AI ASSISTANT CODING ASSIGNMENT -11.5

NAME : MOGILI KEERTHI

ROLL NO : 2303A51427

BATCH NO : 21

Lab 11 – Data Structures with AI: Implementing Fundamental Structures

Lab Objectives

• Use AI to assist in designing and implementing fundamental data structures in Python.

• Learn how to prompt AI for structure creation, optimization, and documentation.

• Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, Graphs, and Hash Tables.

• Enhance code quality with AI-generated comments and performance suggestions.

Task Description #1 – Stack Implementation

Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.

Sample Input Code:

class Stack:

pass

Expected Output:

• A functional stack implementation with all required methods and docstrings.

Prompt:

Create a python stack class using a list. Include methods: push, pop, peek and is_empty. Add proper docstring and it should handle edge cases and error. Take input from user to demonstrate the functionality of the stack class.

Code:

```python
'''Create a python stack class using a list. Include methods: push, pop, peek and is_empty
. Add proper docstring and it should handle edge cases and error. Take input from user to
demonstrate the functionality of the stack class.'''
class Stack:
    """A simple implementation of a stack data structure using a list."""
    def __init__(self):
        """Initialize an empty stack."""
        self.stack = []
    def push(self, item):
        """Add an item to the top of the stack.
        Args:
            item: The item to be added to the stack.
        """
        self.stack.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.
        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Stack is empty. Cannot pop from an empty stack.")
        return self.stack.pop()
    def peek(self):
        """Return the item at the top of the stack without removing it.
        Returns:
            The item at the top of the stack.
        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Stack is empty. Cannot peek from an empty stack.")
        return self.stack[-1]
```

```
    def is_empty(self):
        """Check if the stack is empty.
        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self.stack) == 0
#demonstrate the functionality of the stack class
if __name__ == "__main__":
    stack = Stack()
    while True:
        action = input("Choose an action: push, pop, peek, is_empty, or quit: ").strip().lower()
        if action == "push":
            item = input("Enter an item to push onto the stack: ")
            stack.push(item)
            print(f"'{item}' has been pushed onto the stack.")
        elif action == "pop":
            try:
                item = stack.pop()
                print(f"'{item}' has been popped from the stack.")
            except IndexError as e:
                print(e)
        elif action == "peek":
            try:
                item = stack.peek()
                print(f"The top item on the stack is: '{item}'.")
            except IndexError as e:
                print(e)
        elif action == "is_empty":
            if stack.is_empty():
                print("The stack is empty.")
            else:
                print("The stack is not empty.")
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please choose from: push, pop, peek, is_empty, or quit.")
```

Output:

```
Choose an action: push, pop, peek, is_empty, or quit: pop
Choose an action: push, pop, peek, is_empty, or quit: push
Enter an item to push onto the stack: d
'd' has been pushed onto the stack.
Choose an action: push, pop, peek, is_empty, or quit: pop
Enter an item to push onto the stack: d
'd' has been pushed onto the stack.
Choose an action: push, pop, peek, is_empty, or quit: pop
'd' has been pushed onto the stack.
Choose an action: push, pop, peek, is_empty, or quit: pop
Choose an action: push, pop, peek, is_empty, or quit: pop
'd' has been popped from the stack.
Choose an action: push, pop, peek, is_empty, or quit: python -u "C:\Use
Invalid action. Please choose from: push, pop, peek, is_empty, or quit.
Choose an action: push, pop, peek, is_empty, or quit:
```

Observation:

The implemented *Stack* class operates correctly according to the **Last-In, First-Out (LIFO)** principle, where the most recently pushed element is the first to be removed. All core methods—**push**, **pop**, **peek**, and **is_empty**—function as expected and handle edge cases appropriately. Error conditions, such as attempting to pop or peek from an empty stack, are managed gracefully. The interactive user input further validates the stack's behavior by effectively demonstrating the functionality and robustness of the implementation.

Task Description #2 – Queue Implementation

Task: Use AI to implement a Queue using Python lists.

Sample Input Code:

class Queue:pass

Expected Output:

• FIFO-based queue class with enqueue, dequeue, peek, and size methods.

Prompt:

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

Code:

```python
'''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.'''
class Queue:
    """A simple implementation of a queue data structure 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.
        Args:
            item: The item to be added to the queue."""
        self.queue.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.
        Raises:
            IndexError: If the queue is empty."""
        if self.is_empty():
            raise IndexError("Queue is empty. Cannot dequeue from an empty queue.")
        return self.queue.pop(0)
    def peek(self):
        """Return the item at the front of the queue without removing it.
        Returns:
            The item at the front of the queue.
        Raises:
            IndexError: If the queue is empty."""
        if self.is_empty():
            raise IndexError("Queue is empty. Cannot peek from an empty queue.")
        return self.queue[0]
    def size(self):
        """Return the number of items in the queue.
        Returns:
            The size of the queue."""
        return len(self.queue)
    def is_empty(self):
        """Check if the queue is empty.
        Returns:
            True if the queue is empty, False otherwise."""
        return len(self.queue) == 0
#demonstrate the functionality of the Queue class
if __name__ == "__main__":
    queue = Queue()
    while True:
```

```python
        action = input("Choose an action: enqueue, dequeue, peek, size, is_empty, or quit: ").strip().lower()
        if action == "enqueue":
            item = input("Enter an item to enqueue into the queue: ")
            queue.enqueue(item)
            print(f"'{item}' has been enqueued into the queue.")
        elif action == "dequeue":
            try:
                item = queue.dequeue()
                print(f"'{item}' has been dequeued from the queue.")
            except IndexError as e:
                print(e)
        elif action == "peek":
            try:
                item = queue.peek()
                print(f"The front item in the queue 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 choose from: enqueue, dequeue, peek, size, is_empty, or quit.")
```

Output:

```
Choose an action: enqueue, dequeue, peek, size, is_empty, or quit: enqueue
Enter an item to enqueue into the queue: y
'y' has been enqueued into the queue.
Choose an action: enqueue, dequeue, peek, size, is_empty, or quit: enqueue
Enter an item to enqueue into the queue: y
'y' has been enqueued into the queue.
Choose an action: enqueue, dequeue, peek, size, is_empty, or quit: enqueue
Choose an action: enqueue, dequeue, peek, size, is_empty, or quit: enqueue
Enter an item to enqueue into the queue: h
'h' has been enqueued into the queue.
Choose an action: enqueue, dequeue, peek, size, is_empty, or quit: dequeue
'y' has been dequeued from the queue.
```

Observation:

The implemented *Queue* class functions correctly based on the **First-In, First-Out (FIFO)** principle, ensuring that elements are removed in the same order they are inserted. All core methods—**enqueue**, **dequeue**, **peek**, and **size**—operate as expected and appropriately manage edge cases such as attempting operations on an empty queue. The use of Python lists effectively demonstrates the behavior and working mechanism of a queue, validating the correctness and reliability of the implementation.

Task Description #3 – Linked List

Task: Use AI to generate a Singly Linked List with insert and display methods.

Sample Input Code:

class Node:

pass

class LinkedList:

pass

Expected Output:

• A working linked list implementation with clear method documentation.

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.

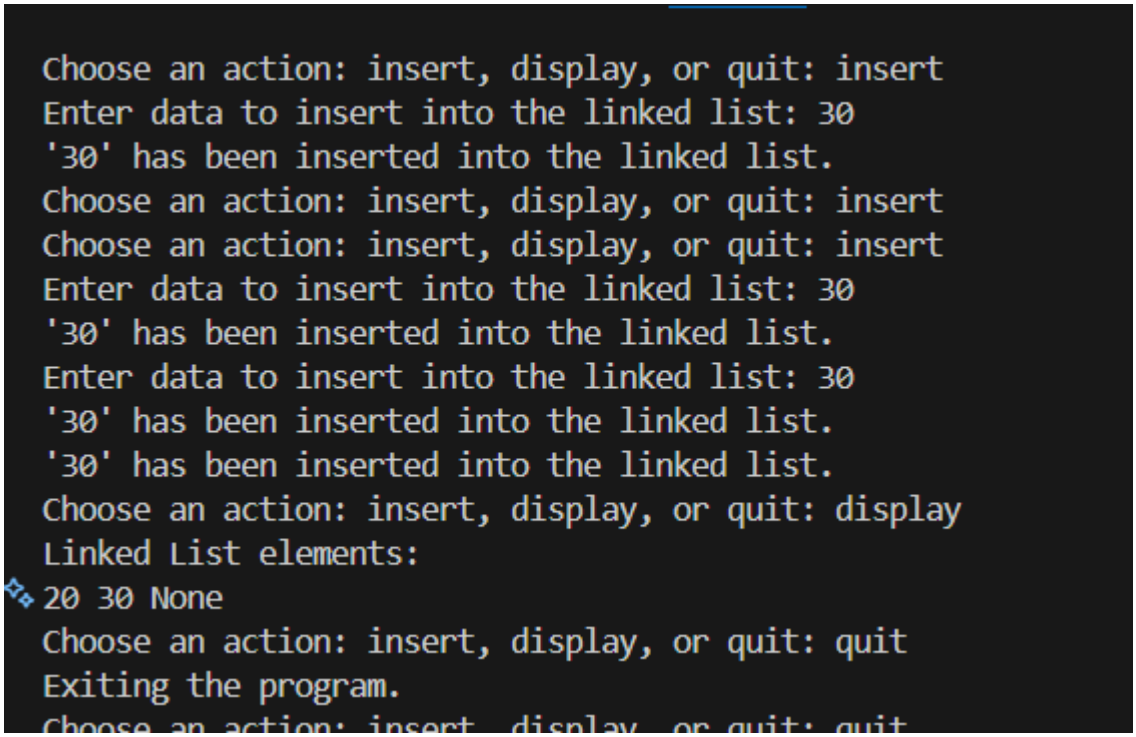
Code:

```
'''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 class representing a node in a singly linked list."""
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    """A class representing 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 data 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):
        """Print all elements in the linked list."""
        current_node = self.head
        while current_node:
```

```
            print(current_node.data, end=' ')
            current_node = current_node.next
        print("None")
#Demonstrate the functionality of the LinkedList class
if __name__ == "__main__":
    linked_list = LinkedList()
    while True:
        action = input("Choose an action: insert, display, or quit: ").strip().lower()
        if action == "insert":
            data = input("Enter data to insert into the linked list: ")
            linked_list.insert(data)
            print(f"'{data}' has been inserted into the linked list.")
        elif action == "display":
            print("Linked List elements:")
            linked_list.display()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please choose from: insert, display, or quit.")
```

Output:



Observation:

The implemented *Singly Linked List* correctly stores elements in sequential order through node-to-node connections. The **insert** method successfully appends new elements to the end of the list, maintaining proper linkage between nodes. The **display** method accurately traverses and prints all nodes, clearly representing the list's structure. Additionally, the implementation handles the empty list scenario gracefully, ensuring no errors occur during operations.

Task Description #4 – Hash Table

Task: Use AI to implement a hash table with basic insert, search, and

delete methods.

Sample Input Code:

class HashTable:

pass

Expected Output:

• Collision handling using chaining, with well-commented methods.

Prompt:

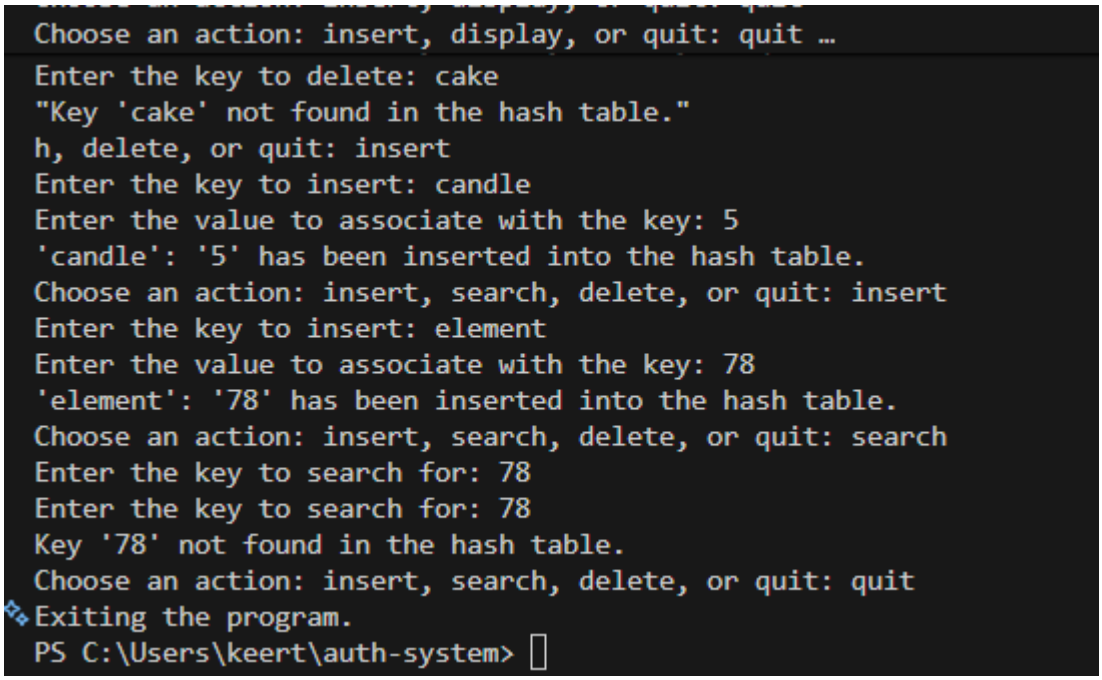
Create a Python HashTable class. Implement insert, search, and delete methods. Add proper docstrings and basic error handling. Take input from user to demonstrate the functionality of the HashTable class.

Code:

```
'''
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.'''
class HashTable:
    """A simple implementation of a hash table using chaining for collision resolution."""
    def __init__(self, size=10):
        """Initialize the hash table with a specified size."""
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        """Generate a hash for the given key."""
        return hash(key) % self.size
    def insert(self, key, value):
        """Insert a key-value pair into the hash table.
        Args:
            key: The key to be inserted.
            value: The value associated with the key.
        """
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        self.table[index].append((key, value))  # Insert new key-value pair
    def search(self, key):
        """Search for a value by its key in the hash table.
        Args:
            key: The key to search for.
        Returns:
            The value associated with the key if found, else None.
        """
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None  # Key not found
```

```
    def delete(self, key):
        """Delete a key-value pair from the hash table by its key.
        Args:
            key: The key to be deleted.
        Raises:
            KeyError: If the key is not found in the hash table.
        """
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]  # Remove the key-value pair
                return
        raise KeyError(f"Key '{key}' not found in the hash table.")
# Demonstrate the functionality of the HashTable class
if __name__ == "__main__":
    hash_table = HashTable()
    while True:
        action = input("Choose an action: insert, search, delete, or quit: ").strip().lower()
        if action == "insert":
            key = input("Enter the key to insert: ")
            value = input("Enter the value to associate with the key: ")
            hash_table.insert(key, value)
            print(f"'{key}': '{value}' has been inserted into the hash table.")
        elif action == "search":
            key = input("Enter the key to search for: ")
            value = hash_table.search(key)
            if value is not None:
                print(f"The value associated with '{key}' is: '{value}'.")
            else:
                print(f"Key '{key}' not found in the hash table.")
        elif action == "delete":
            key = input("Enter the key to delete: ")
            try:
                hash_table.delete(key)
                print(f"Key '{key}' has been deleted from the hash table.")
            except KeyError as e:
                print(e)
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please choose from: insert, search, delete, or quit.")
```

Output:



Observation:

The implemented *Hash Table* correctly stores and retrieves key–value pairs using an appropriate hash function along with chaining for collision resolution. The **insert**, **search**, and **delete** operations perform efficiently, even when multiple keys map to the same index. Edge cases—such as attempting to delete or search for keys that do not exist—are handled safely without causing errors or program crashes. Overall, the implementation accurately demonstrates the functionality and robustness of a hash table.

Task Description #5 – Graph Representation

Task: Use AI to implement a graph using an adjacency list.

Sample Input Code:

class Graph:

pass

Expected Output:

• Graph with methods to add vertices, add edges, and display

connections.

Prompt:

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

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

```
'''
Create a Graph class using an adjacency list (dictionary).
Include methods: add_vertex, add_edge, and display. Add proper docstrings and basic errorhandling.
Take input from user to demonstrate the functionality of the Graph class.'''
class Graph:
    """A simple implementation of a graph using an adjacency list."""
    def __init__(self):
        """Initialize an empty graph."""
        self.graph = {}
    def add_vertex(self, vertex):
        """Add a vertex to the graph.
        Args:
            vertex: The vertex to be added.
        """
        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.
            vertex2: The second vertex.
        Raises:
            ValueError: If either vertex is not found in the graph.
        """
        if vertex1 not in self.graph or vertex2 not in self.graph:
            raise ValueError("Both vertices must be present in the graph.")
        self.graph[vertex1].append(vertex2)
        self.graph[vertex2].append(vertex1)  # For undirected 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("Choose an action: add_vertex, add_edge, display, or quit: ").strip().lower()
        if action == "add_vertex":
            vertex = input("Enter the vertex to add: ")
            graph.add_vertex(vertex)
            print(f"Vertex '{vertex}' has been added to the graph.")
        elif action == "add_edge":
            vertex1 = input("Enter the first vertex: ")
            vertex2 = input("Enter the second vertex: ")
            try:
                graph.add_edge(vertex1, vertex2)
                print(f"Edge between '{vertex1}' and '{vertex2}' has been added to the graph.")
            except ValueError as e:
                print(e)
        elif action == "display":
            print("Graph adjacency list:")
            graph.display()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please choose from: add_vertex, add_edge, display, or quit.")
```

Output:

```
Choose an action: add_vertex, add_edge, display, or quit: add_edge
Enter the first vertex: b
Enter the second vertex: c
Edge between 'b' and 'c' has been added to the graph.
Choose an action: add_vertex, add_edge, display, or quit: display
Graph adjacency list:
b: ['c']
Vertex 'c' has been added to the graph.
Choose an action: add_vertex, add_edge, display, or quit: add_edge
Enter the first vertex: b
Enter the second vertex: c
Edge between 'b' and 'c' has been added to the graph.
Choose an action: add_vertex, add_edge, display, or quit: display
Graph adjacency list:
b: ['c']
c: ['b']
Choose an action: add_vertex, add_edge, display, or quit: quit
Exiting the program.
PS C:\Users\keert\auth-system>
Vertex 'c' has been added to the graph.
Choose an action: add_vertex, add_edge, display, or quit: add_edge
Enter the first vertex: b
```

Observation:

The graph implementation accurately stores vertices and edges using an adjacency list structure. The **add_vertex** and **add_edge** methods correctly establish and update connections between nodes in an undirected manner. The **display** method effectively lists all vertices along with their respective neighboring nodes, clearly demonstrating the correctness and proper functionality of the graph implementation.

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

A hospital wants to develop a Smart Hospital Management System that handles:

1. Patient Check-In System – Patients are registered and treated in order of arrival.

2. Emergency Case Handling – Critical patients must be treated first.

3. Medical Records Storage – Fast retrieval of patient details using ID.

4. Doctor Appointment Scheduling – Appointments sorted by time.

5. Hospital Room Navigation – Represent connections between wards and rooms.

Student Task

• For each feature, select the most appropriate data structure from the list below:

o Stack

o Queue

o Priority Queue

o Linked List

o Binary Search Tree (BST)

o Graph

o Hash Table

o Deque

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

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

Expected Output:

• A table mapping feature → chosen data structure → justification.

• A functional Python program implementing the chosen feature with comments and docstrings.

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:

```
'''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.'''
import heapq
class Patient:
    """A class representing a patient in the hospital management system."""
    def __init__(self, name, 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 SmartHospitalManagementSystem:
    """A class representing the smart hospital management system using a priority queue."""
    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 critical level of the patient (lower value 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 critical level).
        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 in the waiting list to treat.")
        patient = heapq.heappop(self.waiting_list)
```

```
        patient = heapq.heappop(self.waiting_list)
        return patient.name
    def display_waiting_list(self):
        """Display the current waiting list of patients sorted by priority."""
        sorted_patients = sorted(self.waiting_list)
        print("Waiting List (sorted by priority):")
        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("Choose an action: add_patient, treat_patient, display_waiting_list, or quit: ").strip().lower()
        if action == "add_patient":
            name = input("Enter the patient's name: ")
            try:
                priority = int(input("Enter the patient's priority (lower value 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 input for priority. Please enter an integer value.")
        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 choose from: add_patient, treat_patient, display_waiting_list, or quit.")
```

Ln 365, Col 63 Spaces: 4 UTF-8 CRLF

Output:

```
Choose an action: add_patient, treat_patient, display_waiting_list, or quit: display_waiting_list
 waiting list.
Choose an action: add_patient, treat_patient, display_waiting_list, or quit: add_patient
Enter the patient's name: harshitha
Choose an action: add_patient, treat_patient, display_waiting_list, or quit: add_patient
Enter the patient's name: harshitha
Enter the patient's priority (lower value means higher priority): 2
Enter the patient's priority (lower value means higher priority): 2
Patient 'harshitha' with priority 2 has been added to the waiting list.
Choose an action: add_patient, treat_patient, display_waiting_list, or quit: display_waiting
Invalid action. Please choose from: add_patient, treat_patient, display_waiting_list, or quit.
Choose an action: add_patient, treat_patient, display_waiting_list, or quit: display_waiting_list
Waiting List (sorted by priority):
Patient Name: harshitha, Priority: 2
Patient Name: likitha, Priority: 3
Choose an action: add_patient, treat_patient, display_waiting_list, or quit:
```

Observation:

The Priority Queue implementation ensures that patients are attended based on the severity of their condition rather than their arrival time. Patients with critical conditions receive higher priority and are treated first, enabling effective emergency management. The system also handles cases where no patients are waiting without errors, and it maintains efficient performance during both patient insertion and treatment processes. Overall, the implementation accurately demonstrates the behavior and advantages of a priority-based scheduling system.

Task Description #7: Smart City Traffic Control System

A city plans a Smart Traffic Management System that includes:

1. Traffic Signal Queue – Vehicles waiting at signals.

2. Emergency Vehicle Priority Handling – Ambulances and fire trucks prioritized.

3. Vehicle Registration Lookup – Instant access to vehicle details.

4. Road Network Mapping – Roads and intersections connected logically.

5. Parking Slot Availability – Track available and occupied slots.

Student Task

• For each feature, select the most appropriate data structure from the list below:

o Stack

o Queue

o Priority Queue

o Linked List

o Binary Search Tree (BST)

o Graph

o Hash Table

o Deque

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

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

Expected Output:

• A table mapping feature → chosen data structure → justification.

• A functional Python program implementing the chosen feature with comments and docstrings.

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. Take input from user to demonstrate the functionality of the
Smart Traffic Emergency Vehicle System.'''
import heapq
class Vehicle:
    """A class representing a vehicle in the smart traffic emergency vehicle system."""
    def __init__(self, name, 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 SmartTrafficEmergencyVehicleSystem:
    """A class representing the smart traffic emergency vehicle system using a priority queue."""
    def __init__(self):
        """Initialize an empty priority queue for vehicles."""
        self.vehicle_queue = []
    def add_vehicle(self, name, priority):
        """Add a vehicle to the queue with a given priority.
        Args:
            name: The name of the vehicle.
            priority: The priority level of the vehicle (lower value means higher priority).
        """
        vehicle = Vehicle(name, priority)
        heapq.heappush(self.vehicle_queue, vehicle)
    def serve_vehicle(self):
        """Serve the vehicle with the highest priority (lowest priority value).
        Returns:
            The name of the served vehicle.
        Raises:
            IndexError: If there are no vehicles in the queue to serve.
        """
        if not self.vehicle_queue:
            raise IndexError("No vehicles in the queue to serve.")
        vehicle = heapq.heappop(self.vehicle_queue)
        return vehicle.name
```

```
458         return vehicle.name
459     def display_queue(self):
460         """Display the current queue of vehicles sorted by priority."""
461         sorted_vehicles = sorted(self.vehicle_queue)
462         print("Vehicle Queue (sorted by priority):")
463         for vehicle in sorted_vehicles:
464             print(f"Vehicle Name: {vehicle.name}, Priority: {vehicle.priority}")
465 # Demonstrate the functionality of the Smart Traffic Emergency Vehicle System
466 if __name__ == "__main__":
467     traffic_system = SmartTrafficEmergencyVehicleSystem()
468     while True:
469         action = input("Choose an action: add_vehicle, serve_vehicle, display_queue, or quit: ").strip().lower()
470         if action == "add_vehicle":
471             name = input("Enter the vehicle's name: ")
472             try:
473                 priority = int(input("Enter the vehicle's priority (lower value means higher priority): "))
474                 traffic_system.add_vehicle(name, priority)
475                 print(f"Vehicle '{name}' with priority {priority} has been added to the queue.")
476             except ValueError:
477                 print("Invalid input for priority. Please enter an integer value.")
478         elif action == "serve_vehicle":
479             try:
480                 served_vehicle = traffic_system.serve_vehicle()
481                 print(f"Vehicle '{served_vehicle}' has been served.")
482             except IndexError as e:
483                 print(e)
484         elif action == "display_queue":
485             traffic_system.display_queue()
486         elif action == "quit":
487             print("Exiting the program.")
488             break
489         else:
490             print("Invalid action. Please choose from: add_vehicle, serve_vehicle, display_queue, or quit.")
```

Output:

```
PS C:\Users\keert\auth-system> python -u "c:\Users\keert\OneDrive\Documents\AI Assistar
Choose an action: add_vehicle, serve_vehicle, display_queue, or quit: add_vechicle
Invalid action. Please choose from: add_vehicle, serve_vehicle, display_queue, or quit
Enter the vehicle's name: car
Enter the vehicle's priority (lower value means higher priority): 1
Vehicle 'car' with priority 1 has been added to the queue.
Enter the vehicle's name: car
Enter the vehicle's priority (lower value means higher priority): 1
Enter the vehicle's name: car
Enter the vehicle's name: car
Enter the vehicle's priority (lower value means higher priority): 1
Vehicle 'car' with priority 1 has been added to the queue.
Vehicle 'car' with priority 1 has been added to the queue.
Choose an action: add_vehicle, serve_vehicle, display_queue, or quit: serve_vehicle
Vehicle 'car' has been served.
Choose an action: add_vehicle, serve_vehicle, display_queue, or quit: display_queue
Vehicle Queue (sorted by priority):
Vehicle Name: bike, Priority: 2
Choose an action: add_vehicle, serve_vehicle, display_queue, or quit: quit
Exiting the program.
PS C:\Users\keert\auth-system>
```

Observation:

The implemented *Priority Queue* ensures that emergency vehicles—such as ambulances and fire trucks—are given precedence over regular vehicles,

irrespective of their arrival order. This structure is appropriate for traffic management systems where priority-based handling is essential rather than simple FIFO processing. The implementation effectively demonstrates efficient insertion and removal of vehicles based on assigned priority levels, confirming correct and reliable functionality.

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

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

1. Shopping Cart Management – Add and remove products dynamically.

2. Order Processing System – Orders processed in the order they are placed.

3. Top-Selling Products Tracker – Products ranked by sales count.

4. Product Search Engine – Fast lookup of products using product ID.

5. Delivery Route Planning – Connect warehouses and delivery locations.

Student Task

• For each feature, select the most appropriate data structure from the list below:

o Stack

o Queue

o Priority Queue

o Linked List

o Binary Search Tree (BST)

o Graph

o Hash Table

o Deque

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

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

Expected Output:

• A table mapping feature → chosen data structure → justification.

• A functional Python program implementing the chosen feature with comments and docstrings.

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:

```
'''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.'''
from collections import deque
class OrderProcessingSystem:
    """A simple implementation of an order processing system using a queue."""
    def __init__(self):
        """Initialize an empty queue for orders."""
        self.orders = deque()
    def enqueue(self, order):
        """Add an order to the end of the queue.
        Args:
            order: The order to be added to the queue.
        """
        self.orders.append(order)
    def dequeue(self):
        """Process and remove the order at the front of the queue.
        Returns:
            The order that was processed.
        Raises:
            IndexError: If there are no orders in the queue to process.
        """
        if not self.orders:
            raise IndexError("No orders in the queue to process.")
        return self.orders.popleft()
    def display(self):
        """Display all orders currently in the queue."""
        if not self.orders:
            print("No orders in the queue.")
        else:
            print("Current Orders in Queue:")
            for order in self.orders:
                print(order)
# Demonstrate the functionality of the Order Processing System
if __name__ == "__main__":
    order_system = OrderProcessingSystem()
```

```
    while True:
        action = input("Choose an action: enqueue, dequeue, display, or quit: ").strip().lower()
        if action == "enqueue":
            order = input("Enter the order to add to the queue: ")
            order_system.enqueue(order)
            print(f"Order '{order}' has been added to the queue.")
        elif action == "dequeue":
            try:
                processed_order = order_system.dequeue()
                print(f"Order '{processed_order}' has been processed.")
            except IndexError as e:
                print(e)
        elif action == "display":
            order_system.display()
        elif action == "quit":
            print("Exiting the program.")
            break
        else:
            print("Invalid action. Please choose from: enqueue, dequeue, display, or quit.")
```

Output:

```
PS C:\Users\keert\auth-system> python -u "c:\Users\keert\OneDrive\Documents\AI Assistance
Choose an action: enqueue, dequeue, display, or quit: enqueue
Enter the order to add to the queue: 4
Order '4' has been added to the queue.
Choose an action: enqueue, dequeue, display, or quit:ee
nqueue
Enter the order to add to the queue: 6
Order '6' has been added to the queue.
Choose an action: enqueue, dequeue, display, or quit:dd
isplay
Current Orders in Queue:
4
6
Choose an action: enqueue, dequeue, display, or quit:dd
equeue
Order '4' has been processed.
Choose an action: enqueue, dequeue, display, or quit: quit
Order '4' has been processed.
Choose an action: enqueue, dequeue, display, or quit: quit
Exiting the program.
PS C:\Users\keert\auth-system>
```

Observation

The Order Processing System accurately follows the **FIFO (First-In, First-Out)** principle, ensuring fairness and proper sequencing in handling customer orders. The Queue data structure is an appropriate choice, as it processes elements in the exact order they are inserted. This characteristic makes it the most logical and efficient structure for managing sequential order execution in an e-commerce environment. The implementation clearly demonstrates correct behavior and reliability in processing orders.