

Assignment-11.5

Name: R.sai kundan

Ht.no: 2303A51413

Batch. No: 21

Task Description #1 – Stack Implementation

Prompt:

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

Code:

```
lab04.py    assignmnet5.5.py    assignment08.py    17-02-2026.py    ass9.2.py    assignment11.py X :  
assignment11.py > ...  
1  # Create a Python Stack class using a list. I  
2  # include methods: push, pop, peek and is_empty. Add proper docstrings and it should handle edge cases  
3  # and errors.  
4  # Take input from user to demonstrate the functionality of the Stack class.  
5  class Stack:  
6      """A simple Stack implementation using a list."""  
7  
8      def __init__(self):  
9          """Initialize an empty stack."""  
10         self.stack = []  
11  
12     def push(self, item):  
13         """Add an item to the top of the stack."""  
14         self.stack.append(item)  
15  
16     def pop(self):  
17         """Remove and return the top item from the stack."""  
18         if self.is_empty():  
19             raise IndexError("pop from empty stack")  
20         return self.stack.pop()  
21  
22     def peek(self):  
23         """Return the top item from the stack without removing it."""  
24         if self.is_empty():  
25             raise IndexError("peek from empty stack")  
26         return self.stack[-1]  
27  
28     def is_empty(self):  
29         """Return True if the stack is empty, otherwise False."""  
30         return len(self.stack) == 0  
31  
# Demonstrate the functionality of the Stack class  
32 if __name__ == "__main__":  
33     stack = Stack()  
34     while True:  
35         action = input("Enter an action (push, pop, peek, is_empty, quit): ").strip().lower()  
36         if action == "push":  
            item = input("Enter the item to push: ")  
            stack.push(item)  
            print(f"Stack after push: {stack.stack}")  
        elif action == "pop":  
            try:  
                item = stack.pop()  
                print(f"Popped item: {item}")  
            except IndexError:  
                print("Stack is empty. Cannot pop.")  
        elif action == "peek":  
            try:  
                item = stack.peek()  
                print(f"Peeked item: {item}")  
            except IndexError:  
                print("Stack is empty. Cannot peek.")  
        elif action == "is_empty":  
            print(f"Stack is empty: {stack.is_empty()}" )  
        elif action == "quit":  
            break  
        else:  
            print("Unknown action. Try again.")
```

Run ... assistantcoding - Copy

lab04.py assignmnet5.5.py assignment08.py 17-02-2026.py ass9.2.py assignment11.py X

assignment11.py > ...

```
31 # Demonstrate the functionality of the Stack class
32 if __name__ == "__main__":
33     stack = Stack()
34     while True:
35         action = input("Enter an action (push, pop, peek, is_empty, quit): ").strip().lower()
36         if action == "push":
37             item = input("Enter an item to push onto the stack: ")
38             stack.push(item)
39             print(f"'{item}' has been pushed onto the stack.")
40         elif action == "pop":
41             try:
42                 item = stack.pop()
43                 print(f"'{item}' has been popped from the stack.")
44             except IndexError as e:
45                 print(e)
46         elif action == "peek":
47             try:
48                 item = stack.peek()
49                 print(f"The top item is: '{item}' .")
50             except IndexError as e:
51                 print(e)
52         elif action == "is_empty":
53             if stack.is_empty():
54                 print("The stack is empty.")
55             else:
56                 print("The stack is not empty.")
57         elif action == "quit":
58             print("Exiting the program.")
59             break
60         else:
61             print("Invalid action. Please try again.)")
62
```

Ln 62, Col 13 Spaces: 4 UTF-8



Output:

```
Enter an action (push, pop, peek, is_empty, quit): push
Enter an item to push onto the stack: a
'a' has been pushed onto the stack.
Enter an action (push, pop, peek, is_empty, quit): push
Enter an item to push onto the stack: b
'b' has been pushed onto the stack.
Enter an action (push, pop, peek, is_empty, quit): pop
'b' has been popped from the stack.
Enter an action (push, pop, peek, is_empty, quit): peek
The top item is: 'a'.
Enter an action (push, pop, peek, is_empty, quit): is_empty
The stack is not empty.
Enter an action (push, pop, peek, is_empty, quit): quit
Exiting the program.
```

Observation:

The Stack class works correctly by following the Last In, First Out (LIFO) principle, where the most recently pushed element is removed first. The push, pop, peek, and is_empty methods perform as expected, including proper handling of edge cases like popping or peeking from an empty stack. The interactive user input successfully demonstrates the functionality and error handling of the stack implementation.

Task Description #2 – Queue Implementation

Prompt:

Create a Python Queue class using a list. Implement enqueue, dequeue, peek, and size methods. Follow FIFO principle. Add proper docstrings and handle empty queue errors. Take input from user to demonstrate the functionality of the Queue class.

Code:

```
assignment11.py > ...

# Create a Python Queue class using a list.
# Implement enqueue, dequeue, peek, and size methods.
# Follow FIFO principle. Add proper docstrings and
# handle empty queue errors. Take input from asyncio
class Queue:
    """A simple Queue implementation using a list."""

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

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

    def dequeue(self):
        """Remove and return the front item from the queue."""
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.queue.pop(0)

    def peek(self):
        """Return the front item from the queue without removing it."""
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.queue[0]

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

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

# Demonstrate the functionality of the Queue class
if __name__ == "__main__":
    pass
```

```
004.py  assignmnet5.5.py  assignment08.py  17-02-2026.py  ass9.2.py  assignment11.py X
assignment11.py > ...
    return len(self.queue) == 0
# Demonstrate the functionality of the Queue class
if __name__ == "__main__":
    queue = Queue()
    while True:
        action = input("Enter an action (enqueue, dequeue, peek, size, is_empty, quit): ").strip().lower()
        if action == "enqueue":
            item = input("Enter an item to enqueue: ")
            queue.enqueue(item)
            print(f"'{item}' has been enqueued.")
        elif action == "dequeue":
            try:
                item = queue.dequeue()
                print(f"'{item}' has been dequeued.")
            except IndexError as e:
                print(e)
        elif action == "peek":
            try:
                item = queue.peek()
                print(f"The front item is: '{item}'")
            except IndexError as e:
                print(e)
        elif action == "size":
            print(f"The size of the queue is: {queue.size()}")
        elif action == "is_empty":
            if queue.is_empty():
                print("The queue is empty.")
            else:
                print("The queue is not empty.")
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")
```

Output:

```
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): enqueue
Enter an item to enqueue: 2
'2' has been enqueued.
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): dequeue
'2' has been dequeued.
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): 2
Invalid action. Please try again.
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): peek
peek from empty queue
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): size
The size of the queue is: 0.
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): is_empty
The queue is empty.
Enter an action (enqueue, dequeue, peek, size, is_empty, quit): quit
Exiting the program.
```

Observation:

The Queue class correctly follows the First In, First Out (FIFO) principle, ensuring that elements are removed in the same order they were added. All methods including enqueue, dequeue, peek, and size function properly and handle edge cases like empty queue operations. The implementation effectively demonstrates queue behavior using Python lists.

Task Description #3 – Linked List

Prompt:

Create a Python implementation of a Singly Linked List. Define a Node class and a LinkedList class.

Include methods: insert(data) to add at the end and display() to print all elements. Add proper docstrings and basic error handling. Take input from user to demonstrate the functionality of the LinkedList class.

Code:

```
Create a Python implementation of a Singly Linked List.  
Define a Node class and a LinkedList class.  
Include methods: insert(data) to add at the end and display() to print all elements.  
Add proper docstrings and basic error handling.  
Take input from user to demonstrate the functionality of the LinkedList class.  
  
class Node:  
    """A node in a singly linked list."  
  
    def __init__(self, data):  
        """Initialize a node with data and a pointer to the next node."  
        self.data = data  
        self.next = None  
class 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 list."  
        new_node = Node(data)  
        if not self.head:  
            self.head = new_node  
            return  
        last_node = self.head  
        while last_node.next:  
            last_node = last_node.next  
        last_node.next = new_node  
  
    def display(self):  
        """Print all elements in the linked list."  
        if not self.head:  
            print("The linked list is empty.")  
        return
```

```
169     current_node = self.head
170     while current_node:
171         print(current_node.data, end=" -> ")
172         current_node = current_node.next
173     print("None")
174 # Demonstrate the functionality of the LinkedList class
175 if __name__ == "__main__":
176     linked_list = LinkedList()
177     while True:
178         action = input("Enter an action (insert, display, quit): ").strip().lower()
179         if action == "insert":
180             data = input("Enter data to insert into the linked list: ")
181             linked_list.insert(data)
182             print(f"'{data}' has been inserted into the linked list.")
183         elif action == "display":
184             print("The elements in the linked list are:")
185             linked_list.display()
186         elif action == "quit":
187             print("Exiting the program.")
188             break
189         else:
190             print("Invalid action. Please try again.")
191
```

Output:

```
Enter an action (insert, display, quit): display
The elements in the linked list are:
10 -> None
Enter an action (insert, display, quit): quit
Exiting the program.
```

Observation:

The Singly Linked List correctly stores elements in sequential order using node connections. The insert method successfully adds elements at the end of the list, and the display method prints all nodes clearly. The implementation also handles the empty list case properly without errors.

Task Description #4 – Hash Table

Prompt:

Create a Python HashTable class. Implement insert, search, and delete methods. Add proper docstrings and basic error handling. Take input from user to demonstrate the functionality of the HashTable class.

Code:

```
assignment11.py > ...
193  # Create a Python HashTable class. Implement insert, search, and delete methods.
194  # Add proper docstrings and basic error handling.
195  # Take input from user to demonstrate the functionality of the HashTable class.
196  # Code:
197  class HashTable:
198      """A simple implementation of a hash table using chaining for collision resolution."""
199
200      def __init__(self, size=10):
201          """Initialize the hash table with a specified size."""
202          self.size = size
203          self.table = [[] for _ in range(size)]
204
205      def _hash(self, key):
206          """Generate a hash for the given key."""
207          return hash(key) % self.size
208
209      def insert(self, key, value):
210          """Insert a key-value pair into the hash table."""
211          index = self._hash(key)
212          for i, (k, v) in enumerate(self.table[index]):
213              if k == key:
214                  self.table[index][i] = (key, value) # Update existing key
215                  return
216          self.table[index].append((key, value)) # Insert new key-value pair
217
218      def search(self, key):
219          """Search for a value by its key in the hash table."""
220          index = self._hash(key)
221          for k, v in self.table[index]:
222              if k == key:
223                  return v # Return the value if the key is found
224          raise KeyError(f"Key '{key}' not found in the hash table.")
225
226      def delete(self, key):
227          """Delete a key-value pair from the hash table."""
228          index = self._hash(key)
229          for i, (k, v) in enumerate(self.table[index]):
```

```

assignment11.py > ...
197     class HashTable:
198         def delete(self, key):
199             for i, (k, v) in enumerate(self.table.items()):
200                 if k == key:
201                     del self.table[index][i] # Remove the key-value pair
202                     return
203             raise KeyError(f"Key '{key}' not found in the hash table.")
204         # Demonstrate the functionality of the HashTable class
205         if __name__ == "__main__":
206             hash_table = HashTable()
207             while True:
208                 action = input("Enter an action (insert, search, delete, quit): ").strip().lower()
209                 if action == "insert":
210                     key = input("Enter the key to insert: ")
211                     value = input("Enter the value to insert: ")
212                     hash_table.insert(key, value)
213                     print(f"'{key}': '{value}' has been inserted into the hash table.")
214                 elif action == "search":
215                     key = input("Enter the key to search for: ")
216                     try:
217                         value = hash_table.search(key)
218                         print(f"The value for key '{key}' is: '{value}' .")
219                     except KeyError as e:
220                         print(e)
221                 elif action == "delete":
222                     key = input("Enter the key to delete: ")
223                     try:
224                         hash_table.delete(key)
225                         print(f"Key '{key}' has been deleted from the hash table.")
226                     except KeyError as e:
227                         print(e)
228                 elif action == "quit":
229                     print("Exiting the program.")
230                     break
231                 else:
232                     print("Invalid action. Please try again.")

```

Output:

```

Enter an action (insert, display, quit): insert
Enter data to insert into the linked list: 10
'10' has been inserted into the linked list.
Enter an action (insert, display, quit): display
The elements in the linked list are:
10 -> None
Enter an action (insert, display, quit): quit
Invalid action. Please try again.
Enter an action (insert, display, quit): 

```

In 101 Col 1 Spacers: 4 LINES: 9

Observation:

The hash table correctly stores and retrieves key-value pairs using a hash function and chaining for collision handling. The insert, search, and delete operations work efficiently even when multiple keys map to the same index. Edge cases such as deleting or searching for non-existing keys are handled properly without crashing the program.

Task Description #5 – Graph Representation

Prompt:

Create a Graph class using an adjacency list (dictionary).

Include methods: add_vertex, add_edge, and display. Add proper docstrings and basic error handling. Take input from user to demonstrate the functionality of the Graph class.

Code:

```
assignment11.py > ...

263
264     # Create a Graph class using an adjacency list (dictionary).
265     # Include methods: add_vertex, add_edge, and display.
266     # Add proper docstrings and basic error handling.
267     # Take input from user to demonstrate the functionality of the Graph class.
268
269     class Graph:
270         """A simple implementation of a graph using an adjacency list."""
271
272         def __init__(self):
273             """Initialize an empty graph."""
274             self.graph = {}
275
276         def add_vertex(self, vertex):
277             """Add a vertex to the graph."""
278             if vertex in self.graph:
279                 print(f"Vertex '{vertex}' already exists.")
280             else:
281                 self.graph[vertex] = []
282                 print(f"Vertex '{vertex}' has been added.")
283
284         def add_edge(self, vertex1, vertex2):
285             """Add an edge between two vertices in the graph."""
286             if vertex1 not in self.graph or vertex2 not in self.graph:
287                 print("Both vertices must exist in the graph to add an edge.")
288                 return
289             self.graph[vertex1].append(vertex2)
290             self.graph[vertex2].append(vertex1) # For undirected graph
291             print(f"Edge between '{vertex1}' and '{vertex2}' has been added.")
292
293         def display(self):
294             """Display the adjacency list of the graph."""
295             for vertex, edges in self.graph.items():
296                 print(f"{vertex}: {edges}")
297
298     # Demonstrate the functionality of the Graph class
299     if __name__ == "__main__":
300         graph = Graph()
```

```

def display(self):
    """Display the adjacency list of the graph."""
    for vertex, edges in self.graph.items():
        print(f"{vertex}: {edges}")
# Demonstrate the functionality of the Graph class
if __name__ == "__main__":
    graph = Graph()
    while True:
        action = input("Enter an action (add_vertex, add_edge, display, quit): ").strip().lower()
        if action == "add_vertex":
            vertex = input("Enter the vertex to add: ")
            graph.add_vertex(vertex)
        elif action == "add_edge":
            vertex1 = input("Enter the first vertex: ")
            vertex2 = input("Enter the second vertex: ")
            graph.add_edge(vertex1, vertex2)
        elif action == "display":
            print("The adjacency list of the graph is:")
            graph.display()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print(["Invalid action. Please try again."])

```

Output:

```

ter an action (insert, display, quit): insert
ter data to insert into the linked list: 20
0' has been inserted into the linked list.
ter an action (insert, display, quit): display
e elements in the linked list are:
-> 20 -> None
ter an action (insert, display, quit): quit
valid action. Please try again.
ter an action (insert, display, quit): insert
ter data to insert into the linked list: 30
0' has been inserted into the linked list.
ter an action (insert, display, quit): display
e elements in the linked list are:
-> 20 -> 30 -> None

```

Ln 316, Col 55 Spaces: 4

Observation:

The graph implementation correctly stores vertices and edges using an adjacency list structure. The `add_vertex` and `add_edge` methods properly update connections between nodes in an undirected manner. The `display` method successfully shows all vertices along with their connected neighbors, confirming correct functionality.

Task Description #6: Smart Hospital Management System – Data Structure Selection Prompt:

Create a Python program for Smart Hospital Management System. Implement Emergency Case Handling using a Priority Queue. Patients with higher priority (critical level) should be treated first. Include functions to add patient, treat patient, and display waiting list. Add proper docstrings, and basic error handling. Take input from user to demonstrate the functionality of the Smart Hospital Management System.

Code:

```
import heapq
class Patient:
    """A class representing a patient in the hospital."""

    def __init__(self, name, priority):
        """Initialize a patient with a name and priority level."""
        self.name = name
        self.priority = priority

    def __lt__(self, other):
        """Define less than for comparison based on priority."""
        return self.priority < other.priority
class SmartHospitalManagementSystem:
    """A Smart Hospital Management System using a priority queue for emergency case handling."""

    def __init__(self):
        """Initialize an empty priority queue for patients."""
        self.waiting_list = []

    def add_patient(self, name, priority):
        """Add a patient to the waiting list with a given priority.

        Args:
            name: The name of the patient.
            priority: The priority level of the patient (lower number means higher priority).
        """
        patient = Patient(name, priority)
        heapq.heappush(self.waiting_list, patient)

    def treat_patient(self):
        """Treat the patient with the highest priority (lowest number).

        Returns:
            The name of the treated patient.
        """
        Raises:
            IndexError: If there are no patients in the waiting list.
        """
        if not self.waiting_list:
            raise IndexError("No patients to treat")
        patient = heapq.heappop(self.waiting_list)
        return patient.name

    def display_waiting_list(self):
        """Display the waiting list of patients sorted by priority."""
        sorted_patients = sorted(self.waiting_list)
        print("Waiting List:")
        for patient in sorted_patients:
            print(f"Patient Name: {patient.name}, Priority: {patient.priority}")
```

```

# Demonstrate the functionality of the Smart Hospital Management System
if __name__ == "__main__":
    hospital_system = SmartHospitalManagementSystem()
    while True:
        action = input("Enter an action (add_patient, treat_patient, display_waiting_list, quit): ").strip().lower()
        if action == "add_patient":
            name = input("Enter the patient's name: ")
            try:
                priority = int(input("Enter the patient's priority (lower number means higher priority): "))
                hospital_system.add_patient(name, priority)
                print(f"Patient '{name}' with priority {priority} has been added to the waiting list.")
            except ValueError:
                print("Invalid priority. Please enter a number.")
        elif action == "treat_patient":
            try:
                treated_patient = hospital_system.treat_patient()
                print(f"Patient '{treated_patient}' has been treated.")
            except IndexError as e:
                print(e)
        elif action == "display_waiting_list":
            hospital_system.display_waiting_list()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please try again.")

```

Output:

```

Enter an action (add_patient, treat_patient, display_waiting_list, quit): add_patient
Enter the patient's name: John
Enter the patient's priority (lower number means higher priority): 2
Patient 'John' with priority 2 has been added to the waiting list.
Enter an action (add_patient, treat_patient, display_waiting_list, quit): add_patient
Enter the patient's name: Alice
Enter the patient's priority (lower number means higher priority): 5
Patient 'Alice' with priority 5 has been added to the waiting list.
Enter an action (add_patient, treat_patient, display_waiting_list, quit): add_patient
Enter the patient's name: Bob
Enter the patient's priority (lower number means higher priority): 3
Patient 'Bob' with priority 3 has been added to the waiting list.
Enter an action (add_patient, treat_patient, display_waiting_list, quit): treat_patient
Patient 'John' has been treated.
Enter an action (add_patient, treat_patient, display_waiting_list, quit): display_waiting_list
Waiting List:
Patient Name: Bob, Priority: 3
Patient Name: Alice, Priority: 5
Enter an action (add_patient, treat_patient, display_waiting_list, quit): quit
Exiting the program.

```

Observation:

The Priority Queue ensures that patients are treated based on the severity of their condition rather than their arrival time. Patients with critical conditions are given higher priority and attended to first, which supports effective emergency management. The system also properly handles situations when no patients are waiting and maintains efficient performance during patient insertion and treatment.

Task Description #7: Smart City Traffic Control System Prompt:

Create a Smart Traffic Emergency Vehicle System using Priority Queue. Vehicles have name and priority (1 = highest). Implement add_vehicle(), serve_vehicle(), and display_queue(). Include docstrings and basic error handling. Take input from user to demonstrate the functionality of the Smart Traffic Emergency Vehicle System.

Code:

```
# Create a Smart Traffic Emergency Vehicle System using Priority Queue.
# Vehicles have name and priority (1 = highest). Implement add_vehicle(), serve_vehicle(),
# and display_queue(). Include docstrings and basic error handling. T
# ake input from user to demonstrate the functionality of the Smart Traffic Emergency Vehicle System.
import heapq
class EmergencyVehicle:
    """A class representing an emergency vehicle with a name and priority."""
    def __init__(self, name, priority):
        """Initialize the emergency vehicle with a name and priority."""
        self.name = name
        self.priority = priority

    def __lt__(self, other):
        """Define less than for comparison based on priority."""
        return self.priority < other.priority
class SmartTrafficSystem:
    """A smart traffic system that manages emergency vehicles using a priority queue."""

    def __init__(self):
        """Initialize an empty priority queue for emergency vehicles."""
        self.queue = []

    def add_vehicle(self, name, priority):
        """Add an emergency vehicle to the priority queue."""
        vehicle = EmergencyVehicle(name, priority)
        heapq.heappush(self.queue, vehicle)
        print(f"Emergency vehicle '{name}' with priority {priority} has been added to the queue.")

    def serve_vehicle(self):
        """Serve the highest priority emergency vehicle from the queue."""
        if not self.queue:
            print("No emergency vehicles in the queue to serve.")
            return
```

```
assignment11.py > ...
335     class SmartTrafficSystem:
348         def serve_vehicle(self):
351             print("No emergency vehicles in the queue to serve.")
352             return
353             vehicle = heapq.heappop(self.queue)
354             print(f"Serving emergency vehicle '{vehicle.name}' with priority {vehicle.priority}.")
355
356         def display_queue(self):
357             """Display the current emergency vehicles in the queue."""
358             if not self.queue:
359                 print("The emergency vehicle queue is empty.")
360                 return
361                 print("Current emergency vehicles in the queue:")
362                 for vehicle in sorted(self.queue):
363                     print(f"'{vehicle.name}' with priority {vehicle.priority}")
364
# Demonstrate the functionality of the Smart Traffic Emergency Vehicle System
365 if __name__ == "__main__":
366     traffic_system = SmartTrafficSystem()
367     while True:
368         action = input("Enter an action (add_vehicle, serve_vehicle, display_queue, quit): ").strip().lower()
369         if action == "add_vehicle":
370             name = input("Enter the name of the emergency vehicle: ")
371             try:
372                 priority = int(input("Enter the priority of the emergency vehicle (1 = highest): "))
373                 traffic_system.add_vehicle(name, priority)
374             except ValueError:
375                 print("Priority must be an integer. Please try again.")
376         elif action == "serve_vehicle":
377             traffic_system.serve_vehicle()
378         elif action == "display_queue":
379             traffic_system.display_queue()
380         elif action == "quit":
381             print("Exiting the program.")
382             break
383         else:
384             print("Invalid action. Please try again.")
```

Output:

```
Enter an action (add_vehicle, serve_vehicle, display_queue, quit): add_vehicle
Enter the name of the emergency vehicle: honda
Enter the priority of the emergency vehicle (1 = highest): 5
Emergency vehicle 'honda' with priority 5 has been added to the queue.
Enter an action (add_vehicle, serve_vehicle, display_queue, quit): serve_vehicle
Serving emergency vehicle 'honda' with priority 5.
Enter an action (add_vehicle, serve_vehicle, display_queue, quit): clean
Invalid action. Please try again.
Enter an action (add_vehicle, serve_vehicle, display_queue, quit): quit
Exiting the program.
```

Observation:

The Priority Queue ensures that emergency vehicles such as ambulances and fire trucks are served before normal vehicles regardless of arrival order. This structure was chosen because traffic management requires priority-based handling rather than simple FIFO processing. The implementation successfully demonstrates efficient insertion and removal based on priority levels.

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

Prompt:

Create a Python program implementing an Order Processing System using a Queue. Include enqueue (add order), dequeue (process order), and display methods. Add proper docstrings, and basic error handling. Take input from user to demonstrate the functionality of the Order Processing System.

Code:

```
* assignment11.py ...
84     #             print("Invalid action. Please try again.")
85
86     # Create a Python program implementing an Order Processing System using a Queue.
87     # Include enqueue (add order), dequeue (process order), and display methods.
88     # Add proper docstrings, and basic error handling. Take input from user to demonstrate the
89     # functionality of the Order Processing System.
90     # Code:
91     from collections import deque
92     class OrderProcessingSystem:
93         """A simple order processing system using a queue."""
94
95         def __init__(self):
96             """Initialize an empty queue for orders."""
97             self.orders = deque()
98
99         def enqueue(self, order):
100            """Add an order to the processing queue."""
101            self.orders.append(order)
102            print(f"Order '{order}' has been added to the processing queue.")
103
104         def dequeue(self):
105            """Process the next order in the queue."""
106            if not self.orders:
107                print("No orders in the queue to process.")
108                return
109            order = self.orders.popleft()
110            print(f"Processing order '{order}'")
111
112         def display(self):
113             """Display the current orders in the processing queue."""
114             if not self.orders:
115                 print("The order processing queue is empty.")
116                 return
117             print("Current orders in the processing queue:")
118             for order in self.orders:
119                 print(f"'{order}'")
120
# Demonstrate the functionality of the Order Processing System
```

```
assignment11.py > ...
2   class OrderProcessingSystem:
2     def display(self):
3       print("The order processing queue is empty. ")
4       return
5     print("Current orders in the processing queue:")
6     for order in self.orders:
7       print(f'{order}')
8   # Demonstrate the functionality of the Order Processing System
9   if __name__ == "__main__":
10    order_system = OrderProcessingSystem()
11    while True:
12      action = input("Enter an action (enqueue, dequeue, display, quit): ").strip().lower()
13      if action == "enqueue":
14        order = input("Enter the order to add to the processing queue: ")
15        order_system.enqueue(order)
16      elif action == "dequeue":
17        order_system.dequeue()
18      elif action == "display":
19        order_system.display()
20      elif action == "quit":
21        print("Exiting the program.")
22        break
23      else:
24        print("Invalid action. Please try again.")
```

Output:

```
Enter an action (enqueue, dequeue, display, quit): enqueue
Enter the order to add to the processing queue: 10
Order '10' has been added to the processing queue.
Enter an action (enqueue, dequeue, display, quit): dequeue
Processing order '10'.
Enter an action (enqueue, dequeue, display, quit): diasply
Invalid action. Please try again.
Enter an action (enqueue, dequeue, display, quit): display
The order processing queue is empty.
Enter an action (enqueue, dequeue, display, quit): quit
Exiting the program.
```

Observation:

The Order Processing System correctly follows the FIFO principle, ensuring fairness in handling customer orders. The Queue data structure was chosen because it processes elements in the exact order they are inserted. This makes it the most suitable and logical structure for managing sequential order execution in an e-commerce system.

