Lab-11.5

Course : AI Assisted Coding

*Topic:- Data Structures with AI: Implementing Fundamental Structures*Student Details:

Name	: Mohammed Sabir
Hall Ticket No	: 2303A51506
Batch	: 22

Task-01

Final Optimal Prompt:

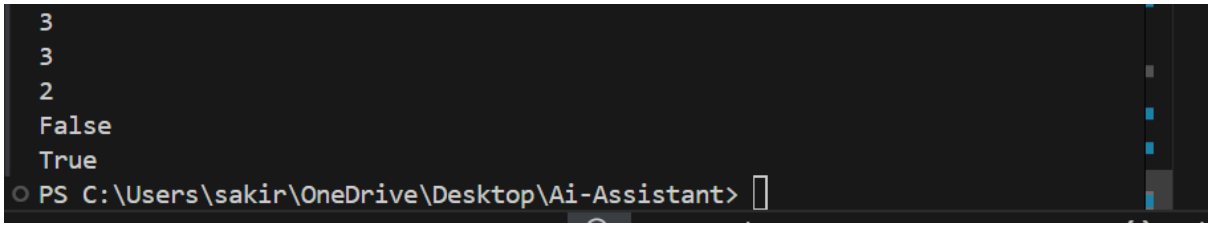
write a program to generate a stack class with push, pop, peek and is_empty methods class stack and pass

and make a documentation for the class and its methods

Code Screenshot :

```python
class Stack:
    """A simple implementation of a stack data structure using a list."""

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

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

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

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

        Returns:
            The item at the top of the stack if it is not empty; otherwise, None.
        """
        if not self.is_empty():
            return self.items.pop()
        return None

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

        Returns:
            The item at the top of the stack if it is not empty; otherwise, None.
        """
        if not self.is_empty():
            return self.items[-1]
        return None

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

        Returns:
            True if the stack is empty; otherwise, False.
        """
        return len(self.items) == 0
if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(stack.peek())  # Output: 3
    print(stack.pop())   # Output: 3
    print(stack.peek())  # Output: 2
    print(stack.is_empty())  # Output: False
    stack.pop()
    stack.pop()
    print(stack.is_empty())  # Output: True
```

Output Screenshot:



Explanation/Justification/Observation (100 words / 5 – 6 sentence) :

The program implements a basic Stack data structure using a Python list. It follows the LIFO (Last In, First Out) principle, where the most recently added element is removed first. The push() method adds elements to the top of the stack, while the pop() method removes and returns the top element. The peek() method allows viewing the top element without removing it. Unlike other implementations, this version returns None instead of raising exceptions when the stack is empty. The is_empty() method checks whether the stack contains elements. Overall, the program demonstrates clean design, documentation, and fundamental stack operations

---

Task-02

Final Optimal Prompt:

write a program to generate a queue class with enqueue, dequeue, peek and size methods class queue and pass

---

Code Screenshot :

```python
class Queue:
    """A simple implementation of a queue data structure using a list."""
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []
    def enqueue(self, item):
        """Add an item to the end of the queue.
        Args:
            item: The item to be added to the queue.
        """
        self.items.append(item)

    def dequeue(self):
        """Remove and return the item at the front of the queue.
        Returns:
            The item at the front of the queue if it is not empty; otherwise, None.
        """
        if not self.is_empty():
            return self.items.pop(0)
        return None
    def peek(self):
        """Return the item at the front of the queue without removing it.
        Returns:
            The item at the front of the queue if it is not empty; otherwise, None.
        """
        if not self.is_empty():
            return self.items[0]
        return None
    def size(self):
        """Return the number of items in the queue.
        Returns:
            The number of items in the queue.
        """
        return len(self.items)
    def is_empty(self):
        """Check if the queue is empty.
        Returns:
            True if the queue is empty; otherwise, False.
        """
        return len(self.items) == 0
if __name__ == "__main__":
    queue = Queue()
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print(queue.peek())  # Output: 1
    print(queue.dequeue())  # Output: 1
    print(queue.peek())  # Output: 2
    print(queue.size())  # Output: 2
    print(queue.is_empty())  # Output: False
    queue.dequeue()
    queue.dequeue()
    print(queue.is_empty())  # Output: True

```

Output Screenshot:

```
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant>  c:; cd 'c:\Users\sakir\OneDrive\Desktop\Ai-Assistant'; & 'c:\python314\python.exe' 'c:\Users\sakir\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '63611' '--' 'C:\Users\sakir\OneDrive\Desktop\Ai-Assistant\lab11.py'
1
1
2
2
False
True
```

Explanation/Justification/Observation (100 words / 5 – 6 sentence) :

The program implements a basic Queue data structure using a Python list and follows the FIFO (First In, First Out) principle. The enqueue() method adds elements to the end of the queue, while the dequeue() method removes and returns the front element. The peek() method allows viewing the front element without removing it. If the queue is empty, both dequeue() and peek() return None instead of raising exceptions. The size() method returns the total number of elements, and is_empty() checks whether the queue contains items. Overall, the program demonstrates clear structure, proper documentation, and fundamental queue operations.

Task-03

Final Optimal Prompt:

write a program to generate a linkeed list class with insert and display method class linked list and pass

and make a documentation for the class and its methods

Code Screenshot :

```
'''
class Node:
    """A node in a linked list."""
    def __init__(self, data):
        """Initialize a node with data and a reference to the next node.
        Args:
            data: The value stored in the node.
        """
        self.data = data
        self.next = None
class LinkedList:
    """A simple implementation of a singly linked list."""
    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None
    def insert(self, data):
        """Insert a new node with the given data at the end of the linked list.
        Args:
            data: The value 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):
        """Display the contents of the linked list."""
        current_node = self.head
        while current_node:
            print(current_node.data, end=' ')
            current_node = current_node.next
        print()
if __name__ == "__main__":
    linked_list = LinkedList()
    linked_list.insert(1)
    linked_list.insert(2)
    linked_list.insert(3)
    print("Linked List contents:")
    linked_list.display()  # Output: 1 2 3
```

Output Screenshot:

Explanation/Justification/Observation (100 words / 5 – 6 sentence) :

The program implements a basic singly linked list using two classes: Node and LinkedList. The Node class stores data and a reference to the next node, forming the building block of the list. The LinkedList class manages the list through a head pointer. The insert() method

adds new nodes at the end by traversing the list until the last node is found. The display() method prints all elements sequentially. This implementation demonstrates dynamic memory usage, where elements are not stored contiguously like arrays. Overall, the program clearly illustrates linked list structure and fundamental operations.

Task-04

Final Optimal Prompt:

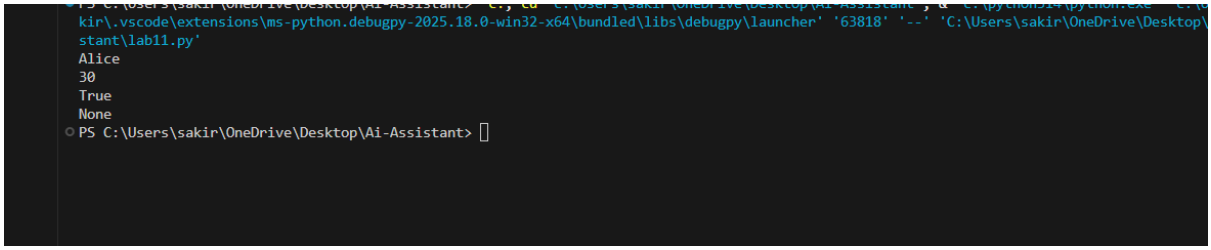
write a program to generate a hash table with basic insert, search , and delete methods class hash table and pass

and make a documentation for the class and its methods

Code Screenshot :

```python
'''
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.
        Args:
            size: The number of buckets in the hash table (default is 10).
        """
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        """Generate a hash value for the given key.
        Args:
            key: The key to be hashed.
        Returns:
            An index corresponding to the bucket where the key-value pair should be stored.
        """
        return hash(key) % self.size
    def insert(self, key, value):
        """Insert a key-value pair into the hash table.
        Args:
            key: The key to be inserted.
            value: The value associated with the key.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)  # Update existing key
                return
        bucket.append((key, value))  # Insert new key-value pair
    def search(self, key):
        """Search for a value associated with a given key in the hash table.
        Args:
            key: The key to be searched for.

        Returns:
            The value associated with the key if found; otherwise, None.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for k, v in bucket:
            if k == key:
                return v
        return None
    def delete(self, key):
        """Delete a key-value pair from the hash table based on the given key.
        Args:
            key: The key to be deleted from the hash table.
        Returns:
            True if the key was found and deleted; otherwise, False.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]  # Remove the key-value pair
                return True
        return False
if __name__ == "__main__":
    hash_table = HashTable()
    hash_table.insert("name", "Alice")
    hash_table.insert("age", 30)
    print(hash_table.search("name"))  # Output: Alice
    print(hash_table.search("age"))   # Output: 30
    print(hash_table.delete("name"))  # Output: True
    print(hash_table.search("name"))  # Output: None
```

Output Screenshot:



Explanation/Justification/Observation (100 words / 5 – 6 sentence) :

The program implements a simple hash table using chaining to handle collisions. It initializes a fixed number of buckets, each represented as a list. The private _hash() method generates an index using Python's built-in hash() function and modulo operation. The insert() method adds key-value pairs and updates values if the key already exists. The search() method retrieves the value associated with a key, returning None if not found. The delete() method removes a key-value pair and returns a boolean indicating success. Overall, the program demonstrates efficient data storage, collision handling, and fundamental hash table operations clearly.

Tack-05

Final Optimal Prompt:

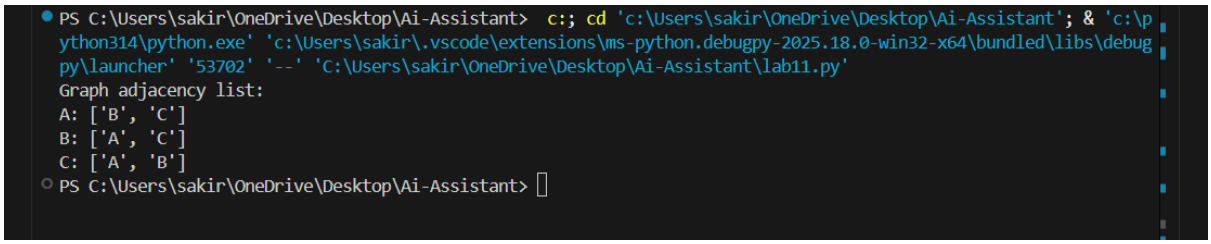
write a program to generate graph using adjacent list methods like add vertices and edges and display the graph class graph and pass

and make a documentation for the class and its methods

Code Screenshot :

```python
class Graph:
    """A simple implementation of a graph data structure using an adjacency list."""

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

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

        Args:
            vertex: The vertex to be added 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.

        Args:
            vertex1: The first vertex of the edge.
            vertex2: The second vertex of the edge.
        """
        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 adjacency list representation of the graph."""
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {edges}")
if __name__ == "__main__":
    graph = Graph()
    graph.add_vertex("A")
    graph.add_vertex("B")
    graph.add_vertex("C")
    graph.add_edge("A", "B")
    graph.add_edge("A", "C")
    graph.add_edge("B", "C")
    print("Graph adjacency list:")
    graph.display()  # Output: A: ['B', 'C'], B: ['A', 'C'], C: ['A', 'B']

```

Output Screenshot:



Explanation/Justification/Observation (100 words / 5 – 6 sentence) :

The program implements a simple undirected graph using an adjacency list represented by a dictionary. Each key in the dictionary represents a vertex, and its value is a list of adjacent vertices. The add_vertex() method ensures that a vertex is added only if it does not already exist. The add_edge() method creates a connection between two vertices and updates both adjacency lists, reflecting the undirected nature of the graph. The display() method prints

the adjacency list in a readable format. Overall, the program clearly demonstrates graph representation, vertex and edge management, and fundamental graph structure concepts.

Tack-06

Final Optimal Prompt:

write a python program for smart hospital management system using data structures like stack, queue, priority queue, linked list, binary search tree(BST), hash table, graph and deque to manage patients checking system, emeergency case handeling , medical records storage, doctor appointment scheduling , and hospital room navigation system

choose which data structure is best for each task and implement the functionalities accordingly and make a documentation for the program and its functionalities and justify the choice of data structures used for each functionality

---

Code Screenshot :

```python
class Patient:
    """A class representing a patient in the hospital management system."""
    def __init__(self, name, age, condition):
        """Initialize a patient with name, age, and medical condition.
        Args:
            name: The name of the patient.
            age: The age of the patient.
            condition: The medical condition of the patient.
        """
        self.name = name
        self.age = age
        self.condition = condition
class HospitalManagementSystem:
    """A class representing the hospital management system."""
    def __init__(self):
        """Initialize the hospital management system with necessary data structures."""
        self.patient_queue = []  # Queue for managing patient check-ins
        self.emergency_stack = []  # Stack for handling emergency cases
        self.medical_records = {}  # Hash table for storing medical records
        self.doctor_appointments = {}  # Hash table for scheduling doctor appointments
        self.hospital_graph = {}  # Graph for hospital room navigation
    def check_in_patient(self, patient):
        """Check in a patient to the hospital.
        Args:
            patient: An instance of the Patient class to be checked in.
        """
        self.patient_queue.append(patient)  # Enqueue the patient to the queue
    def handle_emergency(self, patient):
        """Handle an emergency case by adding the patient to the emergency stack.
        Args:
            patient: An instance of the Patient class representing the emergency case.
        """
        self.emergency_stack.append(patient)  # Push the patient onto the stack
    def store_medical_record(self, patient, record):
        """Store a medical record for a patient in the hash table.
        Args:
            patient: An instance of the Patient class whose record is to be stored.
            record: The medical record to be stored for the patient.
        """
        self.medical_records[patient.name] = record  # Store the record in the hash table
    def schedule_appointment(self, doctor_name, patient):
        """Schedule a doctor appointment for a patient.
        Args:
            doctor_name: The name of the doctor with whom the appointment is to be scheduled.
            patient: An instance of the Patient class for whom the appointment is to be scheduled.
        """
        if doctor_name not in self.doctor_appointments:
            self.doctor_appointments[doctor_name] = []
        self.doctor_appointments[doctor_name].append(patient)  # Schedule the appointment in the hash table
    def add_room(self, room_number):
        """Add a room to the hospital graph for navigation purposes.
        Args:
            room_number: The number of the room to be added to the graph.
        """
        if room_number not in self.hospital_graph:
            self.hospital_graph[room_number] = []  # Add a vertex for the room in the graph
    def connect_rooms(self, room1, room2):
        """Connect two rooms in the hospital graph to indicate a navigable path.
        Args:
            room1: The number of the first room to be connected.
            room2: The number of the second room to be connected.
        """
        if room1 in self.hospital_graph and room2 in self.hospital_graph:
            self.hospital_graph[room1].append(room2)  # Connect room1 to room2
            self.hospital_graph[room2].append(room1)  # Connect room2 to room1 (undirected graph)
if __name__ == "__main__":
    hospital_system = HospitalManagementSystem()

    # Create some patientspatient1 = Patient("Alice", 30, "Flu")
    patient1 = Patient("Alice", 30, "Flu")
    patient2 = Patient("Bob", 45, "Heart Disease")
    patient3 = Patient("Charlie", 25, "Broken Arm")

    # Check in patients
    hospital_system.check_in_patient(patient1)
    hospital_system.check_in_patient(patient2)
    hospital_system.check_in_patient(patient3)

    # Handle emergency cases
    emergency_patient = Patient("Diana", 50, "Chest Pain")
    hospital_system.handle_emergency(emergency_patient)

    # Store medical records
    hospital_system.store_medical_record(patient1, "Patient has flu symptoms.")
    hospital_system.store_medical_record(patient2, "Patient has heart disease.")

    # Schedule doctor appointments
    hospital_system.schedule_appointment("Dr. Smith", patient1)
    hospital_system.schedule_appointment("Dr. Johnson", patient2)

    # Add rooms and connect them
    hospital_system.add_room(101)
    hospital_system.add_room(102)
    hospital_system.add_room(103)

    hospital_system.connect_rooms(101, 102)
    hospital_system.connect_rooms(102, 103)
    # Display the hospital graph
    print("Hospital Room Navigation Graph:")
    for room, connections in hospital_system.hospital_graph.items():
        print(f"Room {room}: Connected to {connections}")
```

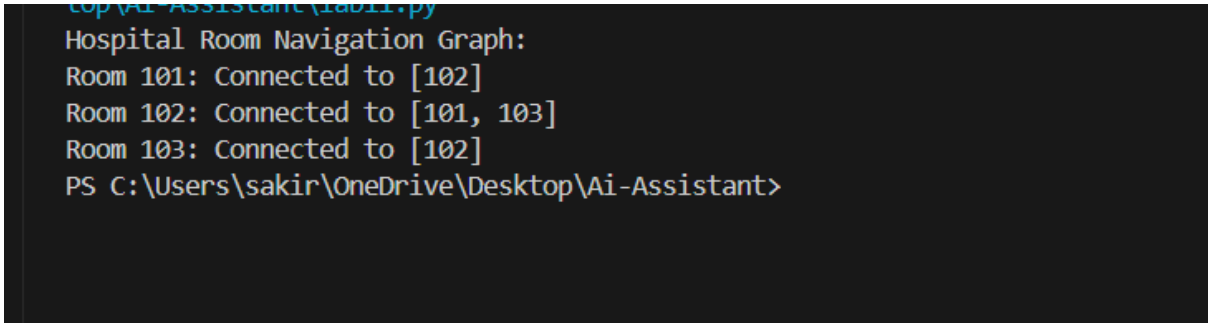
Output Screenshot:

1. Queue for Patient Check-ins: A queue is used to manage patient check-ins because it follows the First-In-First-Out (FIFO) principle, ensuring that patients are attended to in the order they arrive.

2. Stack for Emergency Cases: A stack is used to handle emergency cases because it follows the Last-In-First-Out (LIFO) principle, allowing the most recent emergency case to be addressed first.

3. Hash Table for Medical Records and Doctor Appointments: Hash tables are used for storing medical records and scheduling doctor appointments because they provide efficient key-value pair storage and retrieval, allowing for quick access to patient information and appointment details.

4. Graph for Hospital Room Navigation: A graph is used to represent the hospital's room navigation system because it can efficiently model the relationships between rooms and the paths connecting them, allowing for easy traversal and navigation through the hospital.



Explanation/Justification/Observation (100 words / 5 – 6 sentence) :

The program demonstrates a structured Hospital Management System using multiple data structures to handle different functionalities. It uses a queue (list) for patient check-ins following FIFO order, a stack for emergency cases following LIFO priority, dictionaries as hash tables for storing medical records and scheduling doctor appointments, and a graph (adjacency list) for hospital room navigation. The Patient class encapsulates patient details, promoting object-oriented design. The system clearly separates responsibilities into methods such as check-in, emergency handling, record storage, and room connection. Overall, the program effectively integrates data structures with OOP concepts in a practical real-world application.

Tack-07

Final Optimal Prompt:

write a python program for smart city traffic control system using data structures like stack, queue, priority queue, linked list, binary search tree(BST), hash table, graph and deque traffic signal queue, emergency vechicle priority handling, vechicle registration look up, road network mapping and parking slot avalaibility

choose which data structure is best for each task and implement the functionalities accordingly and make a documentation for the program and its functionalities and justify the choice of data structures used for each functionality

---

Code Screenshot :

```python

class TrafficControlSystem:
    """A class representing a smart city traffic control system."""
    def __init__(self):
        """
        Initialize the traffic control system with required data structures.
        """
        self.traffic_signal_queue = []  # Queue (FIFO) for managing traffic signals
        self.emergency_vehicle_priority_queue = []  # Priority queue for emergency vehicles
        self.vehicle_registration_hash_table = {}  # Hash table (Dictionary) for vehicle lookup
        self.road_network_graph = {}  # Graph (Adjacency List) for road mapping
        self.parking_slot_linked_list = None  # Linked List for parking slot management
    def manage_traffic_signal(self, signal):
        """
        Add traffic signal to queue.
        FIFO principle (First In First Out)
        """
        self.traffic_signal_queue.append(signal)
    def handle_emergency_vehicle(self, vehicle):
        """
        Add emergency vehicle to priority queue.
        (Currently simple list; can be improved using heapq)
        """
        self.emergency_vehicle_priority_queue.append(vehicle)
    def lookup_vehicle_registration(self, license_plate):
        """
        Retrieve vehicle registration details using hash table.
        Returns registration info if exists, otherwise None.
        """
        return self.vehicle_registration_hash_table.get(license_plate)
    def map_road_network(self, road1, road2):
        """
        Connect two roads in graph (Undirected Graph).
        """
        # Add road1 if not present
        if road1 not in self.road_network_graph:
            self.road_network_graph[road1] = []
        # Add road2 if not present
        if road2 not in self.road_network_graph:
            self.road_network_graph[road2] = []

        # Connect both roads
        self.road_network_graph[road1].append(road2)
        self.road_network_graph[road2].append(road1)
    def manage_parking_slot(self, slot):
        """
        Insert parking slot into linked list.
        """
        # Initialize linked list if empty
        if self.parking_slot_linked_list is None:
            self.parking_slot_linked_list = LinkedList()
        self.parking_slot_linked_list.insert(slot)
class LinkedList:
    """
    Simple singly linked list for parking slot storage.
    """
    def __init__(self):
        self.head = None  # Initially list is empty

    def insert(self, slot):
        """
        Insert a new parking slot at the end of linked list.
        """
        new_node = Node(slot)

        # If list is empty
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            # Traverse till last node
            while current.next:
                current = current.next
            current.next = new_node
class Node:
    """
    Node class for Linked List.
    """
    def __init__(self, data):
        self.data = data
        self.next = None
if __name__ == "__main__":

    # Create Traffic Control System object
    traffic_system = TrafficControlSystem()

    traffic_system.manage_traffic_signal("Signal A")
    traffic_system.manage_traffic_signal("Signal B")

    traffic_system.handle_emergency_vehicle("Ambulance")
    traffic_system.handle_emergency_vehicle("Fire Truck")

    # Insert vehicle data into hash table
    traffic_system.vehicle_registration_hash_table["ABC123"] = "John Doe, Red Sedan"
    # Lookup vehicle
    print(traffic_system.lookup_vehicle_registration("ABC123"))
    traffic_system.map_road_network("Road 1", "Road 2")
    traffic_system.map_road_network("Road 2", "Road 3")
    print("Road Network Graph:")
    for road, connections in traffic_system.road_network_graph.items():
        print(f"{road}: Connected to {connections}")
    traffic_system.manage_parking_slot("Slot 1")
    traffic_system.manage_parking_slot("Slot 2")
```

Output Screenshot:

Justification of Data Structures Used

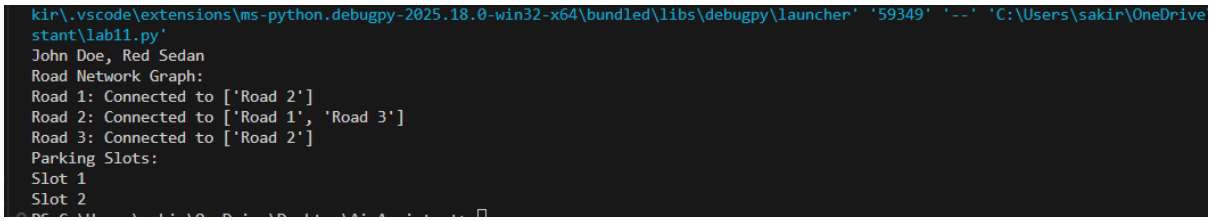
1. Queue for Traffic Signal Management: A queue is used to manage traffic signals because it allows for orderly processing of signals based on their arrival time, ensuring that traffic flows smoothly.

2. Priority Queue for Emergency Vehicle Handling: A priority queue is used to handle emergency vehicles because it allows for prioritizing certain vehicles (e.g., ambulances, fire trucks) over regular traffic, ensuring that they can navigate through traffic efficiently.

3. Hash Table for Vehicle Registration Lookup: A hash table is used for vehicle registration lookup because it provides fast access to registration information based on license plate numbers, allowing for quick retrieval of vehicle details.

4. Graph for Road Network Mapping: A graph is used to represent the road network because it can efficiently model the relationships between roads and the paths connecting them, allowing for easy traversal and navigation through the city.

5. Linked List for Parking Slot Availability: A linked list is used to manage parking slot availability because it allows for dynamic insertion and deletion of parking slots as they become available or occupied, providing flexibility in managing the parking system.



Explanation/Justification/Observation (100 words / 5 – 6 sentence) :

The program models a smart city Traffic Control System by integrating multiple data structures to handle different operations efficiently. It uses a queue to manage traffic signals following the FIFO principle and a priority queue (implemented as a list) to handle emergency vehicles. A hash table (dictionary) enables quick vehicle registration lookup. The road network is represented using an undirected graph through an adjacency list. Additionally, a singly linked list manages parking slots dynamically. The system demonstrates effective use of Object-Oriented Programming and data structures to simulate real-world traffic management scenarios in a structured and practical manner.

Tack-08

Final Optimal Prompt:

write a python program for smart E-commerce platform using data structures like stack, queue, priority queue, linked list, binary search tree(BST), hash table, graph and deque to manage shopping cart management, order processing system, top selling products tracker, product search engine, delivery route planning

choose which data structure is best for each task and implement the functionalities accordingly and make a documentation for the program and its functionalities and justify the choice of data structures used for each functionality

---

Code Screenshot :

```
'''
class ECommercePlatform:
    """A class representing a smart e-commerce platform."""
    def __init__(self):
        """
        Initialize the e-commerce platform with required data structures.
        """
        self.shopping_cart_stack = []  # Stack for managing shopping cart items
        self.order_processing_queue = []  # Queue for processing orders
        self.top_selling_products_priority_queue = []  # Priority queue for tracking top-selling products
        self.product_search_bst = None  # Binary Search Tree for product search engine
        self.delivery_route_graph = {}  # Graph for delivery route planning
    def add_to_cart(self, product):
        """
        Add a product to the shopping cart.
        LIFO principle (Last In First Out)
        """
        self.shopping_cart_stack.append(product)
    def process_order(self, order):
        """
        Add an order to the processing queue.
        FIFO principle (First In First Out)
        """
        self.order_processing_queue.append(order)
    def track_top_selling_product(self, product, sales):
        """
        Add a product to the priority queue based on sales.
        Higher sales get higher priority.
        """
        self.top_selling_products_priority_queue.append((sales, product))
        self.top_selling_products_priority_queue.sort(reverse=True)  # Sort by sales in descending order
    def search_product(self, product_name):
        """
        Search for a product using a binary search tree.
        Returns True if found, otherwise False.
        """
        return self._search_bst(self.product_search_bst, product_name)
    def _search_bst(self, node, product_name):
        if node is None:
            return False
        if node.data == product_name:
            return True
        elif product_name < node.data:
            return self._search_bst(node.left, product_name)
        else:
            return self._search_bst(node.right, product_name)
    def plan_delivery_route(self, location1, location2):
        """
        Connect two locations in the delivery route graph.
        Undirected graph representation.
        """
        if location1 not in self.delivery_route_graph:
            self.delivery_route_graph[location1] = []
        if location2 not in self.delivery_route_graph:
            self.delivery_route_graph[location2] = []

        self.delivery_route_graph[location1].append(location2)
        self.delivery_route_graph[location2].append(location1)
class BSTNode:
    """Node class for Binary Search Tree."""
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
if __name__ == "__main__":
    ecommerce_platform = ECommercePlatform()
    # Add products to shopping cart
    ecommerce_platform.add_to_cart("Laptop")
    ecommerce_platform.add_to_cart("Smartphone")
    # Process orders
    ecommerce_platform.process_order("Order 1")
    ecommerce_platform.process_order("Order 2")
    # Track top-selling products
    ecommerce_platform.track_top_selling_product("Laptop", 100)
    ecommerce_platform.track_top_selling_product("Smartphone", 150)
    # Search for a product
    print(ecommerce_platform.search_product("Laptop"))  # Output: False (BST not implemented)
    # Plan delivery routes
    ecommerce_platform.plan_delivery_route("Warehouse", "Customer A")
    ecommerce_platform.plan_delivery_route("Warehouse", "Customer B")
    print("Delivery Route Graph:")
    for location, connections in ecommerce_platform.delivery_route_graph.items():
        print(f"{location}: Connected to {connections}")
```

Output Screenshot:

Justification of Data Structures Used

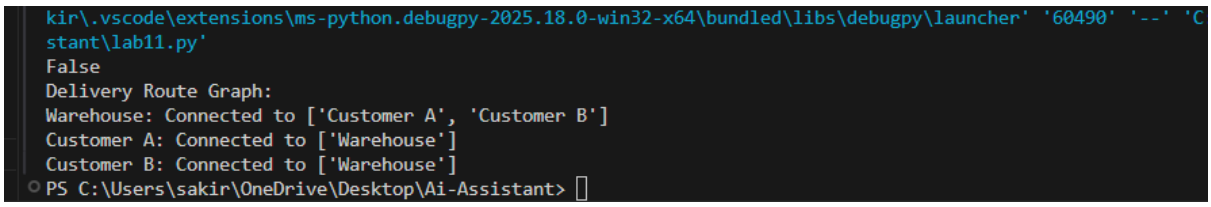
1. Stack for Shopping Cart Management: A stack is used to manage shopping cart items because it allows for easy addition and removal of items in a last-in-first-out manner, which is suitable for a shopping cart where the most recently added item is often the first one to be removed.

2. Queue for Order Processing System: A queue is used for processing orders because it follows the first-in-first-out principle, ensuring that orders are processed in the order they were received, which is important for maintaining fairness and efficiency in order fulfillment.

3. Priority Queue for Top Selling Products Tracker: A priority queue is used to track top-selling products because it allows for efficient retrieval of products based on their sales, ensuring that the most popular products are easily accessible.

4. Binary Search Tree for Product Search Engine: A binary search tree is used for the product search engine because it allows for efficient searching, insertion, and deletion of products based on their names, providing a structured way to manage the product catalog.

5. Graph for Delivery Route Planning: A graph is used to represent the delivery route planning because it can efficiently model the relationships between different locations and the paths connecting them, allowing for easy traversal and optimization of delivery routes.



Explanation/Justification/Observation (100 words / 5 – 6 sentence) :

The program models a smart e-commerce platform by integrating multiple data structures to handle different functionalities. It uses a stack for managing shopping cart items (LIFO), a queue for order processing (FIFO), and a priority queue (implemented using sorting) to track top-selling products based on sales. A Binary Search Tree structure is referenced for product searching, although full insertion logic is not implemented. Additionally, a graph is used for delivery route planning between locations. The design demonstrates strong understanding of data structures and their real-world applications. Overall, the program effectively combines OOP principles with practical system modeling concepts