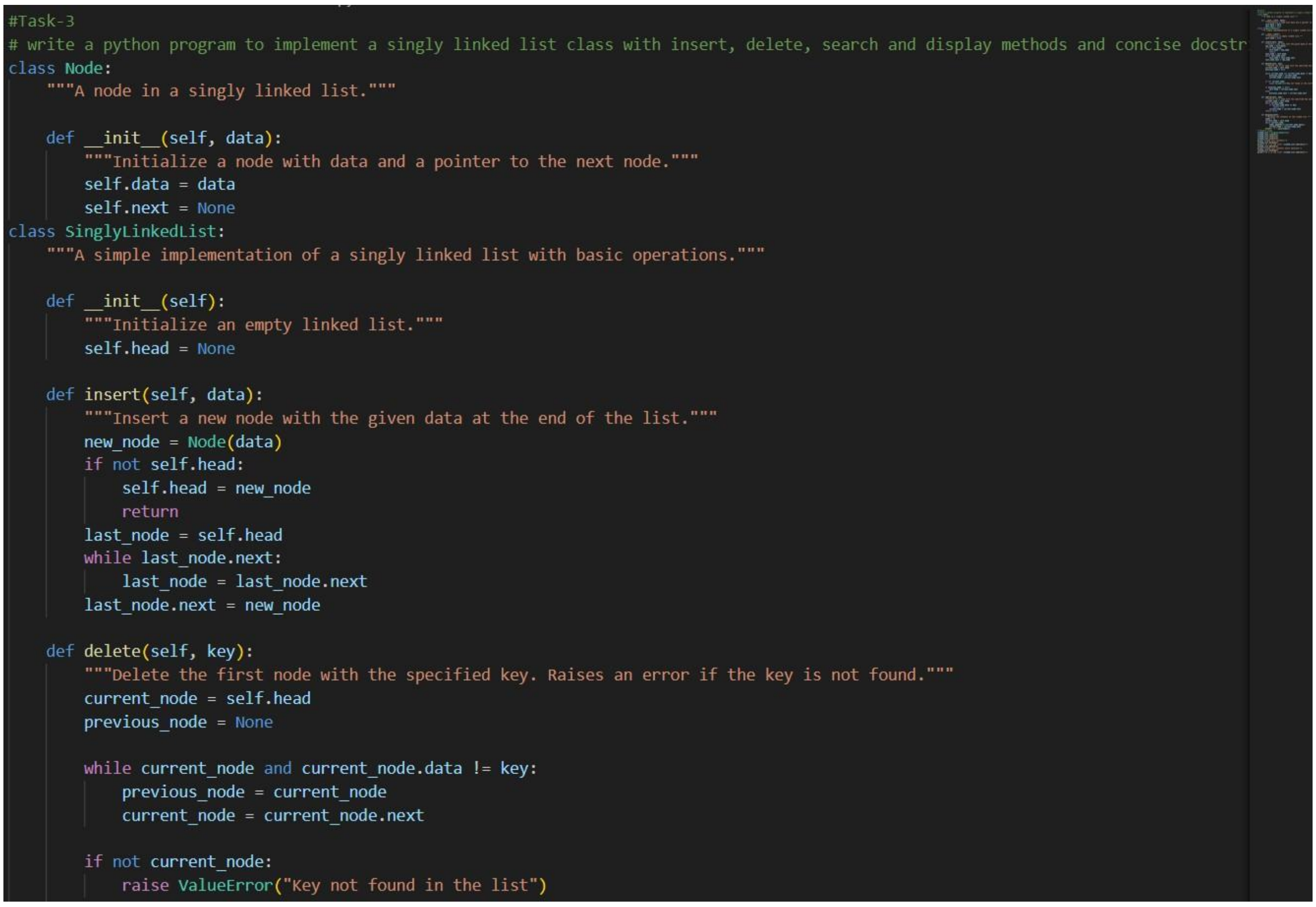
```
Front item is: 1
Dequeued item is: 1
Is the queue empty? False
```

Justification:
The Queue class is implemented with four methods: enqueue, dequeue, peek, and is_empty. Each method has a concise docstring explaining its purpose and behavior.

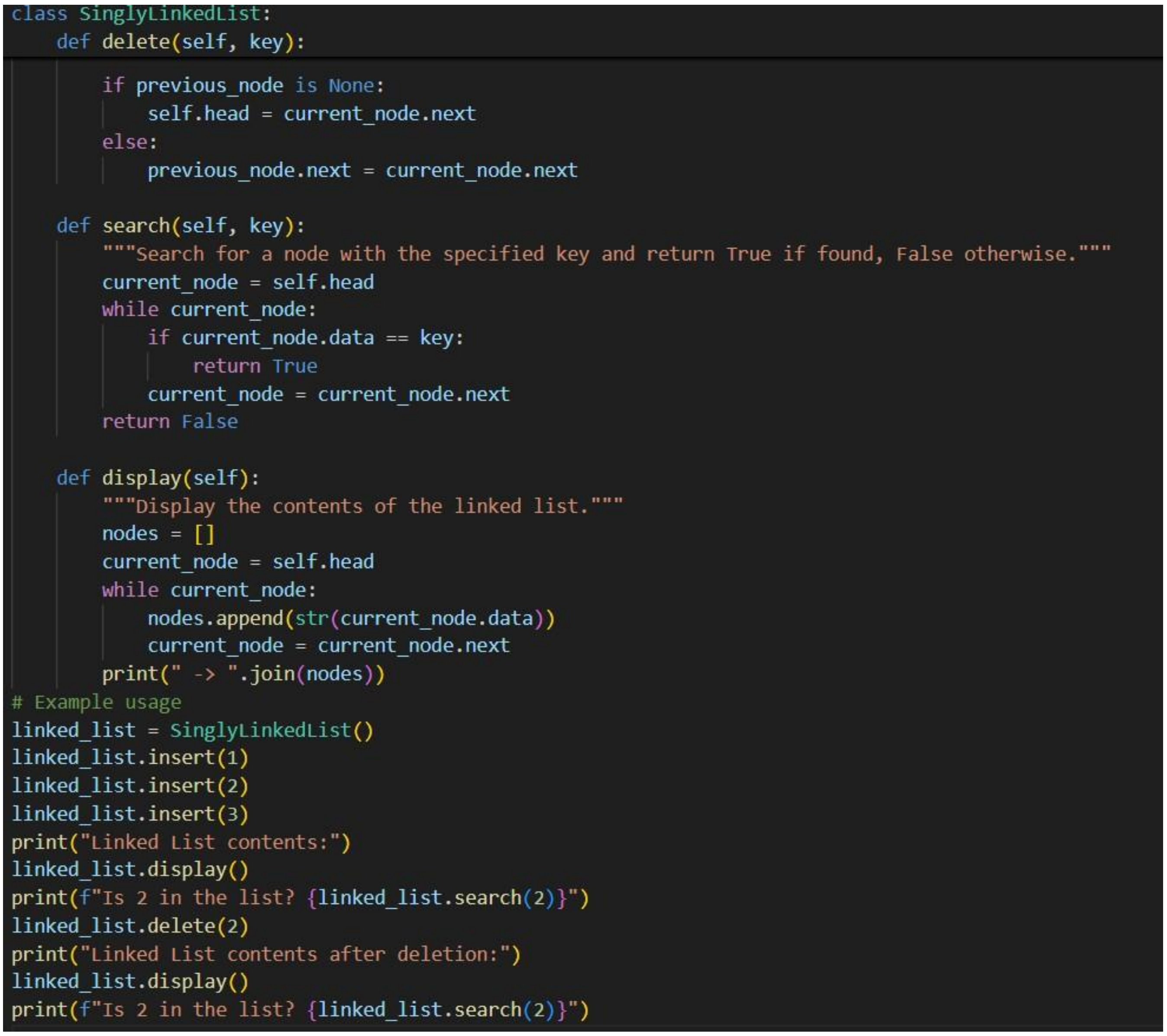
The enqueue method adds an item to the end of the queue, while the dequeue method removes and returns the item at the front of the queue.
The peek method allows us to see the front item without removing it, and the is_empty method checks if the queue is empty.
This implementation provides a clear and functional representation of a queue data structure.

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
Expected Output:
• A working linked list implementation with clear method documentation.

Code:

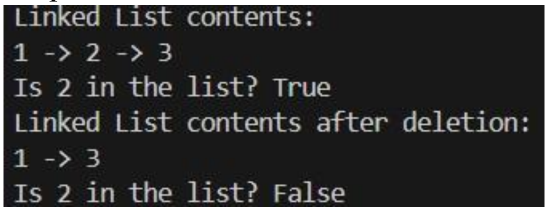


```python
#Task-3
# write a python program to implement a singly linked list class with insert, delete, search and display methods and concise docstr
class Node:
    """A node in a singly linked list."""

    def __init__(self, data):
        """Initialize a node with data and a pointer to the next node."""
        self.data = data
        self.next = None
class SinglyLinkedList:
    """A simple implementation of a singly linked list with basic operations."""

    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 delete(self, key):
        """Delete the first node with the specified key. Raises an error if the key is not found."""
        current_node = self.head
        previous_node = None

        while current_node and current_node.data != key:
            previous_node = current_node
            current_node = current_node.next

        if not current_node:
            raise ValueError("Key not found in the list")
```



```python
class SinglyLinkedList:
    def delete(self, key):

            if previous_node is None:
                self.head = current_node.next
            else:
                previous_node.next = current_node.next

    def search(self, key):
        """Search for a node with the specified key and return True if found, False otherwise."""
        current_node = self.head
        while current_node:
            if current_node.data == key:
                return True
            current_node = current_node.next
        return False

    def display(self):
        """Display the contents of the linked list."""
        nodes = []
        current_node = self.head
        while current_node:
            nodes.append(str(current_node.data))
            current_node = current_node.next
        print(" -> ".join(nodes))
# Example usage
linked_list = SinglyLinkedList()
linked_list.insert(1)
linked_list.insert(2)
linked_list.insert(3)
print("Linked List contents:")
linked_list.display()
print(f"Is 2 in the list? {linked_list.search(2)}")
linked_list.delete(2)
print("Linked List contents after deletion:")
linked_list.display()
print(f"Is 2 in the list? {linked_list.search(2)}")
```

Output:

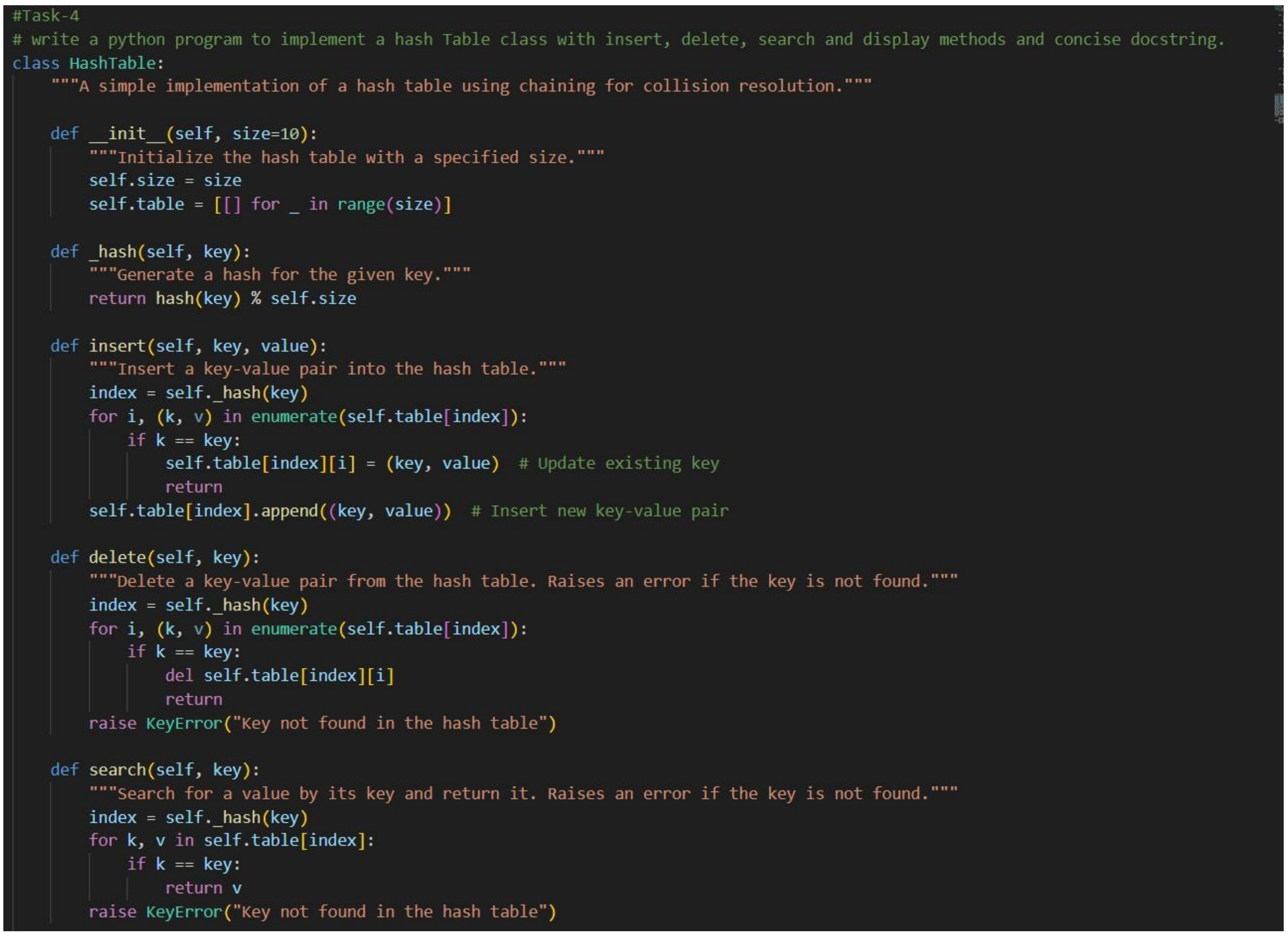


```
Linked List contents:
1 -> 2 -> 3
Is 2 in the list? True
Linked List contents after deletion:
1 -> 3
Is 2 in the list? False
```

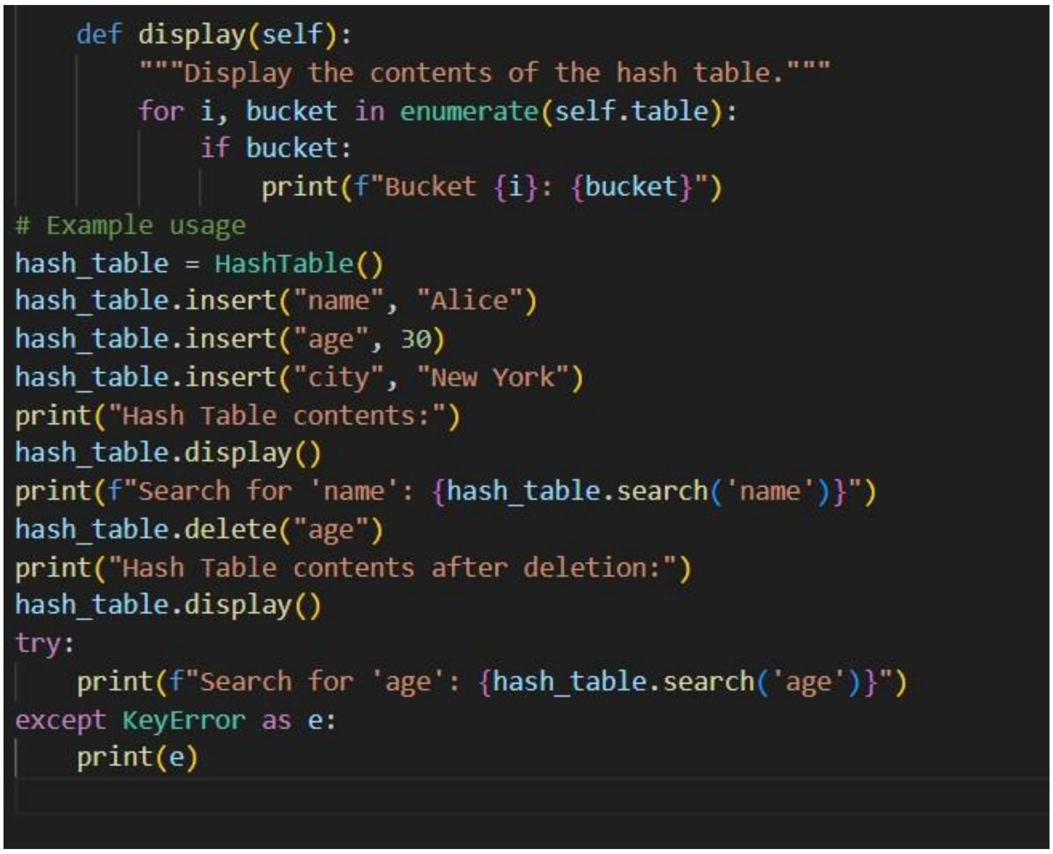
Justification:
The above code implements a singly linked list with the required methods: insert, delete, search, and display.
Each method is accompanied by a concise docstring that explains its purpose and functionality. ❼The example usage demonstrates how to use the linked list and verifies that the methods work as expected.
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
Expected Output:
• Collision handling using chaining, with well-commented methods

Code:

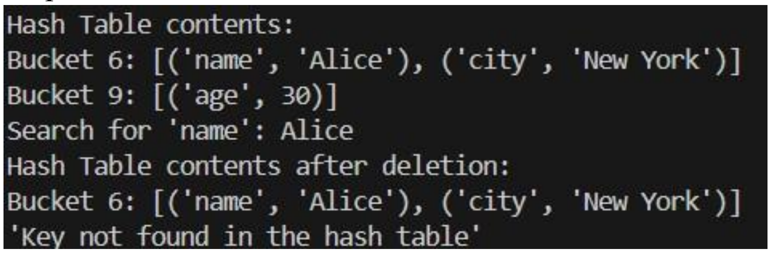


```
#Task-4
# write a python program to implement a hash Table class with insert, delete, search and display methods and concise docstring.
class HashTable:
    """A simple implementation of a hash table using chaining for collision resolution."""

    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 delete(self, key):
        """Delete a key-value pair from the hash table. Raises an error if the key is not found."""
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                return
        raise KeyError("Key not found in the hash table")

    def search(self, key):
        """Search for a value by its key and return it. Raises an error if the key is not found."""
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        raise KeyError("Key not found in the hash table")
```



```python
    def display(self):
        """Display the contents of the hash table."""
        for i, bucket in enumerate(self.table):
            if bucket:
                print(f"Bucket {i}: {bucket}")
# Example usage
hash_table = HashTable()
hash_table.insert("name", "Alice")
hash_table.insert("age", 30)
hash_table.insert("city", "New York")
print("Hash Table contents:")
hash_table.display()
print(f"Search for 'name': {hash_table.search('name')}")
hash_table.delete("age")
print("Hash Table contents after deletion:")
hash_table.display()
try:
    print(f"Search for 'age': {hash_table.search('age')}")
except KeyError as e:
    print(e)
```

Output:

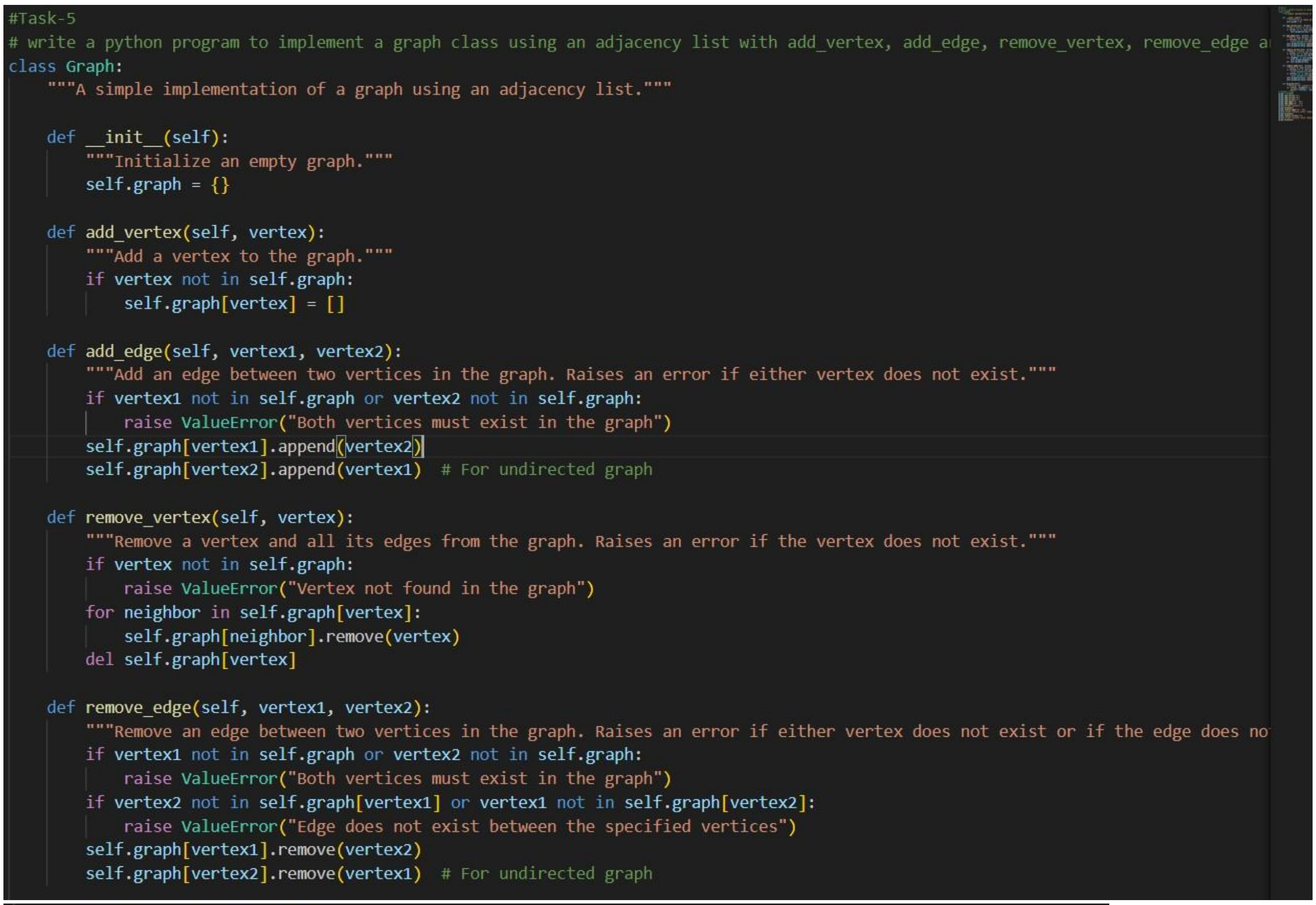


```
Hash Table contents:
Bucket 6: [('name', 'Alice'), ('city', 'New York')]
Bucket 9: [('age', 30)]
Search for 'name': Alice
Hash Table contents after deletion:
Bucket 6: [('name', 'Alice'), ('city', 'New York')]
'Key not found in the hash table'
```

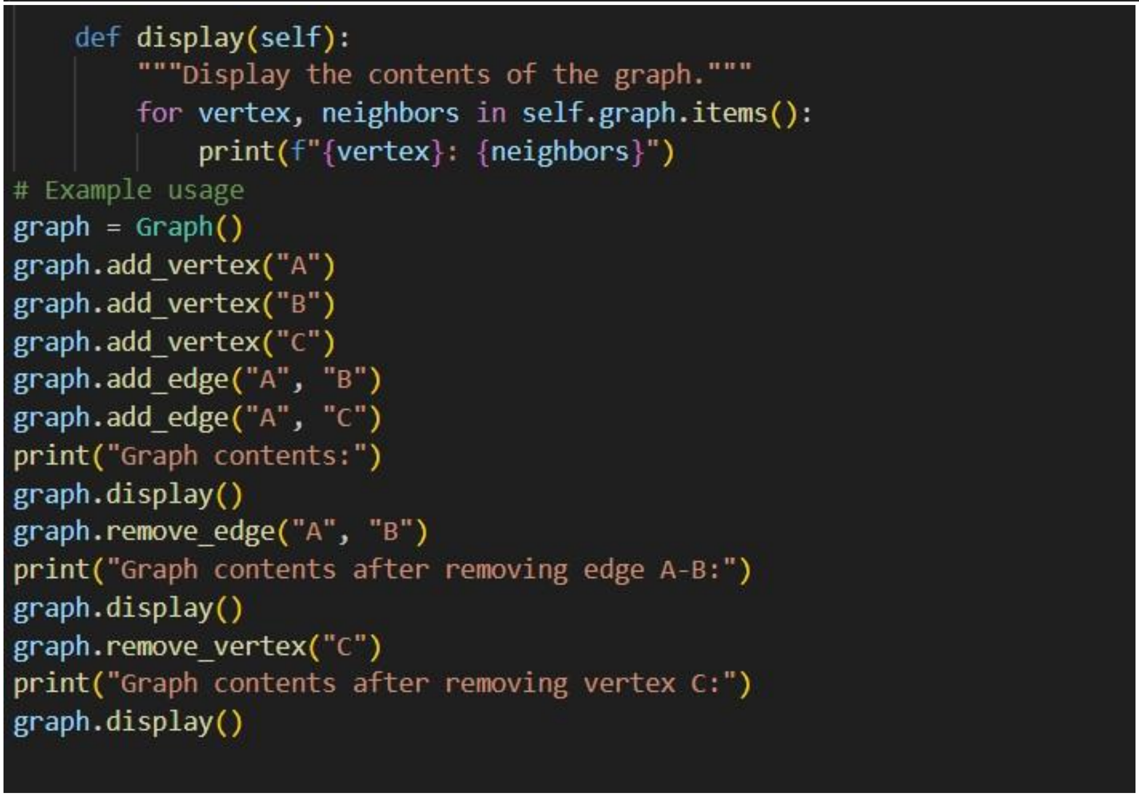
Justification:
The justification for this code is that it demonstrates the correct implementation of a hash table with all required methods (insert, delete, search, display).
It handles collisions using chaining and provides appropriate error handling for missing keys. ❼The example usage shows how to use each method and demonstrates the expected behavior of the hash table operations.

Task Description #5 – Graph Representation
Task: Use AI to implement a graph using an adjacency list.
Sample Input Code: class
Graph:
pass
Expected Output:
• Graph with methods to add vertices, add edges, and display connections.

Code:

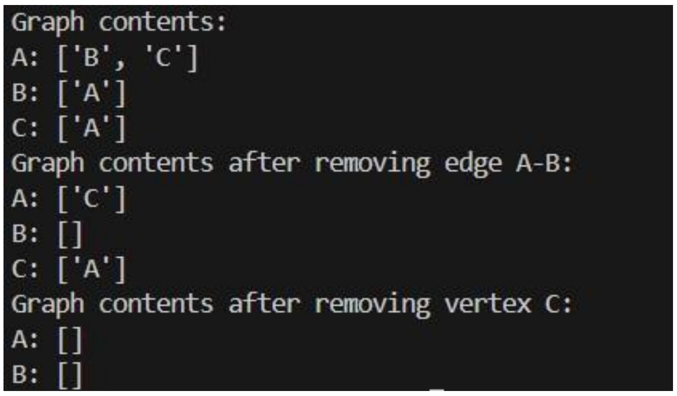


```python
#Task-5
# write a python program to implement a graph class using an adjacency list with add_vertex, add_edge, remove_vertex, remove_edge a
class Graph:
    """A simple implementation of a graph using an adjacency list."""

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

    def add_vertex(self, vertex):
        """Add a vertex to the graph."""
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        """Add an edge between two vertices in the graph. Raises an error if either vertex does not exist."""
        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)  # For undirected graph

    def remove_vertex(self, vertex):
        """Remove a vertex and all its edges from the graph. Raises an error if the vertex does not exist."""
        if vertex not in self.graph:
            raise ValueError("Vertex not found in the graph")
        for neighbor in self.graph[vertex]:
            self.graph[neighbor].remove(vertex)
        del self.graph[vertex]

    def remove_edge(self, vertex1, vertex2):
        """Remove an edge between two vertices in the graph. Raises an error if either vertex does not exist or if the edge does no
        if vertex1 not in self.graph or vertex2 not in self.graph:
            raise ValueError("Both vertices must exist in the graph")
        if vertex2 not in self.graph[vertex1] or vertex1 not in self.graph[vertex2]:
            raise ValueError("Edge does not exist between the specified vertices")
        self.graph[vertex1].remove(vertex2)
        self.graph[vertex2].remove(vertex1)  # For undirected graph
```



```python
    def display(self):
        """Display the contents of the graph."""
        for vertex, neighbors in self.graph.items():
            print(f"{vertex}: {neighbors}")
# 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 contents:")
graph.display()
graph.remove_edge("A", "B")
print("Graph contents after removing edge A-B:")
graph.display()
graph.remove_vertex("C")
print("Graph contents after removing vertex C:")
graph.display()
```

Output:



```
Graph contents:
A: ['B', 'C']
B: ['A']
C: ['A']
Graph contents after removing edge A-B:
A: ['C']
B: []
C: ['A']
Graph contents after removing vertex C:
A: []
B: []
```

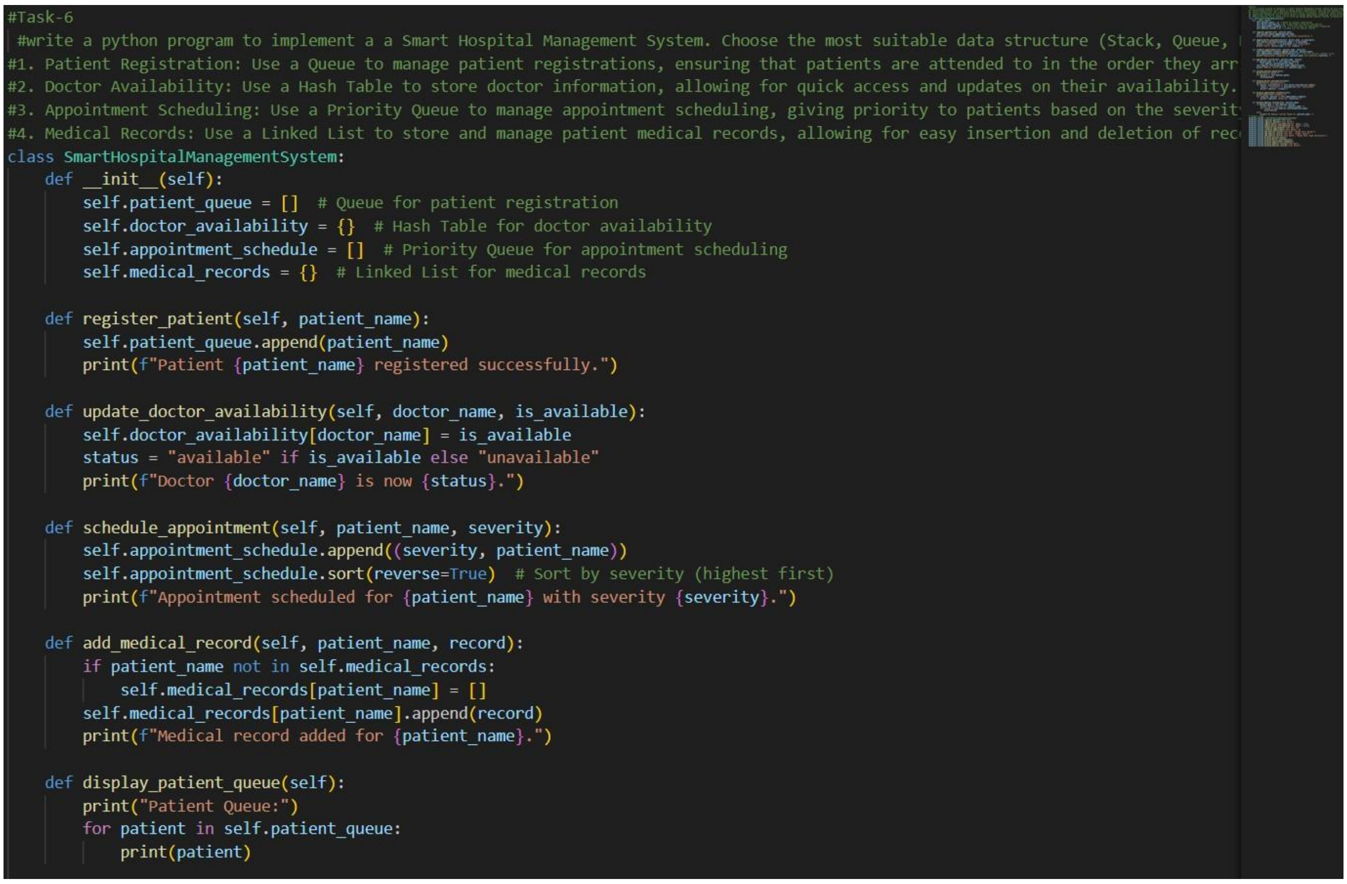
justification:

The Graph class is implemented using an adjacency list, which allows for efficient storage and retrieval of vertices and edges.
The methods provided allow for adding and removing vertices and edges, as well as displaying the graph's contents.
The use of error handling ensures that the operations are performed correctly and that the user is informed of any issues that arise.

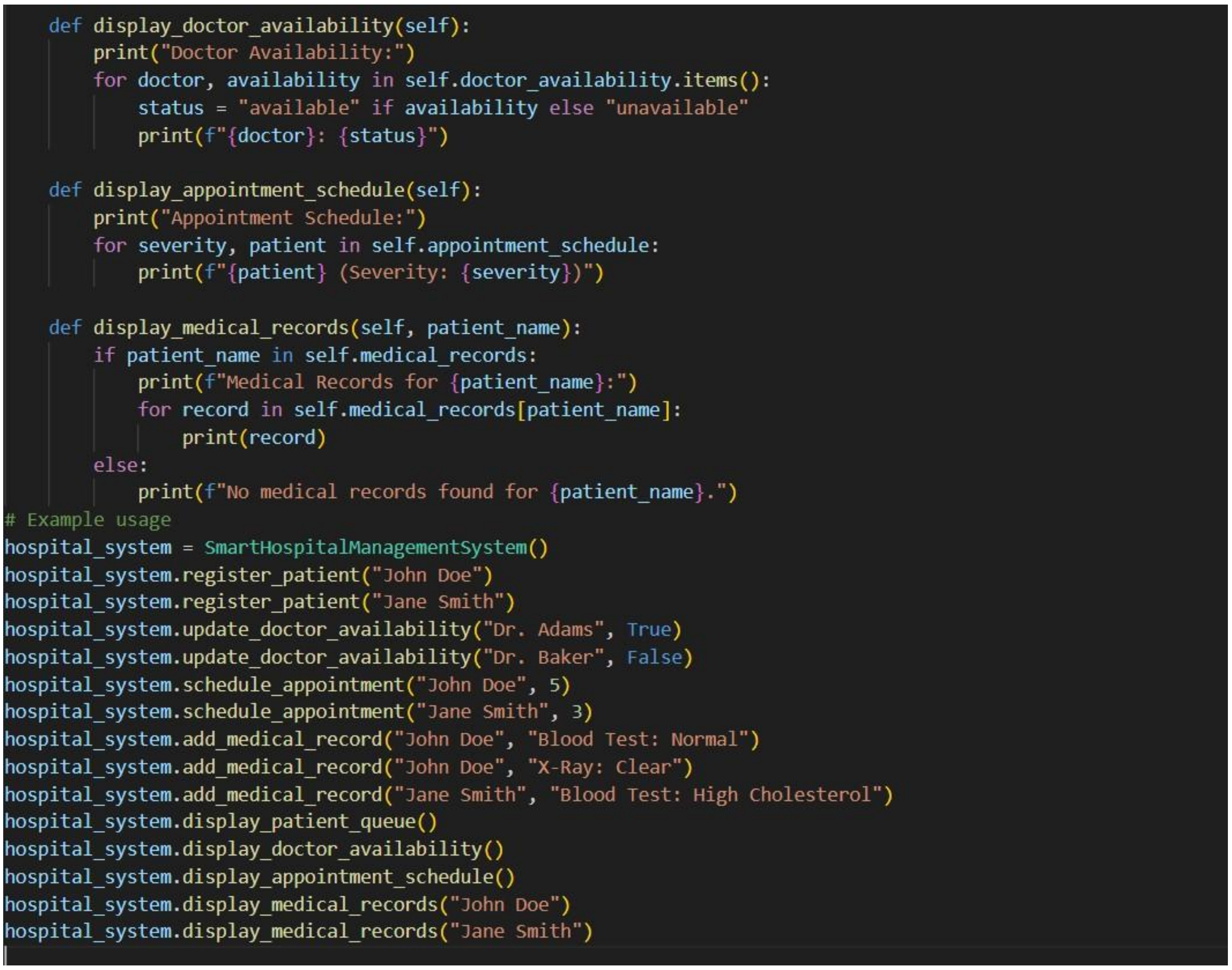
Task Description #6: Smart Hospital Management System – Data Structure Selection
A hospital wants to develop a Smart Hospital Management System that handles:
1. Patient Check-In System – Patients are registered and treated in order of arrival.
2. Emergency Case Handling – Critical patients must be treated 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. Expected Output:
• A table mapping feature → chosen data structure → justification. • A functional Python program implementing the chosen feature with comments and docstrings.

Code:

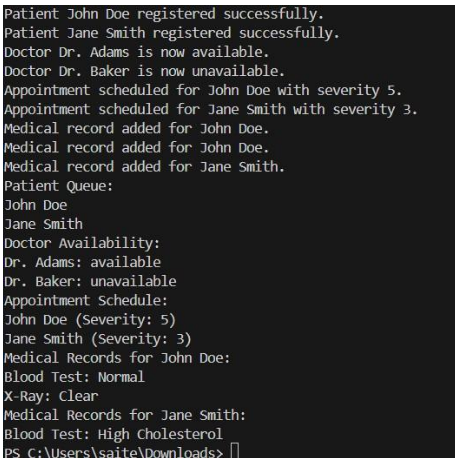


```
#Task-6
 #write a python program to implement a a Smart Hospital Management System. Choose the most suitable data structure (Stack, Queue,
#1. Patient Registration: Use a Queue to manage patient registrations, ensuring that patients are attended to in the order they arr
#2. Doctor Availability: Use a Hash Table to store doctor information, allowing for quick access and updates on their availability.
#3. Appointment Scheduling: Use a Priority Queue to manage appointment scheduling, giving priority to patients based on the severit
#4. Medical Records: Use a Linked List to store and manage patient medical records, allowing for easy insertion and deletion of rec
class SmartHospitalManagementSystem:
    def __init__(self):
        self.patient_queue = []  # Queue for patient registration
        self.doctor_availability = {}  # Hash Table for doctor availability
        self.appointment_schedule = []  # Priority Queue for appointment scheduling
        self.medical_records = {}  # Linked List for medical records

    def register_patient(self, patient_name):
        self.patient_queue.append(patient_name)
        print(f"Patient {patient_name} registered successfully.")

    def update_doctor_availability(self, doctor_name, is_available):
        self.doctor_availability[doctor_name] = is_available
        status = "available" if is_available else "unavailable"
        print(f"Doctor {doctor_name} is now {status}.")

    def schedule_appointment(self, patient_name, severity):
        self.appointment_schedule.append((severity, patient_name))
        self.appointment_schedule.sort(reverse=True)  # Sort by severity (highest first)
        print(f"Appointment scheduled for {patient_name} with severity {severity}.")

    def add_medical_record(self, patient_name, record):
        if patient_name not in self.medical_records:
            self.medical_records[patient_name] = []
        self.medical_records[patient_name].append(record)
        print(f"Medical record added for {patient_name}.")

    def display_patient_queue(self):
        print("Patient Queue:")
        for patient in self.patient_queue:
            print(patient)
```



```python
    def display_doctor_availability(self):
        print("Doctor Availability:")
        for doctor, availability in self.doctor_availability.items():
            status = "available" if availability else "unavailable"
            print(f"{doctor}: {status}")

    def display_appointment_schedule(self):
        print("Appointment Schedule:")
        for severity, patient in self.appointment_schedule:
            print(f"{patient} (Severity: {severity})")

    def display_medical_records(self, patient_name):
        if patient_name in self.medical_records:
            print(f"Medical Records for {patient_name}:")
            for record in self.medical_records[patient_name]:
                print(record)
        else:
            print(f"No medical records found for {patient_name}.")
# Example usage
hospital_system = SmartHospitalManagementSystem()
hospital_system.register_patient("John Doe")
hospital_system.register_patient("Jane Smith")
hospital_system.update_doctor_availability("Dr. Adams", True)
hospital_system.update_doctor_availability("Dr. Baker", False)
hospital_system.schedule_appointment("John Doe", 5)
hospital_system.schedule_appointment("Jane Smith", 3)
hospital_system.add_medical_record("John Doe", "Blood Test: Normal")
hospital_system.add_medical_record("John Doe", "X-Ray: Clear")
hospital_system.add_medical_record("Jane Smith", "Blood Test: High Cholesterol")
hospital_system.display_patient_queue()
hospital_system.display_doctor_availability()
hospital_system.display_appointment_schedule()
hospital_system.display_medical_records("John Doe")
hospital_system.display_medical_records("Jane Smith")
```

Ouput:



```
Patient John Doe registered successfully.
Patient Jane Smith registered successfully.
Doctor Dr. Adams is now available.
Doctor Dr. Baker is now unavailable.
Appointment scheduled for John Doe with severity 5.
Appointment scheduled for Jane Smith with severity 3.
Medical record added for John Doe.
Medical record added for John Doe.
Medical record added for Jane Smith.
Patient Queue:
John Doe
Jane Smith
Doctor Availability:
Dr. Adams: available
Dr. Baker: unavailable
Appointment Schedule:
John Doe (Severity: 5)
Jane Smith (Severity: 3)
Medical Records for John Doe:
Blood Test: Normal
X-Ray: Clear
Medical Records for Jane Smith:
Blood Test: High Cholesterol
PS C:\Users\saite\Downloads>
```

justification:
Patient Registration: Queue (list) is used to maintain order of patient arrivals.
Doctor Availability: Hash Table (dictionary) is used for O(1) access and updates.
Appointment Scheduling: Priority Queue (list sorted by severity) is used to prioritize appointments.

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:

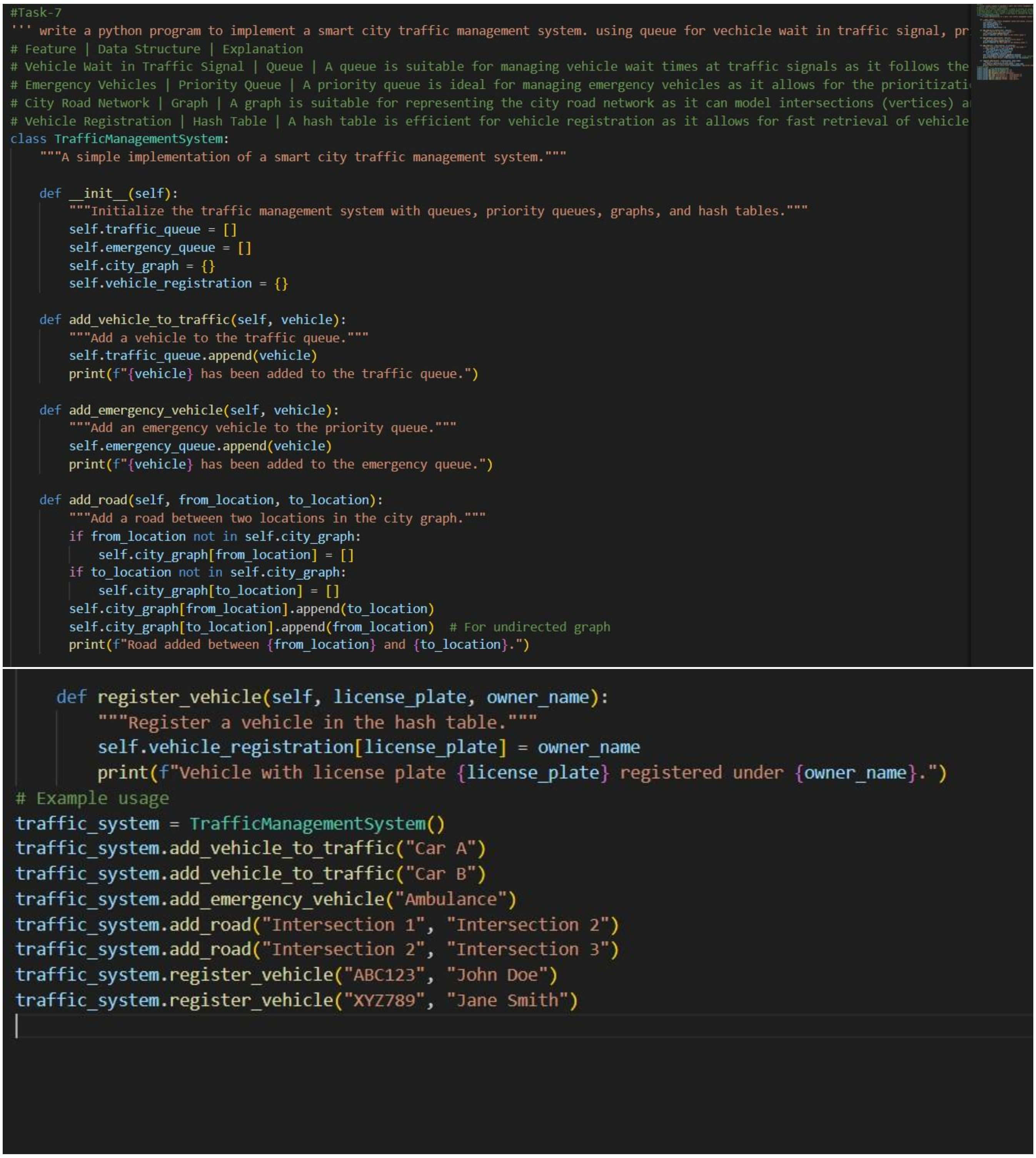
o Stack o Queue o Priority Queue o Linked List o Binary Search Tree (BST) o Graph o Hash Table o Deque

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

• Implement one selected feature as a working Python program with AI-assisted code generation.

Expected Output:

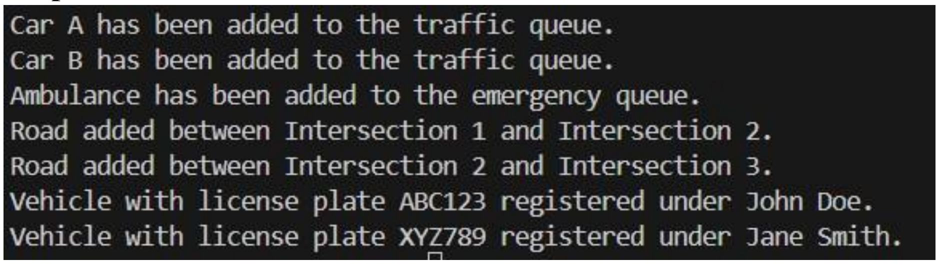
• A table mapping feature → chosen data structure → justification. • A functional Python program implementing the chosen feature with comments and docstrings



```python
#Task-7
''' write a python program to implement a smart city traffic management system. using queue for vechicle wait in traffic signal, pr
# Feature | Data Structure | Explanation
# Vehicle Wait in Traffic Signal | Queue | A queue is suitable for managing vehicle wait times at traffic signals as it follows the
# Emergency Vehicles | Priority Queue | A priority queue is ideal for managing emergency vehicles as it allows for the prioritizati
# City Road Network | Graph | A graph is suitable for representing the city road network as it can model intersections (vertices) a
# Vehicle Registration | Hash Table | A hash table is efficient for vehicle registration as it allows for fast retrieval of vehicle
class TrafficManagementSystem:
    """A simple implementation of a smart city traffic management system."""

    def __init__(self):
        """Initialize the traffic management system with queues, priority queues, graphs, and hash tables."""
        self.traffic_queue = []
        self.emergency_queue = []
        self.city_graph = {}
        self.vehicle_registration = {}

    def add_vehicle_to_traffic(self, vehicle):
        """Add a vehicle to the traffic queue."""
        self.traffic_queue.append(vehicle)
        print(f"{vehicle} has been added to the traffic queue.")

    def add_emergency_vehicle(self, vehicle):
        """Add an emergency vehicle to the priority queue."""
        self.emergency_queue.append(vehicle)
        print(f"{vehicle} has been added to the emergency queue.")

    def add_road(self, from_location, to_location):
        """Add a road between two locations in the city graph."""
        if from_location not in self.city_graph:
            self.city_graph[from_location] = []
        if to_location not in self.city_graph:
            self.city_graph[to_location] = []
        self.city_graph[from_location].append(to_location)
        self.city_graph[to_location].append(from_location)  # For undirected graph
        print(f"Road added between {from_location} and {to_location}.")
```

```python
    def register_vehicle(self, license_plate, owner_name):
        """Register a vehicle in the hash table."""
        self.vehicle_registration[license_plate] = owner_name
        print(f"Vehicle with license plate {license_plate} registered under {owner_name}.")
# Example usage
traffic_system = TrafficManagementSystem()
traffic_system.add_vehicle_to_traffic("Car A")
traffic_system.add_vehicle_to_traffic("Car B")
traffic_system.add_emergency_vehicle("Ambulance")
traffic_system.add_road("Intersection 1", "Intersection 2")
traffic_system.add_road("Intersection 2", "Intersection 3")
traffic_system.register_vehicle("ABC123", "John Doe")
traffic_system.register_vehicle("XYZ789", "Jane Smith")
```

Output:



```
Car A has been added to the traffic queue.
Car B has been added to the traffic queue.
Ambulance has been added to the emergency queue.
Road added between Intersection 1 and Intersection 2.
Road added between Intersection 2 and Intersection 3.
Vehicle with license plate ABC123 registered under John Doe.
Vehicle with license plate XYZ789 registered under Jane Smith.
```

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

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

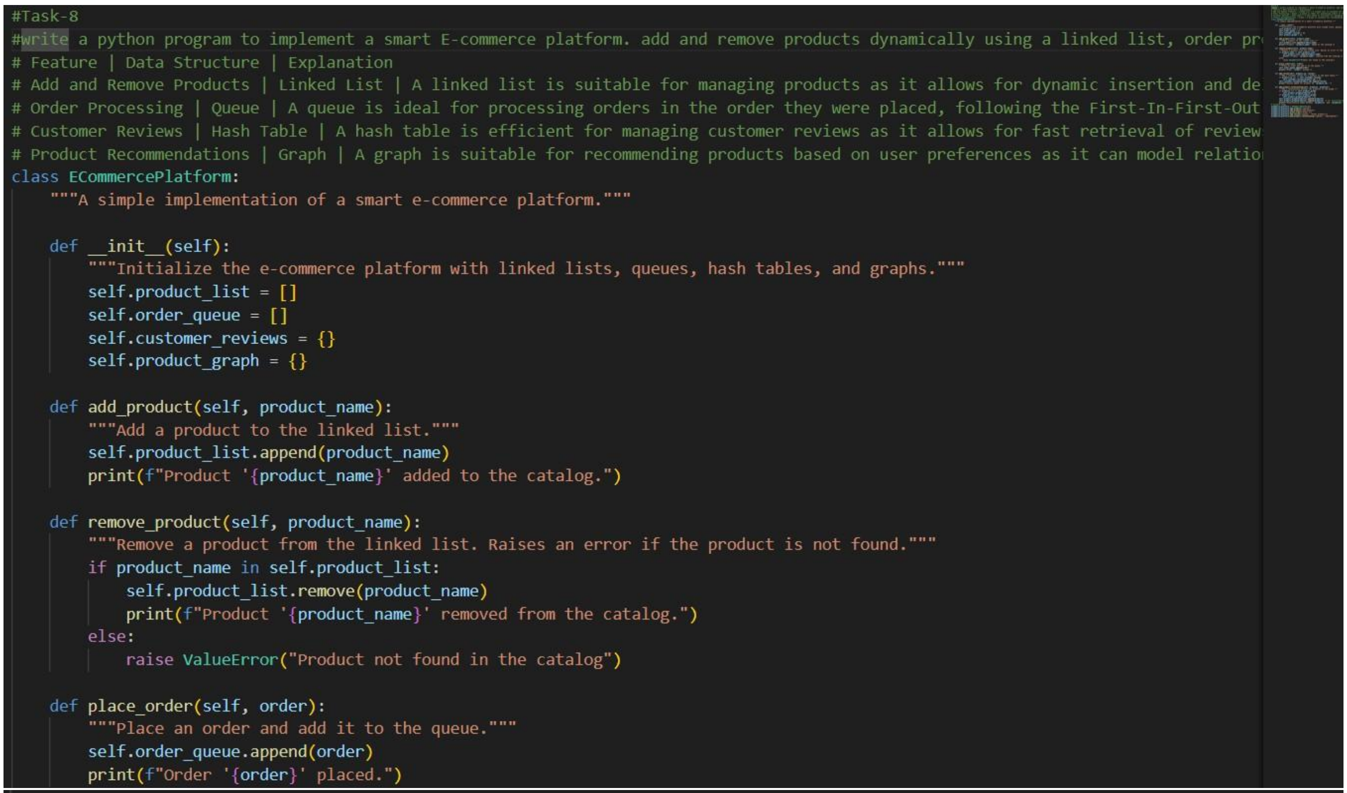
1. **Shopping Cart Management – Add and remove products dynamically.**
2. **Order Processing System – Orders processed in the order they are placed.**
3. **Top-Selling Products Tracker – Products ranked by sales count.**
4. **Product Search Engine – Fast lookup of products using product ID.**
5. **Delivery Route Planning – Connect warehouses and delivery locations.**

Student Task

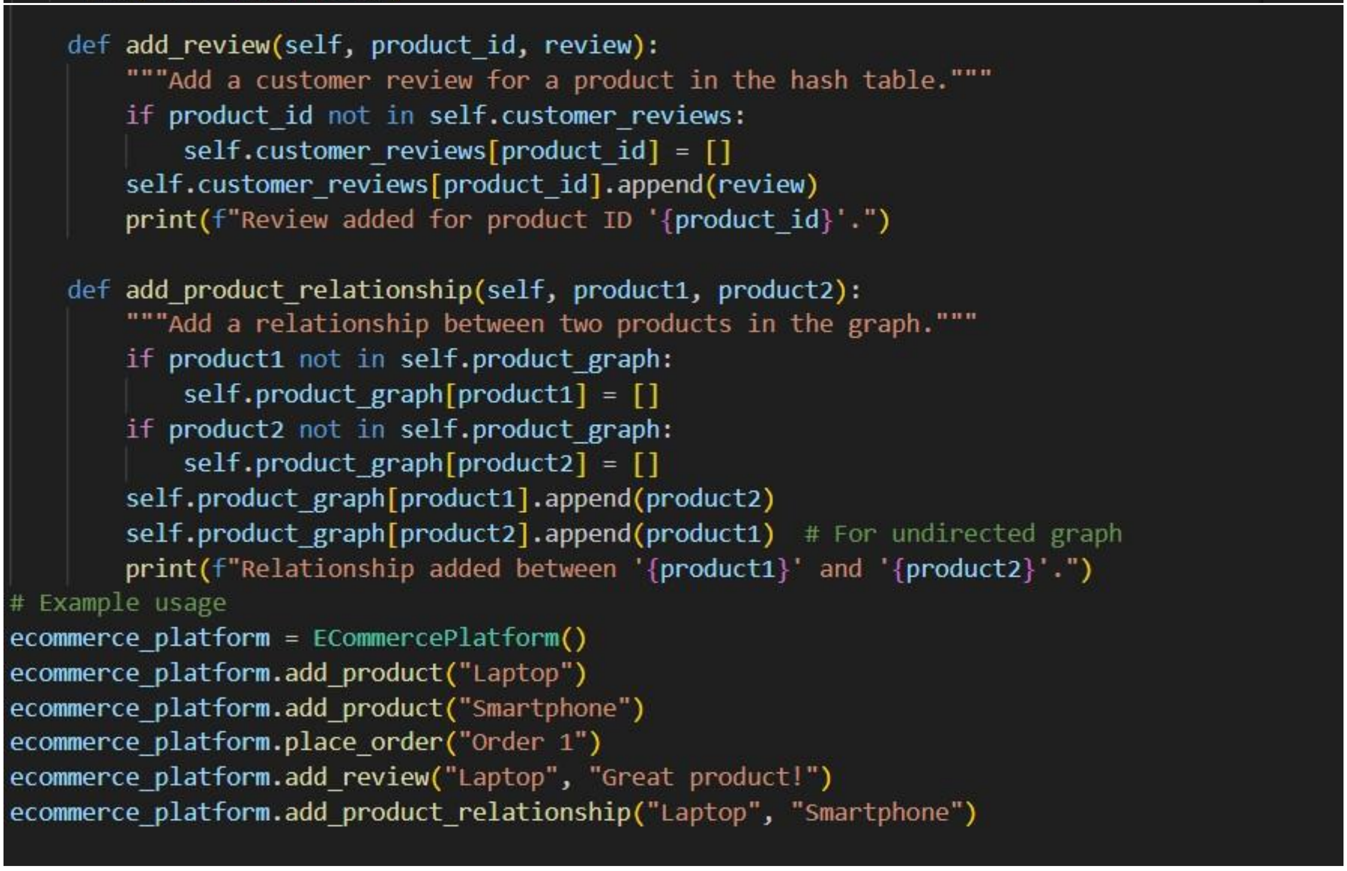
- **For each feature, select the most appropriate data structure from the list below:**
  o Stack o Queue o Priority Queue o Linked List o Binary Search Tree (BST) o Graph o Hash Table o Deque
- **Justify your choice in 2–3 sentences per feature.**
- **Implement one selected feature as a working Python program with AI-assisted code generation.**

Expected Output:

- **A table mapping feature → chosen data structure → justification. • A functional Python program implementing the chosen feature with comments and docstrings**

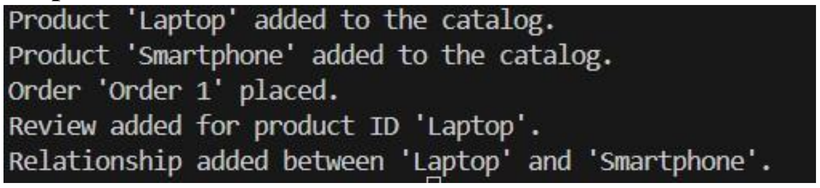


```python
#Task-8
#write a python program to implement a smart E-commerce platform. add and remove products dynamically using a linked list, order pr
# Feature | Data Structure | Explanation
# Add and Remove Products | Linked List | A linked list is suitable for managing products as it allows for dynamic insertion and de
# Order Processing | Queue | A queue is ideal for processing orders in the order they were placed, following the First-In-First-Out
# Customer Reviews | Hash Table | A hash table is efficient for managing customer reviews as it allows for fast retrieval of review
# Product Recommendations | Graph | A graph is suitable for recommending products based on user preferences as it can model relatio
class ECommercePlatform:
    """A simple implementation of a smart e-commerce platform."""

    def __init__(self):
        """Initialize the e-commerce platform with linked lists, queues, hash tables, and graphs."""
        self.product_list = []
        self.order_queue = []
        self.customer_reviews = {}
        self.product_graph = {}

    def add_product(self, product_name):
        """Add a product to the linked list."""
        self.product_list.append(product_name)
        print(f"Product '{product_name}' added to the catalog.")

    def remove_product(self, product_name):
        """Remove a product from the linked list. Raises an error if the product is not found."""
        if product_name in self.product_list:
            self.product_list.remove(product_name)
            print(f"Product '{product_name}' removed from the catalog.")
        else:
            raise ValueError("Product not found in the catalog")

    def place_order(self, order):
        """Place an order and add it to the queue."""
        self.order_queue.append(order)
        print(f"Order '{order}' placed.")
```



```python
    def add_review(self, product_id, review):
        """Add a customer review for a product in the hash table."""
        if product_id not in self.customer_reviews:
            self.customer_reviews[product_id] = []
        self.customer_reviews[product_id].append(review)
        print(f"Review added for product ID '{product_id}'.")

    def add_product_relationship(self, product1, product2):
        """Add a relationship between two products in the graph."""
        if product1 not in self.product_graph:
            self.product_graph[product1] = []
        if product2 not in self.product_graph:
            self.product_graph[product2] = []
        self.product_graph[product1].append(product2)
        self.product_graph[product2].append(product1)  # For undirected graph
        print(f"Relationship added between '{product1}' and '{product2}'.")
# Example usage
ecommerce_platform = ECommercePlatform()
ecommerce_platform.add_product("Laptop")
ecommerce_platform.add_product("Smartphone")
ecommerce_platform.place_order("Order 1")
ecommerce_platform.add_review("Laptop", "Great product!")
ecommerce_platform.add_product_relationship("Laptop", "Smartphone")
```

Output:



```
Product 'Laptop' added to the catalog.
Product 'Smartphone' added to the catalog.
Order 'Order 1' placed.
Review added for product ID 'Laptop'.
Relationship added between 'Laptop' and 'Smartphone'.
```

justification:

The linked list allows for efficient management of the product catalog, enabling easy addition and removal of products without worrying about memory allocation.
The queue ensures that orders are processed in the order they were placed, maintaining fairness and efficiency.
The hash table provides fast access to customer reviews based on product IDs, allowing for quick retrieval of feedback.
The graph structure models relationships between products, enabling personalized recommendations based on user preferences and interactions.