- **G.Rishika**
- **2303A52197**
- **batch 35**

```
Start coding or generate with AI.
```

```
class Stack:
    """Implements a basic Stack data structure."""

    def __init__(self):
        """Initializes an empty stack."""
        self._items = []

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

    def pop(self):
        """Removes and returns the item from the top of the stack.
        Raises IndexError if the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """Returns the item at the top of the stack without removing it.
        Raises IndexError if the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]

    def is_empty(self):
        """Checks if the stack is empty.
        Returns True if the stack is empty, False otherwise.
        """
        return len(self._items) == 0

    def size(self):
        """Returns the number of items in the stack."""
        return len(self._items)

    def __str__(self):
        """Returns a string representation of the stack."""
        return str(self._items)
```

You can test the `Stack` implementation with the following code:

```
```

```
# Example usage:
my_stack = Stack()

print(f"Is stack empty? {my_stack.is_empty()}")  # Expected: True

my_stack.push(10)
my_stack.push(20)
my_stack.push(30)

print(f"Stack after pushes: {my_stack}") # Expected: [10, 20, 30]
print(f"Stack size: {my_stack.size()}")    # Expected: 3

print(f"Top element: {my_stack.peek()}")   # Expected: 30

popped_item = my_stack.pop()
print(f"Popped item: {popped_item}")      # Expected: 30
print(f"Stack after pop: {my_stack}")     # Expected: [10, 20]

print(f"Is stack empty? {my_stack.is_empty()}")  # Expected: False

my_stack.pop()
my_stack.pop()

print(f"Stack after all pops: {my_stack}") # Expected: []
print(f"Is stack empty? {my_stack.is_empty()}")  # Expected: True

# Attempt to pop from an empty stack (will raise an IndexError)
try:
    my_stack.pop()
except IndexError as e:
    print(f"Error: {e}")
```

```
Is stack empty? True
Stack after pushes: [10, 20, 30]
Stack size: 3
Top element: 30
Popped item: 30
Stack after pop: [10, 20]
Is stack empty? False
Stack after all pops: []
Is stack empty? True
Error: pop from empty stack
```

```
class Queue:
    """Implements a basic Queue data structure (FIFO)."""

    def __init__(self):
        """Initializes an empty queue."""
        self._items = []

    def enqueue(self, item):
        """Adds an item to the rear of the queue.
        (Appends to the end of the list)."""
        self._items.append(item)

    def dequeue(self):
```

```
        """Removes and returns the item from the front of the queue.
        Raises IndexError if the queue is empty.
        """
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.pop(0)

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

    def is_empty(self):
        """Checks if the queue is empty.
        Returns True if the queue is empty, False otherwise.
        """
        return len(self._items) == 0

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

    def __str__(self):
        """Returns a string representation of the queue."""
        return str(self._items)


# Example usage:
my_queue = Queue()

print(f"Is queue empty? {my_queue.is_empty()}")  # Expected: True

my_queue.enqueue(10)
my_queue.enqueue(20)
my_queue.enqueue(30)

print(f"Queue after enqueues: {my_queue}") # Expected: [10, 20, 30]
print(f"Queue size: {my_queue.size()}")    # Expected: 3

print(f"Front element: {my_queue.peek()}")   # Expected: 10

dequeued_item = my_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")      # Expected: 10
print(f"Queue after dequeue: {my_queue}")     # Expected: [20, 30]

print(f"Is queue empty? {my_queue.is_empty()}")  # Expected: False

my_queue.dequeue()
my_queue.dequeue()

print(f"Queue after all dequeues: {my_queue}") # Expected: []
print(f"Is queue empty? {my_queue.is_empty()}")  # Expected: True
```

```
# Attempt to dequeue from an empty queue (will raise an IndexError)
try:
    my_queue.dequeue()
except IndexError as e:
    print(f"Error: {e}")
```

```
Is queue empty? True
Queue after enqueues: [10, 20, 30]
Queue size: 3
Front element: 10
Dequeued item: 10
Queue after dequeue: [20, 30]
Is queue empty? False
Queue after all dequeues: []
Is queue empty? True
Error: dequeue from empty queue
```

```
class Node:
    """Represents a node in a singly linked list."""

    def __init__(self, data=None):
        """Initializes a new node."

        Args:
            data: The data to be stored in the node.
        """
        self.data = data
        self.next = None  # Pointer to the next node

class LinkedList:
    """Implements a singly linked list with insert and display methods."""

    def __init__(self):
        """Initializes an empty linked list."""
        self.head = None  # The head of the list, initially None

    def insert_at_beginning(self, data):
        """Inserts a new node with the given data at the beginning of the list."

        Args:
            data: The data for the new node.
        """
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def display(self):
        """Prints all the elements in the linked list from head to tail."""
        current = self.head
        elements = []
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" -> ".join(elements))
```

```
    def is_empty(self):
        """Checks if the linked list is empty.
        Returns True if the list is empty, False otherwise.
        """
        return self.head is None

    def size(self):
        """Returns the number of nodes in the linked list."

        Returns:
            int: The number of nodes in the list.
        """
        count = 0
        current = self.head
        while current:
            count += 1
            current = current.next
        return count

# Example Usage:
my_list = LinkedList()
print("Initial list:", end=" ")
my_list.display() # Expected: (empty line or just a space)
print(f"Is list empty? {my_list.is_empty()}") # Expected: True

my_list.insert_at_beginning(30)
my_list.insert_at_beginning(20)
my_list.insert_at_beginning(10)

print("List after insertions:", end=" ")
my_list.display() # Expected: 10 -> 20 -> 30
print(f"List size: {my_list.size()}") # Expected: 3
print(f"Is list empty? {my_list.is_empty()}") # Expected: False

my_list.insert_at_beginning(5)
print("List after another insertion:", end=" ")
my_list.display() # Expected: 5 -> 10 -> 20 -> 30
print(f"List size: {my_list.size()}") # Expected: 4
```

```
Initial list:
Is list empty? True
List after insertions: 10 -> 20 -> 30
List size: 3
Is list empty? False
List after another insertion: 5 -> 10 -> 20 -> 30
List size: 4
```

```
class HashTable:
    """Implements a hash table with insert, search, and delete methods, using chaining for collision handling."""

    def __init__(self, capacity=10):
        """Initializes the hash table with a given capacity.

        Args:
            capacity (int): The initial number of buckets in the hash table.
```

```
        """
        self._capacity = capacity
        # Each bucket is a list to handle collisions (chaining)
        self._table = [[] for _ in range(self._capacity)]

    def _hash_function(self, key):
        """Computes the hash value for a given key.

        Args:
            key: The key to hash.

        Returns:
            int: The index in the hash table where the key should be stored.
        """
        return hash(key) % self._capacity

    def insert(self, key, value):
        """Inserts a key-value pair into the hash table.
        If the key already exists, its value is updated.

        Args:
            key: The key to insert.
            value: The value associated with the key.
        """
        index = self._hash_function(key)
        bucket = self._table[index]

        # Check if key already exists in the bucket; if so, update its value
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value) # Update the existing key-value pair
                return
        # If key does not exist, add the new key-value pair to the bucket
        bucket.append((key, value))
        print(f"Inserted: ({key}, {value}) at index {index}")

    def search(self, key):
        """Searches for a key in the hash table.

        Args:
            key: The key to search for.

        Returns:
            Any: The value associated with the key if found, None otherwise.
        """
        index = self._hash_function(key)
        bucket = self._table[index]

        # Iterate through the bucket to find the key
        for k, v in bucket:
            if k == key:
                return v # Key found, return its value
        return None # Key not found

    def delete(self, key):
        """Deletes a key-value pair from the hash table.
```

```
        Args:
            key: The key to delete.

        Returns:
            bool: True if the key was found and deleted, False otherwise.
        """
        index = self._hash_function(key)
        bucket = self._table[index]

        # Iterate through the bucket to find and remove the key
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i] # Remove the key-value pair
                print(f"Deleted: {key} from index {index}")
                return True
        print(f"Key not found for deletion: {key}")
        return False # Key not found

    def display(self):
        """Prints the contents of the hash table."""
        print("\n--- Hash Table Contents ---")
        for i, bucket in enumerate(self._table):
            print(f"Bucket {i}: {bucket}")
        print("---------------------------")

# Example Usage:
hash_table = HashTable(capacity=5)

hash_table.insert("apple", 10)
hash_table.insert("banana", 20)
hash_table.insert("cherry", 30)
hash_table.insert("date", 40) # Might cause collision depending on hash and capacity
hash_table.insert("elderberry", 50)
hash_table.insert("fig", 60) # Likely to cause collision

hash_table.display()

print(f"\nSearching for 'banana': {hash_table.search('banana')}") # Expected: 20
print(f"Searching for 'grape': {hash_table.search('grape')}")   # Expected: None

hash_table.insert("apple", 15) # Update existing key
hash_table.display()

hash_table.delete("cherry")
hash_table.delete("grape") # Key not found
hash_table.display()

print(f"Searching for 'cherry' after deletion: {hash_table.search('cherry')}") # Expected: None
```

```
Inserted: (apple, 10) at index 0
Inserted: (banana, 20) at index 4
Inserted: (cherry, 30) at index 2
Inserted: (date, 40) at index 2
Inserted: (elderberry, 50) at index 1
```

```
Inserted: (fig, 60) at index 2

--- Hash Table Contents ---
Bucket 0: [('apple', 10)]
Bucket 1: [('elderberry', 50)]
Bucket 2: [('cherry', 30), ('date', 40), ('fig', 60)]
Bucket 3: []
Bucket 4: [('banana', 20)]
---------------------------

Searching for 'banana': 20
Searching for 'grape': None

--- Hash Table Contents ---
Bucket 0: [('apple', 15)]
Bucket 1: [('elderberry', 50)]
Bucket 2: [('cherry', 30), ('date', 40), ('fig', 60)]
Bucket 3: []
Bucket 4: [('banana', 20)]
---------------------------
Deleted: cherry from index 2
Key not found for deletion: grape

--- Hash Table Contents ---
Bucket 0: [('apple', 15)]
Bucket 1: [('elderberry', 50)]
Bucket 2: [('date', 40), ('fig', 60)]
Bucket 3: []
Bucket 4: [('banana', 20)]
---------------------------
Searching for 'cherry' after deletion: None
```

```
class Graph:
    """Implements a graph using an adjacency list representation."""

    def __init__(self):
        """Initializes an empty graph."
        The graph is represented using a dictionary where keys are vertices
        and values are lists of adjacent vertices (neighbors).
        """
        self.graph = {}

    def add_vertex(self, vertex):
        """Adds a vertex to the graph if it doesn't already exist."

        Args:
            vertex: The vertex to be added.
        """
        if vertex not in self.graph:
            self.graph[vertex] = []
            print(f"Added vertex: {vertex}")
        else:
            print(f"Vertex {vertex} already exists.")

    def add_edge(self, vertex1, vertex2):
        """Adds an undirected edge between two vertices."
        If the vertices do not exist, they are added to the graph first.
```

```
        Args:
            vertex1: The first vertex of the edge.
            vertex2: The second vertex of the edge.
        """
        # Ensure both vertices exist in the graph
        self.add_vertex(vertex1)
        self.add_vertex(vertex2)

        # Add edge from vertex1 to vertex2 (if not already present)
        if vertex2 not in self.graph[vertex1]:
            self.graph[vertex1].append(vertex2)
            print(f"Added edge: {vertex1} -- {vertex2}")
        else:
            print(f"Edge {vertex1} -- {vertex2} already exists.")

        # Add edge from vertex2 to vertex1 (since it's an undirected graph)
        if vertex1 not in self.graph[vertex2]:
            self.graph[vertex2].append(vertex1)

    def display(self):
        """Prints the adjacency list representation of the graph."""
        print("\n--- Graph Adjacency List ---")
        if not self.graph:
            print("Graph is empty.")
            return
        for vertex, neighbors in self.graph.items():
            print(f"{vertex}: {neighbors}")
        print("----------------------------")

# Example Usage:
my_graph = Graph()

print("Graph initialization:")
my_graph.display() # Expected: Graph is empty.

print("\nAdding vertices:")
my_graph.add_vertex('A')
my_graph.add_vertex('B')
my_graph.add_vertex('C')
my_graph.add_vertex('A') # Attempt to add existing vertex

print("\nAdding edges:")
my_graph.add_edge('A', 'B')
my_graph.add_edge('B', 'C')
my_graph.add_edge('C', 'A')
my_graph.add_edge('D', 'A') # 'D' will be added automatically
my_graph.add_edge('A', 'B') # Attempt to add existing edge

my_graph.display()

# Another example
print("\nCreating another graph:")
new_graph = Graph()
new_graph.add_edge(1, 2)
new_graph.add_edge(2, 3)
new_graph.add_edge(3, 4)
```

```
new_graph.add_edge(4, 1)
new_graph.display()
```

```
Graph initialization:

--- Graph Adjacency List ---
Graph is empty.

Adding vertices:
Added vertex: A
Added vertex: B
Added vertex: C
Vertex A already exists.

Adding edges:
Vertex A already exists.
Vertex B already exists.
Added edge: A -- B
Vertex B already exists.
Vertex C already exists.
Added edge: B -- C
Vertex C already exists.
Vertex A already exists.
Added edge: C -- A
Added vertex: D
Vertex A already exists.
Added edge: D -- A
Vertex A already exists.
Vertex B already exists.
Edge A -- B already exists.

--- Graph Adjacency List ---
A: ['B', 'C', 'D']
B: ['A', 'C']
C: ['B', 'A']
D: ['A']
----------------------------

Creating another graph:
Added vertex: 1
Added vertex: 2
Added edge: 1 -- 2
Vertex 2 already exists.
Added vertex: 3
Added edge: 2 -- 3
Vertex 3 already exists.
Added vertex: 4
Added edge: 3 -- 4
Vertex 4 already exists.
Vertex 1 already exists.
Added edge: 4 -- 1

--- Graph Adjacency List ---
1: [2, 4]
2: [1, 3]
3: [2, 4]
4: [3, 1]
----------------------------
```

```
Start coding or generate with AI.
```

Smart City Traffic Control System: Data Structure Selection and Implementation

Data Structure Choices for Smart Traffic Management System Features

Smart Traffic Management Feature	Selected Data Structure	Justification
Traffic Signal Queue	Queue	A Queue is ideal for managing vehicles waiting at a traffic signal as it operates on a First-In, First-Out (FIFO) principle. This ensures that vehicles are processed in the order the
Emergency Vehicle Priority Handling	Priority Queue	A Priority Queue allows emergency vehicles to be processed before regular traffic, even if they arrive later, based on their priority level. This ensures critical services can naviga
Vehicle Registration Lookup	Hash Table	A Hash Table provides extremely fast average-case O(1) time complexity for looking up vehicle information using a unique identifier like a license plate or VIN. This enables quic
Road Network Mapping	Graph	A Graph naturally represents a road network, where intersections are nodes/vertices and roads are edges, potentially with weights for distance or travel time. This structure is e
Parking Slot Availability	Hash Table	A Hash Table can efficiently store and retrieve the status (available/occupied) of parking slots based on their unique IDs. This allows for quick updates when a slot becomes occ

Implementation of Priority Queue for Emergency Vehicle Priority Handling

This Python program implements a `PriorityQueue` class using Python's `heapq` module to represent the 'Emergency Vehicle Priority Handling' feature. In this system, patients with lower priority values are considered more critical and are processed first. The example usage simulates inserting patients with different criticality levels and then processing them in order of priority.

```
import heapq

class PriorityQueue:
    """Implements a Priority Queue using a min-heap for efficient retrieval of the highest priority item."""

    def __init__(self):
        """Initializes an empty Priority Queue."""
        self._queue = []  # Stores (priority, item) tuples
        self._index = 0   # Used to break ties for items with the same priority (ensures stability)

    def insert(self, item, priority):
        """Inserts an item into the priority queue with a given priority.
        Lower priority value indicates higher priority.

        Args:
            item: The item to be inserted.
            priority (int/float): The priority of the item.
        """
        # Use _index to ensure that items with the same priority are treated FIFO
        heapq.heappush(self._queue, (priority, self._index, item))
        self._index += 1
        print(f"Inserted: '{item}' with priority {priority}")

    def extract_min(self):
        """Removes and returns the item with the highest priority (lowest priority value).
        Raises IndexError if the queue is empty.

        Returns:
            Any: The item with the highest priority.
        """
```

```
            if self.is_empty():
                raise IndexError("extract_min from empty priority queue")
            priority, _, item = heapq.heappop(self._queue)
            print(f"Extracted: '{item}' with priority {priority}")
            return item

        def peek(self):
            """Returns the item with the highest priority (lowest priority value) without removing it.
            Raises IndexError if the queue is empty.

            Returns:
                Any: The item with the highest priority.
            """
            if self.is_empty():
                raise IndexError("peek from empty priority queue")
            priority, _, item = self._queue[0]
            return item

        def is_empty(self):
            """Checks if the priority queue is empty.

            Returns:
                bool: True if the queue is empty, False otherwise.
            """
            return len(self._queue) == 0

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

        def __str__(self):
            """Returns a string representation of the priority queue."""
            # For display, sort by priority and then by insertion order
            sorted_items = sorted(self._queue)
            return str([(priority, item) for priority, _, item in sorted_items])

    # Example Usage for Emergency Case Handling:
    print("--- Priority Queue for Emergency Cases ---")
    emergency_queue = PriorityQueue()

    print(f"Is queue empty? {emergency_queue.is_empty()}") # Expected: True

    emergency_queue.insert("Patient C (Minor cut)", 3) # Lower priority value means more critical
    emergency_queue.insert("Patient A (Heart attack)", 1)
    emergency_queue.insert("Patient B (Broken arm)", 2)
    emergency_queue.insert("Patient D (Fever)", 4)
    emergency_queue.insert("Patient E (Chest pain)", 1) # Same priority as A

    print(f"\nCurrent queue: {emergency_queue}")
    print(f"Queue size: {emergency_queue.size()}")
    print(f"Next to be treated (peek): {emergency_queue.peek()}") # Expected: Patient A or E (priority 1)

    print("\nProcessing patients:")
    while not emergency_queue.is_empty():
        patient = emergency_queue.extract_min()
        print(f"Treating: {patient}")
```

```
print(f"\nIs queue empty after processing? {emergency_queue.is_empty()}") # Expected: True

# Attempt to peek/extract from an empty queue
try:
    emergency_queue.peek()
except IndexError as e:
    print(f"Error: {e}")
try:
    emergency_queue.extract_min()
except IndexError as e:
    print(f"Error: {e}")
```

```
--- Priority Queue for Emergency Cases ---
Is queue empty? True
Inserted: 'Patient C (Minor cut)' with priority 3
Inserted: 'Patient A (Heart attack)' with priority 1
Inserted: 'Patient B (Broken arm)' with priority 2
Inserted: 'Patient D (Fever)' with priority 4
Inserted: 'Patient E (Chest pain)' with priority 1

Current queue: [(1, 'Patient A (Heart attack)'), (1, 'Patient E (Chest pain)'), (2, 'Patient B (Broken arm)'), (3, 'Patient C (Minor cut)'), (4, 'Patient D (Fever)')]
Queue size: 5
Next to be treated (peek): Patient A (Heart attack)

Processing patients:
Extracted: 'Patient A (Heart attack)' with priority 1
Treating: Patient A (Heart attack)
Extracted: 'Patient E (Chest pain)' with priority 1
Treating: Patient E (Chest pain)
Extracted: 'Patient B (Broken arm)' with priority 2
Treating: Patient B (Broken arm)
Extracted: 'Patient C (Minor cut)' with priority 3
Treating: Patient C (Minor cut)
Extracted: 'Patient D (Fever)' with priority 4
Treating: Patient D (Fever)

Is queue empty after processing? True
Error: peek from empty priority queue
Error: extract_min from empty priority queue
```

Task

```
class Queue:
    """Implements a basic Queue data structure (FIFO)."""

    def __init__(self):
        """Initializes an empty queue."""
        self._items = []

    def enqueue(self, item):
        """Adds an item to the rear of the queue.
        (Appends to the end of the list)."""
```

```
        self._items.append(item)

    def dequeue(self):
        """Removes and returns the item from the front of the queue.
        Raises IndexError if the queue is empty.
        """
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.pop(0)

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

    def is_empty(self):
        """Checks if the queue is empty.
        Returns True if the queue is empty, False otherwise.
        """
        return len(self._items) == 0

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

    def __str__(self):
        """Returns a string representation of the queue."""
        return str(self._items)


# Example usage for 'Order Processing System':
print("--- Order Processing System Simulation ---")
order_queue = Queue()

print(f"Is the order queue empty? {order_queue.is_empty()}")  # Expected: True

# Enqueue some orders
print("\nEnqueuing orders...")
order_queue.enqueue("Order #101 - Laptop")
order_queue.enqueue("Order #102 - Smartphone")
order_queue.enqueue("Order #103 - Headphones")

print(f"Orders in queue: {order_queue}") # Expected: ['Order #101 - Laptop', 'Order #102 - Smartphone', 'Order #103 - Headphones'
print(f"Number of pending orders: {order_queue.size()}")    # Expected: 3
```

```
# Peek at the next order to be processed
print(f"Next order to process (peek): {order_queue.peek()}")   # Expected: Order #101 - Laptop

# Dequeue orders as they are processed
print("\nProcessing orders...")
processed_order_1 = order_queue.dequeue()
print(f"Processed order: {processed_order_1}")      # Expected: Order #101 - Laptop
print(f"Orders in queue after processing: {order_queue}")     # Expected: ['Order #102 - Smartphone', 'Order #103 - Headphones']

processed_order_2 = order_queue.dequeue()
print(f"Processed order: {processed_order_2}")      # Expected: Order #102 - Smartphone
print(f"Orders in queue after processing: {order_queue}")     # Expected: ['Order #103 - Headphones']

print(f"Is the order queue empty? {order_queue.is_empty()}")  # Expected: False

# Process the last order
order_queue.dequeue()
print(f"Orders in queue after all processing: {order_queue}") # Expected: []
print(f"Is the order queue empty? {order_queue.is_empty()}")  # Expected: True

# Attempt to dequeue from an empty queue (will raise an IndexError)
print("\nAttempting to dequeue from an empty queue:")
try:
    order_queue.dequeue()
except IndexError as e:
    print(f"Error: {e}")

# Attempt to peek from an empty queue
print("\nAttempting to peek from an empty queue:")
try:
    order_queue.peek()
except IndexError as e:
    print(f"Error: {e}")
```

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t

E-Commerce Feature	Selected Data Structure	Justification
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue,

Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Data Structure Selection and Justification for E-Commerce Features

Subtask:

Analyze each e-commerce system feature (Shopping Cart Management, Order Processing System, Top-Selling Products Tracker, Product Search Engine, Delivery Route Planning), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for E-Commerce System Features

E-Commerce Feature	Selected Data Structure	Justification
Shopping Cart Management	Hash Table	A Hash Table allows for O(1) average-case time complexity for adding, removing, updating quantities, and checking if an item exists by its ID. This provides efficient and responsive man
Order Processing System	Queue	A Queue operates on a First-In, First-Out (FIFO) principle, which perfectly matches the requirement for processing orders in the sequence they are received. New orders are added to t
Top-Selling Products Tracker	Priority Queue	A Max-Heap based Priority Queue efficiently stores products, prioritizing them by sales volume. It allows for quick retrieval of the current top-selling product (O(1)) and efficient update
Product Search Engine	Hash Table	A Hash Table (often used within an inverted index) provides extremely fast average-case time complexity (O(1)) for retrieving products based on keywords or product IDs. This is critica
Delivery Route Planning	Graph	A Graph naturally represents delivery locations as nodes and roads/routes as edges, with edge weights indicating distance or time. Graph algorithms like Dijkstra's or A* can then be ap

Implement Queue for Order Processing

Subtask:

Implement a `Queue` class in Python with essential methods (e.g., `enqueue`, `dequeue`, `peek`, `is_empty`) and example usage, representing the 'Order Processing System' feature.

Summary:

Data Analysis Key Findings

- **Shopping Cart Management**: A **Hash Table** was chosen. This data structure provides average-case O(1) time complexity for adding, removing, updating quantities, and checking item existence, ensuring efficient and responsive cart management.
- **Order Processing System**: A **Queue** was selected. Its First-In, First-Out (FIFO) principle perfectly matches the requirement for processing orders in the sequence they are received.
- **Top-Selling Products Tracker**: A **Priority Queue** (specifically, a Max-Heap) was identified. This allows for quick O(1) retrieval of the top-selling product and efficient O(log n) updates or insertions when sales figures change.
- **Product Search Engine**: A **Hash Table** (often used within an inverted index) was chosen. This provides extremely fast average-case O(1) time complexity for retrieving products by keywords or IDs, which is critical for rapid search results.
- **Delivery Route Planning**: A **Graph** was deemed most appropriate. It naturally represents delivery locations as nodes and routes as edges, enabling the use of graph algorithms like Dijkstra's or A* for optimal pathfinding.

Insights or Next Steps

- The strategic selection of data structures directly impacts the performance and efficiency of key e-commerce features, with choices often driven by required operational speed (e.g., O(1) for frequent lookups) and data access patterns (e.g., FIFO for order processing).
- The next step could involve implementing these e-commerce features using the identified data structures to build functional prototypes and validate the performance benefits in a real-world context.

Task

Data Structure Selection and Justification for Smart City Features

Subtask:

Analyze each Smart Traffic Management System feature (Traffic Signal Queue, Emergency Vehicle Priority Handling, Vehicle Registration Lookup, Road Network Mapping, Parking Slot Availability), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for Smart Traffic Management System Features

Smart City Feature	Selected Data Structure	Justification
Traffic Signal Queue	Queue	Vehicles at a traffic light are typically processed in the order they arrive, embodying the First-In, First-Out (FIFO) principle. A Queue efficiently manages this sequence, allowing
Emergency Vehicle Priority Handling	Priority Queue	Emergency vehicles require immediate processing over regular traffic. A Priority Queue ensures that elements (vehicles) with higher priority are dequeued first, enabling swift ar
Vehicle Registration Lookup	Hash Table	To quickly retrieve vehicle information using a unique registration number, a Hash Table offers average O(1) time complexity for search, insertion, and deletion operations. This a
Road Network Mapping	Graph	A city's road network naturally forms a graph, where intersections are nodes/vertices and roads are edges. This structure is ideal for representing connectivity, calculating distar

Smart City Feature	Selected Data Structure	Justification
Parking Slot Availability	Hash Table	Efficiently checking and updating the availability of parking slots requires rapid access by slot ID. A Hash Table provides O(1) average-case time complexity for lookups and mod

Implement Priority Queue for Emergency Vehicle Priority Handling

Subtask:

Implement a `PriorityQueue` class in Python with essential methods (e.g., `insert`, `extract_min` or `extract_max`, `peek`, `is_empty`) and example usage, representing the 'Emergency Vehicle Priority Handling' feature.

Data Structure Selection and Justification for Smart City Features

Subtask:

Analyze each Smart Traffic Management System feature (Traffic Signal Queue, Emergency Vehicle Priority Handling, Vehicle Registration Lookup, Road Network Mapping, Parking Slot Availability), select the most appropriate data structure from the given list (Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST), Graph, Hash Table, Deque), and provide a 2-3 sentence justification for each choice.

Data Structure Choices for Smart Traffic Management System Features

Smart Traffic Management Feature	Selected Data Structure	Justification
Traffic Signal Queue	Queue	A Queue is ideal for managing vehicles waiting at a traffic signal as it operates on a First-In, First-Out (FIFO) principle. This ensures that vehicles are processed in the order the
Emergency Vehicle Priority Handling	Priority Queue	A Priority Queue allows emergency vehicles to be processed before regular traffic, even if they arrive later, based on their priority level. This ensures critical services can naviga
Vehicle Registration Lookup	Hash Table	A Hash Table provides extremely fast average-case O(1) time complexity for looking up vehicle information using a unique identifier like a license plate or VIN. This enables quic
Road Network Mapping	Graph	A Graph naturally represents a road network, where intersections are nodes/vertices and roads are edges, potentially with weights for distance or travel time. This structure is e
Parking Slot Availability	Hash Table	A Hash Table can efficiently store and retrieve the status (available/occupied) of parking slots based on their unique IDs. This allows for quick updates when a slot becomes occ

Implement Priority Queue for Emergency Vehicle Priority Handling

Subtask:

Implement a `PriorityQueue` class in Python with essential methods (e.g., `insert`, `extract_min` or `extract_max`, `peek`, `is_empty`) and example usage, representing the 'Emergency Vehicle Priority Handling' feature.

Summary:

Q&A

The task was to analyze each Smart Traffic Management System feature, select the most appropriate data structure from a given list, and provide a 2-3 sentence justification for each choice.

Here are the selected data structures and their justifications:

- **Traffic Signal Queue**: A Queue was chosen because it adheres to the First-In, First-Out (FIFO) principle, which is essential for processing vehicles in the order they arrive at a traffic signal.
- **Emergency Vehicle Priority Handling**: A Priority Queue was selected to ensure that emergency vehicles, with their higher priority, are processed ahead of regular traffic, facilitating quicker response times.
- **Vehicle Registration Lookup**: A Hash Table was chosen for its average O(1) time complexity, enabling very fast lookups of vehicle information using unique identifiers.
- **Road Network Mapping**: A Graph was deemed most suitable as it naturally models a city's road network, representing intersections as nodes and roads as edges, which is vital for route optimization and traffic analysis.
- **Parking Slot Availability**: A Hash Table was chosen to efficiently manage and retrieve parking slot status (available/occupied) due to its O(1) average-case time complexity for lookups and updates.

Data Analysis Key Findings

- The analysis successfully identified the most suitable data structures for five key Smart Traffic Management System features.
- **Queue** was selected for "Traffic Signal Queue" to manage vehicles on a FIFO basis, ensuring fair processing based on arrival time.
- **Priority Queue** was assigned to "Emergency Vehicle Priority Handling" to prioritize critical services, allowing them to bypass regular traffic.
- **Hash Table** was chosen for "Vehicle Registration Lookup" and "Parking Slot Availability" due to its O(1) average-case time complexity, enabling rapid data retrieval and status updates critical for real-time systems.
- **Graph** was identified as the ideal structure for "Road Network Mapping" given its ability to model complex interconnections between intersections and roads, which is fundamental for navigation and traffic flow analysis.

Insights or Next Steps

- The strategic selection of data structures directly impacts the efficiency, responsiveness, and scalability of smart city applications, ensuring optimal performance for critical services.
- The next step involves implementing the chosen `PriorityQueue` class, which will be crucial for the "Emergency Vehicle Priority Handling" feature, requiring methods like `insert`, `extract_min`, `peek`, and `is_empty`.