SR UNIVERSITY

School of Computer Science and Artificial Intelligence

Lab Assignment # 1

Name of Student : Thota Sai Teja
Enrollment No. : 2303A51535
Batch No. : 22

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

Prompt:-
write a python program to stack class with push, pop, peek and isEmpty methods and conicse docstring.

Code:-

ssignments Codes > Assignment11.py > ...

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

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

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

    def pop(self):
        """Remove and return the item at the top of the stack. Raises an error if the stack is empty."""
        if self.is_empty():
            raise IndexError("Pop from an empty stack")
        return self.items.pop()

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

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

Output:-

```
Top item is: 2
Popped item is: 2
Is the stack empty? False
PS C:\AIAC LAB>
```

Justification:-

A stack is a Last-In-First-Out (LIFO) data structure, which is ideal for scenarios where the most recently added item needs to be accessed first. The push method allows adding items to the top of the stack, while the pop method removes and returns the top item. The peek method provides a way to view the top item without modifying the stack, and the isEmpty method checks if the stack has any items, ensuring that operations are performed safely.

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:-

write a python program to implement a queue class with enqueue, dequeue, peek and isEmpty methods and concise docstring.

Code:-

```python
#Task-2
# write a python program to implement a queue class with enqueue, dequeue, peek and isEmpty methods and concise docstring.
class Queue:
    """A simple implementation of a queue data structure with basic operations."""

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

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

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

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

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

Output:-

```
Front item is: 1
Dequeued item is: 1
Is the queue empty? False
PS C:\AIAC LAB>
```

Justification:-

A queue is suitable for managing vehicle wait times at traffic signals as it follows the First-In-First-Out (FIFO) principle, ensuring that vehicles are processed in the order they arrive at the signal.

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:-

write a python program to implement a singly linked list class with insert, delete, search and display methods and concise docstring.

Code:-

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

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

    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None

    def insert(self, data):
        """Insert a new node with the given data at the end of the list."""
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

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

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

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

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

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

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

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

Output:-

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

Justification:-

A singly linked list is suitable for this implementation as it allows for dynamic memory allocation and efficient insertion and deletion of nodes without the need for contiguous memory. The insert method adds new nodes at the end of the list, while the delete method removes nodes based on their value. The search method checks for the presence of a specific value, and the display method provides a visual representation of the list's contents. This structure is ideal for scenarios where the size of the list can change frequently and where efficient memory usage is a concern.

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:-
write a python program to implement a hash Table class with insert, delete, search and display methods and concise docstring.

Code:-

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

    def __init__(self, size=10):
        """Initialize the hash table with a specified size."""
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        """Generate a hash for the given key."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        self.table[index].append((key, value))  # Insert new key-value pair

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

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

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

SR UNIVERSITY

Output:-

```
Hash Table contents:
Bucket 5: [('age', 30)]
Bucket 8: [('name', 'Alice'), ('city', 'New York')]
Search for 'name': Alice
Hash Table contents after deletion:
Bucket 8: [('name', 'Alice'), ('city', 'New York')]
'Key not found in the hash table'
```

Justification:-

A hash table is efficient for storing and retrieving key-value pairs, making it ideal for applications like a hash table where fast access to data is crucial. The use of chaining allows for effective collision resolution, ensuring that the hash table can handle multiple entries with the same hash value without losing data integrity.

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:-

write a python program to implement a graph class using an adjacency list with add_vertex, add_edge, remove_vertex, remove_edge and display methods and concise docstring.

Code:-

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

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

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

    def add_edge(self, vertex1, vertex2):
        """Add an edge between two vertices in the graph. Raises an error if either vertex does not exist."""
        if vertex1 not in self.graph or vertex2 not in self.graph:
            raise ValueError("Both vertices must exist in the graph")
        self.graph[vertex1].append(vertex2)
        self.graph[vertex2].append(vertex1)  # For undirected graph

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

    def remove_edge(self, vertex1, vertex2):
        """Remove an edge between two vertices in the graph. Raises an error if either vertex does not exist or if the edge does not exist."""
        if vertex1 not in self.graph or vertex2 not in self.graph:
            raise ValueError("Both vertices must exist in the graph")
        if vertex2 not in self.graph[vertex1] or vertex1 not in self.graph[vertex2]:
            raise ValueError("Edge does not exist between the specified vertices")
        self.graph[vertex1].remove(vertex2)
        self.graph[vertex2].remove(vertex1)  # For undirected graph
```

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

Output:-

```
Graph contents:
A: ['B', 'C']
B: ['A']
C: ['A']
Graph contents after removing edge A-B:
A: ['C']
B: []
C: ['A']
Graph contents after removing vertex C:
A: []
B: []
PS C:\AIAC LAB>
```

Justification:-

A graph is suitable for representing complex relationships between entities, such as social networks, transportation systems, or any scenario where connections between nodes are important. The adjacency list representation allows for efficient storage and traversal of the graph, making it ideal for scenarios with a large number of vertices and edges.

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:-
write a python program to implement a a Smart Hospital Management System. Choose the most suitable data structure (Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, or Deque) for each feature: patient check-in, emergency cases, medical records storage, appointment scheduling, and hospital navigation, and briefly explain why in 2–3 sentences each. Present your answers in a clear table. Then, implement one of the features as a working Python program with proper comments and docstrings.

Explanation:-
write a python program to implement a a Smart Hospital Management System. Choose the most suitable data structure (Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, or Deque) for each feature:
patient check-in, emergency cases, medical records storage, appointment scheduling, and hospital navigation, and briefly explain why in 2–3 sentences each. Present your answers in a clear table. Then, implement one of the features as a working Python program with proper comments and docstrings.
Feature | Data Structure | Explanation
Patient Check-in | Queue | A queue is suitable for patient check-in as it follows the First-In-First-Out (FIFO) principle, ensuring that patients are attended to in the order they arrive.
Emergency Cases | Priority Queue | A priority queue is ideal for emergency cases as it allows for the prioritization of patients based on the severity of their condition, ensuring that critical cases are attended to first.
Medical Records Storage | Hash Table | A hash table is efficient for storing medical records as it allows for fast retrieval of patient information using unique keys (e.g., patient ID).
Appointment Scheduling | Linked List | A linked list can be used for appointment scheduling as it allows for dynamic insertion and deletion of appointments without the need for contiguous memory allocation.Hospital Navigation | Graph | A graph is suitable for hospital navigation as it can represent the various locations (vertices) and paths (edges) within the hospital, allowing for efficient route finding.

Code:-

```python
class PatientCheckIn:
    """A simple implementation of a patient check-in system using a queue."""

    def __init__(self):
        """Initialize an empty queue for patient check-in."""
        self.queue = []

    def check_in(self, patient_name):
        """Add a patient to the check-in queue."""
        self.queue.append(patient_name)
        print(f"{patient_name} has checked in.")

    def attend_patient(self):
        """Attend to the next patient in the queue. Raises an error if the queue is empty."""
        if not self.queue:
            raise IndexError("No patients in the queue")
        patient_name = self.queue.pop(0)
        print(f"Attending to {patient_name}.")

    def display_queue(self):
        """Display the current patients in the check-in queue."""
        if not self.queue:
            print("No patients in the queue.")
        else:
            print("Current patients in the queue:")
            for patient in self.queue:
                print(patient)
# Example usage
check_in_system = PatientCheckIn()
check_in_system.check_in("Alice")
check_in_system.check_in("Bob")
check_in_system.display_queue()
check_in_system.attend_patient()
check_in_system.display_queue()
```

Output:-

```
Alice has checked in.
Bob has checked in.
Current patients in the queue:
Alice
Bob
Attending to Alice.
Current patients in the queue:
Bob
```

Justification:-

A queue is suitable for patient check-in as it ensures that patients are attended to in the order they arrive, providing a fair and organized system for managing patient flow. This structure allows for efficient handling of patient check-ins and ensures that no patient is overlooked or attended to out of turn.

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:-
write a python program to implement a smart city traffic management system. using queue for vechicle wait in traffic signal, priority queue for emergency vehicles, graph for city road network, and hash table for vehicle registration. Briefly explain why each data structure is suitable for its respective feature in 2–3 sentences each. '''

Explanation:-
Feature | Data Structure | Explanation
Vehicle Wait in Traffic Signal | Queue | A queue is suitable for managing vehicle wait times at traffic signals as it follows the First-In-First-Out (FIFO) principle, ensuring that vehicles are processed in the order they arrive at the signal.
Emergency Vehicles | Priority Queue | A priority queue is ideal for managing emergency vehicles as it allows for the prioritization of vehicles based on their urgency, ensuring that emergency vehicles can pass through traffic signals without delay.
City Road Network | Graph | A graph is suitable for representing the city road network as it can model intersections (vertices) and roads (edges), allowing for efficient route finding and traffic flow management.
Vehicle Registration | Hash Table | A hash table is efficient for vehicle registration as it allows for fast retrieval of vehicle information using unique keys (e.g., license plate number).

Code:-

```python
class TrafficManagementSystem:
    """A simple implementation of a smart city traffic management system."""

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

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

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

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

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

Output:-

```
Car A has been added to the traffic queue.
Car B has been added to the traffic queue.
Ambulance has been added to the emergency queue.
Road added between Intersection 1 and Intersection 2.
Road added between Intersection 2 and Intersection 3.
Vehicle with license plate ABC123 registered under John Doe.
PS C:\AIAC LAB>
```

Justification:-

A queue is used for managing vehicle wait times at traffic signals to ensure that vehicles are processed in the order they arrive, while a priority queue is used for emergency vehicles to allow for prioritization based on urgency. A graph is suitable for representing the city road network as it can model intersections and roads, allowing for efficient route finding. A hash table is efficient for vehicle registration as it allows for fast retrieval of vehicle information using unique keys like license plate numbers.

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:-
write a python program to implement a smart E-commerce platform. add and remove products dynamically using a linked list, order processing order they were placed using a queue, manage customer reviews using a hash table, and recommend products based on user preferences using a graph. Briefly explain why each data structure is suitable for its respective feature in 2–3 sentences each.

Explanation:-
Feature | Data Structure | Explanation
Add and Remove Products | Linked List | A linked list is suitable for managing products as it allows for dynamic insertion and deletion of products without the need for contiguous memory allocation, making it efficient for a growing product catalog.
Order Processing | Queue | A queue is ideal for processing orders in the order they were placed, following the First-In-First-Out (FIFO) principle, ensuring that customers receive their orders in a timely manner.
Customer Reviews | Hash Table | A hash table is efficient for managing customer reviews as it allows for fast retrieval of reviews using unique keys (e.g., product ID), enabling quick access to feedback for each product.
Product Recommendations | Graph | A graph is suitable for recommending products based on user preferences as it can model relationships between products (vertices) and user interactions (edges), allowing for personalized recommendations based on similar products or user behavior.

Code:-

```python
class ECommercePlatform:
    """A simple implementation of a smart e-commerce platform."""

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

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

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

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

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

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

Output:-

```
Product 'Laptop' added to the catalog.
Product 'Smartphone' added to the catalog.
Order 'Order 1' placed.
Review added for product ID 'Laptop'.
Relationship added between 'Laptop' and 'Smartphone'.
PS C:\AIAC LAB>
```

Justification:-

A queue is suitable for patient check-in as it ensures that patients are attended to in the order they arrive, providing a fair and organized system for managing patient flow. This structure allows for efficient handling of patient check-ins and ensures that no patient is overlooked or attended to out of turn.