Task 1: Stack Implementation

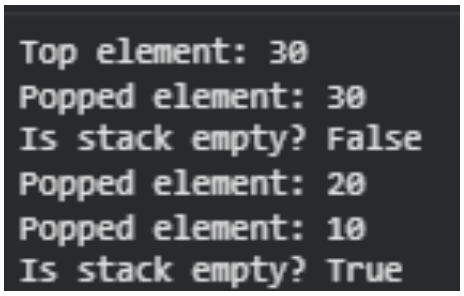
Prompt:

Generate a Python Stack class with push, pop, peek, and is_empty methods using a list internally. Include proper docstrings for the class and each method and ensure the implementation is fully functional.

Code:

```
class Stack:
    """
    A class to represent a Stack data structure (LIFO - Last In First Out).
    """

    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 top element.
        Raises IndexError if stack is empty.
        """
        if self.is_empty():
            raise IndexError("Pop from empty stack")
        return self.items.pop()

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

    def is_empty(self):
        """Return True if stack is empty, else False."""
        return len(self.items) == 0

# ------------------ Testing the Stack ------------------

s = Stack()

s.push(10)
s.push(20)
s.push(30)

print("Top element:", s.peek())
print("Popped element:", s.pop())
print("Is stack empty?", s.is_empty())
print("Popped element:", s.pop())
print("Popped element:", s.pop())
print("Is stack empty?", s.is_empty())
```

Sample Input\Output:



```
Top element: 30
Popped element: 30
Is stack empty? False
Popped element: 20
Popped element: 10
Is stack empty? True
```

Exaplanation

1. The Stack class implements the **LIFO (Last In First Out)** principle.
2. Internally, it uses a Python list to store elements.
3. The push() method adds elements to the top of the stack using append().
4. The pop() method removes and returns the top element, raising an error if the stack is empty.
5. The peek() method allows viewing the top element without removing it.
6. The is_empty() method checks whether the stack contains any elements.

Task 2: Queue Implementation

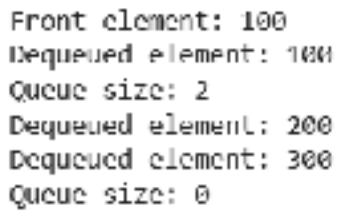
Prompt:

Generate a Python Queue class implemented using a list that follows FIFO principle. Include enqueue, dequeue, peek, and size methods with proper docstrings and demonstration output.

Code:

```
class Queue:
    """
    A class to represent a Queue data structure (FIFO - First In First Out).
    """

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

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

        :param item: Element to be added
        """
        self.items.append(item)

    def dequeue(self):
        """
        Remove and return the front element of the queue.

        :return: Front element of the queue
        :raises IndexError: If queue is empty
        """
        if self.size() == 0:
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)

    def peek(self):
        """
        Return the front element without removing it.

        :return: Front element
        :raises IndexError: If queue is empty
        """
        if self.size() == 0:
            raise IndexError("Peek from empty queue")
        return self.items[0]
    def size(self):
        """
        Return the number of elements in the queue.
        :return: Length of queue
        """
        return len(self.items)
# ------------------ Testing the Queue ------------------
q = Queue()
q.enqueue(100)
q.enqueue(200)
q.enqueue(300)
print("Front element:", q.peek())
print("Dequeued element:", q.dequeue())
print("Queue size:", q.size())
print("Dequeued element:", q.dequeue())
print("Dequeued element:", q.dequeue())
print("Queue size:", q.size())
```

<u>Sample Input\Output</u>:



```
Front element: 100
Dequeued element: 100
Queue size: 2
Dequeued element: 200
Dequeued element: 300
Queue size: 0
```

<u>Explanantion:</u>

- The Queue class follows the FIFO (First In First Out) principle.
- It uses a Python list to store elements internally.
- The enqueue() method adds elements to the rear using append().
- The dequeue() method removes elements from the front using pop(0).
- The peek() method shows the front element without removing it.
- The size() method returns the total number of elements in the queue.

Task 3: Linked List

Prompt:

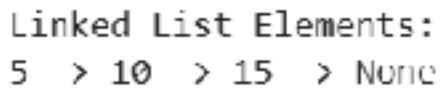
Generate a Python implementation of a Singly Linked List with Node and LinkedList classes. Include insert and display methods with proper docstrings and demonstration output.

Code:

```
class Node:
    """
    A class to represent a node in a singly linked list.
    """
    def __init__(self, data):
        """
        Initialize node with data and next pointer.
        :param data: Value to store in the node
        """
        self.data = data
        self.next = None
class LinkedList:
    """
    A class to represent a Singly Linked List.
    """
    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None
    def insert(self, data):
        """
        Insert a new node at the end of the linked list.

        :param data: Value to be inserted
        """
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = new_node
    def display(self):
        """
        Display all elements in the linked list.
        """
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")
# ------------------ Testing the Linked List ------------------
ll = LinkedList()
ll.insert(5)
ll.insert(10)
ll.insert(15)
```

```
print("Linked List Elements:")
ll.display()
```

Sample Input\Output:



```
Linked List Elements:
5  > 10  > 15  > None
```

Explanantion:

- The Node class represents each element in the linked list, storing data and a reference to the next node.
- The LinkedList class manages the list using a head pointer.
- The insert() method adds a new node at the end of the list.
- If the list is empty, the new node becomes the head.
- The display() method traverses the list and prints elements in sequence.
- The structure follows a dynamic memory approach unlike lists with fixed indexing.

Task 4: **Hash Table**

Prompt:

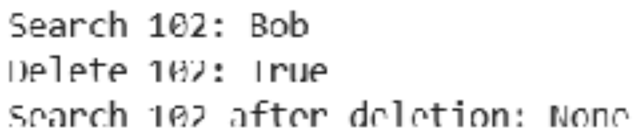
Generate a Python HashTable class implementing insert, search, and delete methods using collision handling with chaining. Include proper docstrings and demonstrate functionality with sample output.

Code:

```
class HashTable:
    """
    A class to represent a Hash Table using chaining
    for collision handling.
    """

    def __init__(self, size=10):
        """
        Initialize hash table with given size.

        :param size: Number of buckets
        """
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        """
        Private method to generate hash index.

        :param key: Key to hash
        :return: Index position
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """
        Insert a key-value pair into the hash table.
        If key already exists, update its value.
        """
        index = self._hash(key)
        bucket = self.table[index]

        for pair in bucket:
            if pair[0] == key:
                pair[1] = value
                return

        bucket.append([key, value])

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

        :return: Value if found, else None
        """
        index = self._hash(key)
        bucket = self.table[index]

        for pair in bucket:
            if pair[0] == key:
                return pair[1]

        return None

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

        :return: True if deleted, else False
        """
        index = self._hash(key)
        bucket = self.table[index]

        for i, pair in enumerate(bucket):
            if pair[0] == key:
                bucket.pop(i)
                return True

        return False
```

```
# ------------------ Testing the Hash Table ------------------

ht = HashTable(5)

ht.insert("101", "Alice")
ht.insert("102", "Bob")
ht.insert("103", "Charlie")

print("Search 102:", ht.search("102"))
print("Delete 102:", ht.delete("102"))
print("Search 102 after deletion:", ht.search("102"))
```

Sample Input\Output:



```
Search 102: Bob
Delete 102: True
Search 102 after deletion: None
```

Explananation:

- The hash table uses an array of lists (buckets) to handle collisions using chaining.
- The _hash() function computes the index using Python’s built-in hash() function.
- The insert() method adds key-value pairs or updates existing keys.
- The search() method looks for a key inside its bucket.
- The delete() method removes the key-value pair if found.
- Chaining ensures multiple keys can exist at the same index safely.

Task 5: Graph Representation

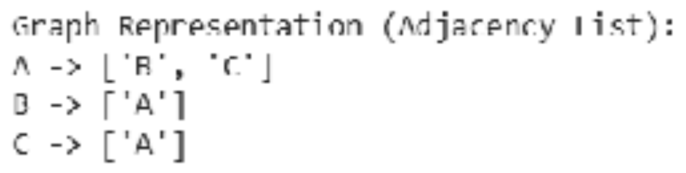
Prompt:

Generate a Python Graph class implemented using an adjacency list.
Include add_vertex, add_edge, and display methods with proper docstrings and sample output.

Code:

```
class Graph:
    """
    A class to represent a Graph using an adjacency list.
    """
    def __init__(self):
        """
        Initialize an empty graph with an adjacency list.
        """
        self.adj_list = {}

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

        :param vertex: The vertex to add
        """
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []

    def add_edge(self, vertex1, vertex2):
        """
        Add an edge between two vertices (Undirected Graph).

        :param vertex1: First vertex
        :param vertex2: Second vertex
        """
        if vertex1 not in self.adj_list:
            self.add_vertex(vertex1)
        if vertex2 not in self.adj_list:
            self.add_vertex(vertex2)

        self.adj_list[vertex1].append(vertex2)
        self.adj_list[vertex2].append(vertex1)

    def display(self):
        """
        Display the adjacency list representation of the graph.
        """
        for vertex in self.adj_list:
            print(f"{vertex} -> {self.adj_list[vertex]}")


# ------------------ Testing the Graph ------------------

g = Graph()

g.add_vertex("A")
g.add_vertex("B")
g.add_vertex("C")

g.add_edge("A", "B")
g.add_edge("A", "C")

print("Graph Representation (Adjacency List):")
g.display()
```

Sample Input\Output:



```
Graph Representation (Adjacency List):
A -> ['B', 'C']
B -> ['A']
C -> ['A']
```

Explananation:

- The graph is implemented using a **dictionary**, where each key is a vertex.
- The value of each key is a list representing adjacent vertices (Adjacency List).
- add_vertex() adds a new vertex if it doesn't already exist.
- add_edge() connects two vertices (undirected connection).
- display() prints the full adjacency list representation.
- This approach is memory-efficient compared to adjacency matrix.

Task 6: Smart Hospital Management System Data Structure Selection

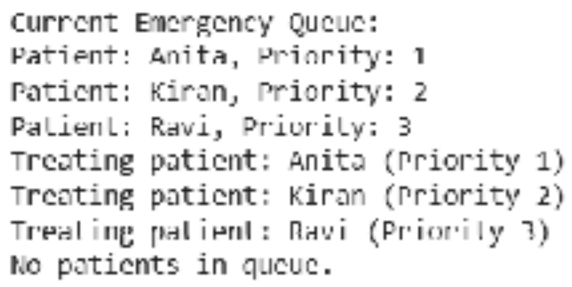
Prompt:

Generate a Python program to implement Emergency Case Handling using a Priority Queue. Critical patients must be treated first, include docstrings and demonstration output.

Code:

```
import heapq

class EmergencyQueue:
    """
    A class to manage emergency patients using a Priority Queue.
    Lower priority number means higher priority.
    """

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

    def add_patient(self, name, priority):
        """
        Add a patient to the emergency queue.

        :param name: Patient name
        :param priority: Priority level (lower value = higher priority)
        """
        heapq.heappush(self.queue, (priority, name))

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

        :return: Name of treated patient
        """
        if not self.queue:
            return "No patients in queue."
        priority, name = heapq.heappop(self.queue)
        return f"Treating patient: {name} (Priority {priority})"

    def display_queue(self):
        """Display current patients in queue."""
        print("Current Emergency Queue:")
        for patient in sorted(self.queue):
            print(f"Patient: {patient[1]}, Priority: {patient[0]}")

# ------------------ Testing the System ------------------

eq = EmergencyQueue()

eq.add_patient("Ravi", 3)
eq.add_patient("Anita", 1)   # Critical
eq.add_patient("Kiran", 2)

eq.display_queue()

print(eq.treat_patient())
print(eq.treat_patient())
print(eq.treat_patient())
print(eq.treat_patient())
```

Sample Input\Output:



```
Current Emergency Queue:
Patient: Anita, Priority: 1
Patient: Kiran, Priority: 2
Patient: Ravi, Priority: 3
Treating patient: Anita (Priority 1)
Treating patient: Kiran (Priority 2)
Treating patient: Ravi (Priority 3)
No patients in queue.
```

Explanantion:

Feature	**Selected Data Structure**	**Justification**
Patient Check-In System	**Queue**	Patients are treated in the order they arrive. Queue follows FIFO (First In First Out), making it ideal for registration and treatment sequence handling.
Emergency Case Handling	**Priority Queue**	Critical patients must be treated first regardless of arrival time. Priority Queue ensures higher-priority cases are processed before normal cases.
Medical Records Storage	**Hash Table**	Patient records need fast retrieval using ID. Hash Table provides O(1) average time complexity for search operations.
Doctor Appointment Scheduling	**Binary Search Tree (BST)**	Appointments sorted by time require ordered data. BST maintains sorted structure and allows efficient insertion and retrieval.

Hospital Room Navigation	**Graph**	Rooms and wards are connected like nodes in a network. Graph efficiently represents connections and navigation paths.

- A **Priority Queue** is implemented using Python’s heapq module.
- Patients are inserted with a priority value where smaller numbers mean higher urgency.
- The add_patient() method inserts patients into the heap.
- The treat_patient() method removes the highest priority patient first.
- This ensures emergency cases are treated before normal cases.
- Time complexity for insertion and removal is O(log n), making it efficient.

Task 7: Smart City Traffic Control System

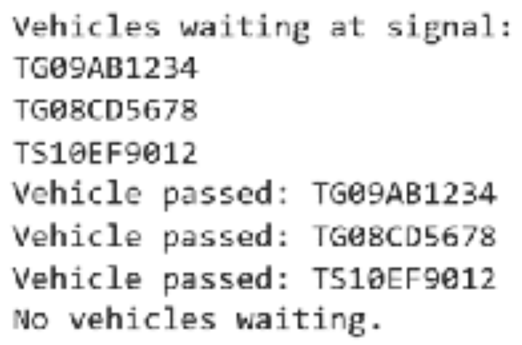
Prompt:

Generate a Python program to implement a Traffic Signal Queue using FIFO principle. Include proper docstrings and demonstrate vehicle enqueue and dequeue operations with output.

Code:

```
class TrafficSignalQueue:
    """
    A class to manage vehicles at a traffic signal using Queue (FIFO).
    """

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

    def add_vehicle(self, vehicle_number):
        """
        Add a vehicle to the traffic queue.

        :param vehicle_number: Registration number of vehicle
        """
        self.queue.append(vehicle_number)

    def allow_vehicle(self):
        """
        Allow the first vehicle to pass the signal.

        :return: Vehicle number that passed
        """
        if not self.queue:
            return "No vehicles waiting."
        return f"Vehicle passed: {self.queue.pop(0)}"
    def display_queue(self):
        """Display vehicles currently waiting at the signal."""
        print("Vehicles waiting at signal:")
        for vehicle in self.queue:
            print(vehicle)


# ------------------ Testing the System ------------------

ts = TrafficSignalQueue()

ts.add_vehicle("TS09AB1234")
ts.add_vehicle("TS08CD5678")
ts.add_vehicle("TS10EF9012")

ts.display_queue()

print(ts.allow_vehicle())
print(ts.allow_vehicle())
print(ts.allow_vehicle())
print(ts.allow_vehicle())
```

Sample Input\Output:



```
Vehicles waiting at signal:
TG09AB1234
TG08CD5678
TS10EF9012
Vehicle passed: TG09AB1234
Vehicle passed: TG08CD5678
Vehicle passed: TS10EF9012
No vehicles waiting.
```

Explananation:

- The Traffic Signal Queue is implemented using a Python list following FIFO order.
- Vehicles are added using append() to simulate arrival at the signal.
- The allow_vehicle() method removes vehicles using pop(0) to allow the first vehicle to pass.
- This ensures vehicles move in the same order they arrived.
- The system models real-world traffic flow logic.
- Queue operations maintain fairness and systematic movement.

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

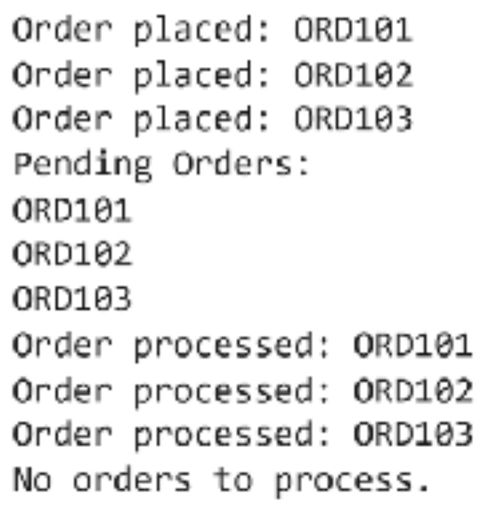
Prompt:

Generate a Python program to implement an Order Processing System using Queue (FIFO). Include proper docstrings and demonstrate order placement and processing with output.

Code:

```
class OrderQueue:
    """
    A class to manage customer orders using Queue (FIFO principle).
    """

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

    def place_order(self, order_id):
        """
        Add a new order to the queue.

        :param order_id: Unique ID of the order
        """
        self.orders.append(order_id)
        print(f"Order placed: {order_id}")

    def process_order(self):
        """
        Process the earliest placed order.

        :return: Processed order ID
        """
        if not self.orders:
            return "No orders to process."
        order = self.orders.pop(0)
        return f"Order processed: {order}"

    def display_orders(self):
        """Display all pending orders."""
        print("Pending Orders:")
        for order in self.orders:
            print(order)


# ------------------ Testing the System ------------------

oq = OrderQueue()

oq.place_order("ORD101")
oq.place_order("ORD102")
oq.place_order("ORD103")
oq.display_orders()
print(oq.process_order())
print(oq.process_order())
print(oq.process_order())
print(oq.process_order())
```

Sample Input\Output:



```
Order placed: ORD101
Order placed: ORD102
Order placed: ORD103
Pending Orders:
ORD101
ORD102
ORD103
Order processed: ORD101
Order processed: ORD102
Order processed: ORD103
No orders to process.
```

Explananation:

Feature	**Selected Data Structure**	**Justification**
Shopping Cart Management	**Linked List**	Products are added and removed dynamically. Linked List allows efficient insertion and deletion without shifting elements.
Order Processing System	**Queue**	Orders must be processed in the order they are placed. Queue follows FIFO, making it ideal for order handling.
Top-Selling Products Tracker	**Priority Queue**	Products ranked by sales count require highest sales first. Priority Queue efficiently maintains sorted order based on priority.
Product Search Engine	**Hash Table**	Fast lookup using Product ID requires constant-time access. Hash Table provides O(1) average search time.
Delivery Route Planning	**Graph**	Warehouses and delivery locations are connected like a network. Graph models routes and paths effectively.

- The Order Processing System uses a Queue to maintain FIFO order.
- Orders are added using append() which places them at the rear.
- Orders are processed using pop(0) which removes the first placed order.
- This ensures fairness and systematic processing.
- The implementation models real-world e-commerce order handling.
- The system remains simple and efficient for moderate-sized queues.