

AI ASSISTED CODING

ASSIGNMENT – 11.5

Name : B . Manali Reddy

Ht No : 230352187

Batch : 34

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

Docstring.

Prompt :

Generate a Python class Stack using a list.

Include push, pop, peek, and is_empty methods.

Add docstrings, comments, and example usage.

The screenshot shows a Google Colab notebook titled "Untitled10.ipynb". The left pane contains Python code for a `Stack` class, which uses a list as its underlying container. The right pane shows the Gemini AI interface with the following interaction:

Gemini

A: Generate a Python class Stack using a list. Include push, pop, peek, and is_empty methods. Add docstrings, comments, and example usage.

◆ I have created a Stack class for you. It uses a standard Python list as the underlying container and includes methods for common stack operations like push, pop, peek, and is_empty. I've also included some example usage to demonstrate how it works.

```

class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return len(self.items) == 0
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.items.pop()
    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]
    def size(self):
        return len(self.items)
    def __repr__(self):
        return f"Stack: {self.items}"
if __name__ == "__main__":
    s = Stack()
    print(f"Is stack empty? ({s.is_empty()})")
    print("Pushing items: 10, 20, 30")
    s.push(10)
    s.push(20)
    s.push(30)
    print(f"Current Stack: {s}")
    print(f"Top item (peek): ({s.peek()})")
    print(f"Poped item: ({s.pop()})")
    print(f"Stack after pop: {s}")
    print(f"Stack size: ({s.size()})")

```

Explain the time complexity of the Stack methods | Add a method to clear the stack | Test popping from an empty stack

What can I help you build?

+ Gemini 3 Flash ▾

Gemini can make mistakes so double-check it and use code with caution. Learn more

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 used :

Generate a Python class Queue using a list.

Include enqueue, dequeue, peek, and size methods.

Follow FIFO principle with comments and example usage.

The screenshot shows a Google Colab notebook titled "Untitled10.ipynb". The code cell contains the following Python code for a Queue class:

```

class Queue:
    def __init__(self):
        # Initialize an empty list to store queue elements
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        # pop(0) removes the first element (FIFO)
        return self.items.pop(0)

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.items[0]

    def size(self):
        return len(self.items)

    def __str__(self):
        return str([self.item])

if __name__ == "__main__":
    q = Queue()
    print("Is queue empty? ", q.is_empty())
    print("Enqueue items: 'A', 'B', 'C'")
    q.enqueue('A')
    q.enqueue('B')
    q.enqueue('C')
    print("Current Queue: ", q)
    print("Front item (peek): ", q.peek())
    print("Dequeue item: ", q.dequeue())
    print("Current Queue: ", q)
    print("Queue after dequeue: Queue: ['B', 'C']")
    print("Queue size: ", q.size())

```

The output of the code is displayed below the code cell:

```

-- Is queue empty? True
Enqueue items: 'A', 'B', 'C'
Current Queue: ['A', 'B', 'C']
Front item (peek): A
Dequeue item: A
Current Queue: ['B', 'C']
Queue after dequeue: Queue: ['B', 'C']
Queue size: 2

```

To the right of the code cell, there is a sidebar titled "Gemini" with the following text:

- A Generate a Python class Queue using a list. Include enqueue, dequeue, peek, and size methods. Follow FIFO principle with comments and example usage.
- I've implemented the Queue class for you. It follows the First-In-First-Out (FIFO) principle, where the first element added is the first one to be removed. I've included the enqueue, dequeue, peek, and size methods, along with an example of how to use them.

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 used :

Generate Node and LinkedList classes in Python.

Include insert (at end) and display methods.

Add clear docstrings and example usage.

```

class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
    def insert(self, data):
        new_node = Node(data)
        if self.head == None:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
    def display(self):
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" -> ".join(elements) + " -> None")
    if __name__ == "__main__":
        ll = LinkedList()
        print("Inserting 10, 20, 30 into the Linked List...")
        ll.insert(10)
        ll.insert(20)
        ll.insert(30)
        print("Current Linked List:")
        ll.display()

```

-- Inserting 10, 20, 30 into the Linked List...
Current Linked List:
10 -> 20 -> 30 -> None

Gemini

A Generate Node and LinkedList classes in Python. Include insert (at end) and display methods. Add clear docstrings and example usage.

I've created the `Node` and `LinkedList` classes for you. The `LinkedList` includes an `insert` method to add nodes to the end and a `display` method to visualize the list. I've also included an example to show how to use them.

Explain the Node and LinkedList classes | Add a method to remove elements from LinkedList
Generate a Python class for a Binary Search Tree

What can I help you build?
+ Gemini 3 Flash >

Gemini can make mistakes so double-check it and use code with caution. Learn more

Task Description #4 – Hash Table

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

Sample Input Code:

class Hash Table:

pass

Expected Output:

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

Prompt used :

Generate a Python HashTable class using chaining.

Include insert, search, and delete methods.

Add comments, hash function, and example usage.

```

class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [None] * size  # All elements initialized to None
        self.table_size = len(self.table)
        self.buckets = 3  # self.size // self.table_size
        self.insert('apple', 'red')
        self.insert('banana', 'yellow')
        self.insert('cherry', 'red')

    def insert(self, key, value):
        index = self._hash(key) % self.size
        pair = (key, value)
        if self.table[index] is None:
            self.table[index] = [pair]
        else:
            for pair1 in self.table[index]:
                if pair1[0] == key:
                    pair1[1] = value  # Update existing key
                    return
            self.table[index].append(pair)  # Insert new pair

    def search(self, key):
        index = self._hash(key) % self.size
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return None

    def delete(self, key):
        index = self._hash(key) % self.size
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                del self.table[index][i]
                return True
        return False

    def _hash(self, key):
        sum = 0
        for char in key:
            sum += ord(char)
        return sum % self.size

    def __str__(self):
        result = ''
        for bucket in self.table:
            if bucket is not None:
                result += f'{bucket}\n'
        return result

m = HashTable()
m.insert('apple', 'red')
m.insert('banana', 'yellow')
m.insert('cherry', 'red')
print(f'Inserting: ("apple": 5, "banana": 10, "cherry": 15")')
m.insert('banana', 15)
m.insert('banana', 20)
print(f'{"apple": 5, "banana": 20, "cherry": 15}')
print(f'{"Search banana": {m.search("banana")}}')
print(f'Deleting banana... ')
m.delete('banana')
print(f'{"banana after deletion: {m}"')
print(f'{"apple": 5, "banana": 20, "cherry": 15}')
-- Inserting: ("apple": 5, "banana": 10, "cherry": 20)
Search banana: 20
Delete banana...
Deleting apple...
Value prior deletion: {"banana": 20, "cherry": 15}

```

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 Used :

Generate a Python Graph class using adjacency list.

Include add_vertex, add_edge, and display methods.

Add comments and example usage.

The screenshot shows a Google Colab interface. On the left, a code cell contains Python code for a `Graph` class using an adjacency list. The code includes methods for initializing the graph, adding vertices, adding edges, and displaying the graph. A sample usage of the class is shown at the bottom of the cell.

```

class Graph:
    def __init__(self):
        # Initialize an empty dictionary to store adjacency lists
        self.adjacency_list = {}

    def add_vertex(self, vertex):
        if vertex not in self.adjacency_list:
            self.adjacency_list[vertex] = []

    def add_edge(self, v1, v2):
        self.add_vertex(v1)
        self.add_vertex(v2)
        self.adjacency_list[v1].append(v2)
        self.adjacency_list[v2].append(v1)

    def display(self):
        for vertex in self.adjacency_list:
            print(f'{vertex}: {", ".join(map(str, self.adjacency_list[vertex]))}')


if __name__ == '__main__':
    g = Graph()
    print("Adding vertices and edges (A-B, A-C, B-D)...")
    g.add_edge('A', 'B')
    g.add_edge('A', 'C')
    g.add_edge('B', 'D')
    print("Graph Adjacency List:")
    g.display()

```

On the right, a Gemini sidebar provides feedback on the code:

- A: Generate a Python HashTable class using chaining. Include insert, search, and delete methods. Add comments, hash function, and example usage.
- ◆ I've created a `HashTable` class that uses chaining to handle collisions. It includes the `insert`, `search`, and `delete` methods; a standard internal hash function using the modulo operator; and an example demonstrating these operations.
- A: Generate a Python Graph class using adjacency list. Include add_vertex, add_edge, and display methods. Add comments and example usage.
- ◆ I've created a `Graph` class for you. It uses an adjacency list (implemented as a dictionary) to store vertices and their connected neighbors. It includes methods for adding vertices and edges, as well as a `display` method to visualize the connections.

Below the sidebar are buttons for "Add a directed edge to the graph", "Explain how to perform a Breadth-First Search (BFS) on the graph", and "Add a new class for a tree data structure". There is also a "What can I help you build?" input field and a "Gemini 3 Flash" button.

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 Used : (FOR TABLE)

Generate a formatted table for a Smart Hospital Management System.

Features:

1. Patient Check-In System
2. Emergency Case Handling
3. Medical Records Storage
4. Doctor Appointment Scheduling
5. Hospital Room Navigation

Choose the most appropriate data structure from:

Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST),
Graph, Hash Table, Deque.

Requirements:

- Create a table: Feature | Data Structure | Justification
- Provide 2–3 sentence justification for each feature.
- Keep explanations clear and concise.

Prompt Used : (FOR CODE)

Generate a working Python program implementing Emergency Case Handling

using a Priority Queue.

Requirements:

- Use heapq module.
- Each patient has name and severity (lower number = higher priority).
- Methods:

 add_patient(name, severity)

 treat_patient()

 display_patients()

- Add docstrings and comments.
- Include example usage.
- Mention time complexity in comments.

```

class EmergencyRoom:
    def __init__(self):
        self.patients = []

    def add_patient(self, name, severity):
        self.patients.append((name, severity))

    def treat_patient(self):
        if not self.patients:
            print("The emergency room is empty.")
        else:
            patient = self.patients.pop(0)
            print(f"Treating {patient[0]} (Severity: {patient[1]})")
            return patient

    def display_patients(self):
        print([patient[0] for patient in self.patients])

```

Gemini

ensure that the patient with the highest priority (lowest severity number) is always treated first. I've also added notes on the time complexity for each operation.

A Generate a working Python program implementing Emergency Case Handling using a Priority Queue.
Requirements: - Use heapq module. - Each patient has name and severity (lower number = higher priority). - Methods: add_patient(name, severity) treat_patient() display_patients(). - Add docstrings and comments. - Include example usage. - Mention time complexity in comments.

◆ I've implemented the Emergency Case Handling system for you. It uses the heapq module to ensure that the patient with the highest priority (lowest severity value) is always retrieved first. I've also included docstrings, example usage, and time complexity notes as requested.

B Generate a working Python program implementing Emergency Case Handling using a Priority Queue.
Requirements: - Use heapq module. - Each patient has name and severity (lower number = higher priority). - Methods: add_patient(name, severity) treat_patient() display_patients(). - Add docstrings and comments. - Include example usage. - Mention time complexity in comments.

◆ I've implemented the Emergency Case Handling system for you. It uses the heapq module to ensure that the patient with the highest priority (lowest severity value) is always retrieved first. I've also included docstrings, example usage, and time complexity notes as requested.

C Explain the time complexity of "EmergencyRoom" methods | Add more patients to the "EmergencyRoom" example | Refactor the HashTable class for better performance | What can I help you build? | Gemini 3 Flash ▾

Gemini can make mistakes so double-check it and use code with caution. Learn more

```

class HashTable:
    def __init__(self):
        self.table = {}

    def add_patient(self, name, severity):
        self.table[name] = severity

    def treat_patient(self):
        if not self.table:
            print("The emergency room is empty.")
        else:
            patient = min(self.table.items(), key=lambda x: x[1])
            print(f"Treating {patient[0]} (Severity: {patient[1]})")
            del self.table[patient[0]]

    def display_patients(self):
        print([name for name in self.table])

# Example Usage
if __name__ == "__main__":
    er = EmergencyRoom()

    # Adding patients with different severity levels
    er.add_patient("Alice Brown", 3)
    er.add_patient("Bob Miller", 1)
    er.add_patient("Charlie Davis", 1)

    # Show current queue
    er.display_patients()

    # Treating patients in order of priority
    er.treat_patient() # Should treat Charlie (1)
    er.treat_patient() # Should treat Bob (2)

    # Final status
    er.display_patients()

```

Gemini

ensure that the patient with the highest priority (lowest severity number) is always treated first. I've also added notes on the time complexity for each operation.

A Generate a working Python program implementing Emergency Case Handling using a Priority Queue.
Requirements: - Use heapq module. - Each patient has name and severity (lower number = higher priority). - Methods: add_patient(name, severity) treat_patient() display_patients(). - Add docstrings and comments. - Include example usage. - Mention time complexity in comments.

◆ I've implemented the Emergency Case Handling system for you. It uses the heapq module to ensure that the patient with the highest priority (lowest severity value) is always retrieved first. I've also included docstrings, example usage, and time complexity notes as requested.

B Generate a working Python program implementing Emergency Case Handling using a Priority Queue.
Requirements: - Use heapq module. - Each patient has name and severity (lower number = higher priority). - Methods: add_patient(name, severity) treat_patient() display_patients(). - Add docstrings and comments. - Include example usage. - Mention time complexity in comments.

◆ I've implemented the Emergency Case Handling system for you. It uses the heapq module to ensure that the patient with the highest priority (lowest severity value) is always retrieved first. I've also included docstrings, example usage, and time complexity notes as requested.

C Explain the time complexity of "EmergencyRoom" methods | Add more patients to the "EmergencyRoom" example | Refactor the HashTable class for better performance | What can I help you build? | Gemini 3 Flash ▾

Gemini can make mistakes so double-check it and use code with caution. Learn more

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:

- Stack
- Queue
- Priority Queue
- Linked List
- Binary Search Tree (BST)
- Graph
- Hash Table
- 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 Used : (FOR TABLE)

Generate a formatted table for a Smart Traffic Management System.

Features:

1. Traffic Signal Queue
2. Emergency Vehicle Priority Handling
3. Vehicle Registration Lookup
4. Road Network Mapping
5. Parking Slot Availability

Choose the most appropriate data structure from:

Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST),
Graph, Hash Table, Deque.

Requirements:

- Format as a table: Feature | Data Structure | Justification
- Provide 2–3 sentence justification per feature.
- Keep explanations simple and clear.

PROMPT USED : (FOR CODE)

Generate a working Python program implementing Vehicle Registration
Lookup

using a Hash Table.

Requirements:

- Store vehicle_number as key.
- Store owner_name and vehicle_type as values.
- Methods:

register_vehicle()

search_vehicle()

delete_vehicle()

- Add proper docstrings and comments.
- Include example usage.

- Mention time complexity in comments.

The screenshot shows a Google Colab notebook titled "Untitled10.ipynb". The code implements a VehicleRegistrationSystem using a Hash Table. It includes methods for registering vehicles, searching by vehicle number, and deleting vehicles. Time complexity notes are provided for each method. The Gemini AI sidebar is visible on the right, offering suggestions and explanations related to the code.

```
class VehicleRegistrationSystem:
    """
    A system to manage vehicle registrations using a Hash Table.
    Provides constant time complexity O(1) for lookups, insertions, and deletions.
    """

    def __init__(self):
        # Python's built-in dictionary is implemented as a Hash Table
        self.registration_table = {}

    def register_vehicle(self, vehicle_number, owner_name, vehicle_type):
        """
        Adds or updates a vehicle registration.
        Time Complexity: O(1) average.
        """
        self.registration_table[vehicle_number] = {
            'owner_name': owner_name,
            'vehicle_type': vehicle_type
        }
        print(f"(Registered) Vehicle {vehicle_number} | Owner: {owner_name}")

    def search_vehicle(self, vehicle_number):
        """
        Retrieves vehicle details using the registration number.
        Time Complexity: O(1) average.
        """
        details = self.registration_table.get(vehicle_number)
        if details:
            print(f"(Found) {vehicle_number}: Owner: {details['owner_name']}, Type: {details['vehicle_type']}")
            return details
        else:
            print(f"(Not Found) Vehicle {vehicle_number} is not in the system.")
            return None

    def delete_vehicle(self, vehicle_number):
        """
        Removes a vehicle registration from the system.
        Time Complexity: O(1) average.
        """
        if vehicle_number in self.registration_table:
            removed = self.registration_table.pop(vehicle_number)
            print(f"(Deleted) Vehicle {vehicle_number} removed from records.")
            return True
        else:
            print(f"(Error) Cannot delete: {vehicle_number} not found.")
            return False
```

The screenshot shows a Google Colab notebook titled "Untitled10.ipynb". The code implements a VehicleRegistrationSystem using a Hash Table. It includes methods for registering vehicles, searching by vehicle number, and deleting vehicles. Example usage is shown at the top, and the code is annotated with time complexity notes. The Gemini AI sidebar is visible on the right, offering suggestions and explanations related to the code.

```
# Example Usage
if __name__ == "__main__":
    vrs = VehicleRegistrationSystem()

    # Registering vehicles
    vrs.register_vehicle("ABC-123", "Alice Smith", "Sedan")
    vrs.register_vehicle("XYZ-987", "Bob Johnson", "Truck")
    vrs.register_vehicle("FAST-01", "Charlie Brown", "Sports Car")

    print("... Searching Records ...")
    vrs.search_vehicle("ABC-123")
    vrs.search_vehicle("XYZ-987") # Non-existent

    print("... Updating/Deleting Records ...")
    vrs.delete_vehicle("XYZ-987")
    vrs.search_vehicle("XYZ-987") # Verify deletion

    # [Registered] Vehicle: ABC-123 | Owner: Alice Smith
    # [Registered] Vehicle: XYZ-987 | Owner: Bob Johnson
    # [Deleted] Vehicle: XYZ-987 | Owner: Charlie Brown

    ... Searching Records ...
    [Found] ABC-123: Owner: Alice Smith, Types: Sedan
    [Not Found] XYZ-987 is not in the system.

    ... Updating/Deleting Records ...
    [Deleted] Vehicle XYZ-987 removed from records.
    [Not Found] Vehicle XYZ-987 is not in the system.
```

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:
 - Stack
 - Queue
 - Priority Queue
 - Linked List
 - Binary Search Tree (BST)
 - Graph
 - Hash Table
 - 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 USED : (FOR TABLE)

Generate a formatted table for a Smart E-Commerce Platform.

Features:

1. Shopping Cart Management
2. Order Processing System
3. Top-Selling Products Tracker
4. Product Search Engine
5. Delivery Route Planning

Choose the most appropriate data structure from:

Stack, Queue, Priority Queue, Linked List, Binary Search Tree (BST),
Graph, Hash Table, Deque.

Requirements:

- Format as: Feature | Data Structure | Justification
- Provide 2–3 sentence justification per feature.
- Keep explanations short and clear.

PROMPT USED : (FOR CODE)

colab.research.google.com/drive/1KvM_-eF8ORozljf7nDkTlxYooAkY-5c#scrollTo=1b50d3d7

Untitled10.ipynb

Smart E-Commerce Platform: Data Structure Mapping

Feature	Recommended Data Structure	Justification
1. Shopping Cart Management	Linked List	A shopping cart involves frequent additions and removals as users browse. A Linked List allows for efficient, dynamic updates to the list of items without needing to reallocate memory.
2. Order Processing System	Queue (FIFO)	Orders must be handled in the sequence they are received to ensure fairness. A Queue follows a First-In-First-Out approach, ensuring that the customer who placed their order gets it first.
3. Top-Selling Products Tracker	Priority Queue	Keeping track of top-selling items requires frequent updates to rankings. A Priority Queue (Max-Heap) allows the system to instantly retrieve the highest-selling products.
4. Product Search Engine	Hash Table	Customers expect near-instant results when searching for a product by its unique ID or SKU. A Hash Table provides O(1) average time complexity for lookups, making it the ideal choice for search.
5. Delivery Route Planning	Graph	Logistics involves navigating a network of warehouses and delivery points (nodes) connected by roads (edges). A Graph structure enables the use of shortest path algorithms like Dijkstra's.

```

from collections import deque
class OrderProcessingSystem:
    def __init__(self):
        self.order_queue = deque()

    def place_order(self, order_id, customer_name):
        self.order_queue.appendleft((order_id, customer_name))

    def process_order(self):
        if not self.order_queue:
            print("No pending orders to process.")
            return None
        processed_order = self.order_queue.popleft()
        print(f"Processing Order ID: {processed_order[0]} | Customer: {customer_name}")
        return processed_order

    def display_orders(self):
        pending_orders = self.order_queue
        if pending_orders:
            print("Pending Orders (FIFO) ---")
            for order in pending_orders:
                print(f"- Order ID: {order[0]} | Customer: {customer_name}")
        else:
            print("No pending orders in queue.")


    def __str__(self):
        pending_orders = self.order_queue
        if pending_orders:
            print("Pending Orders (FIFO) ---")
            for order in pending_orders:
                print(f"- Order ID: {order[0]} | Customer: {customer_name}")
        else:
            print("No pending orders in queue.")


if __name__ == "__main__":
    oos = OrderProcessingSystem()
    oos.place_order("1001", "David Wilson")
    oos.place_order("1002", "Emma Thompson")
    oos.place_order("1003", "Frank Castle")
    oos.display_orders()
    oos.process_order() # Processes 1001
    oos.process_order() # Processes 1002
    oos.display_orders()

```

Gemini

A Generate a working Python program implementing Order Processing System using a Queue. Requirements: - Each order contains order_id and customer_name. - Methods: place_order() process_order() display_order(). - Follow FIFO principle. - Add doctests and comments. - Include example usage. - Mention time complexity in comments.

I have implemented the Order Processing System using a Queue. This follows the First-In-First-Out (FIFO) principle, ensuring that the earliest orders are processed first. I've included the required methods, doctests, and complexity analysis.

Test the OrderProcessingSystem with more complex scenarios Explain the time complexity of process_order in OrderProcessingSystem Add a new feature to the VehicleRegistrationSystem

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Gemini 3 Flash ▾

Variables Terminal 10:24AM Python 3

colab.research.google.com/drive/1KvM_-eF8ORozljf7nDkTlxYooAkY-5c#scrollTo=1b50d3d7

Untitled10.ipynb

```

if not self.order_queue:
    print("The order queue is empty.")
    return None
print("... Pending Orders (FIFO) ...")
for order in self.order_queue:
    print(f"- Order ID: {order[0]} | Customer: {customer_name}")
    print("\n")

# Example Usage:
if __name__ == "__main__":
    oos = OrderProcessingSystem()
    oos.place_order("1001", "David Wilson")
    oos.place_order("1002", "Emma Thompson")
    oos.place_order("1003", "Frank Castle")
    oos.display_orders()
    oos.process_order() # In sequence
    oos.process_order() # Processes 1001
    oos.process_order() # Processes 1002
    oos.display_orders()

    print([f"[Placed] Order ID: {order[0]} | Customer: {customer_name}" for order in oos.order_queue])
    print([f"[Processing] Order ID: {order[0]} For {customer_name}" for order in oos.order_queue])
    print([f"[Placed] Order ID: {order[0]} | Customer: {customer_name}" for order in oos.order_queue])
    print([f"- Pending Order (FIFO) ---" for order in oos.order_queue])
    print([f"- Order #1001 (David Wilson)" for order in oos.order_queue])
    print([f"- Order #1002 (Emma Thompson)" for order in oos.order_queue])
    print([f"- Order #1003 (Frank Castle)" for order in oos.order_queue])

    print([f"[Placed] Order ID: {order[0]} | Customer: {customer_name}" for order in oos.order_queue])
    print([f"[Processing] Order ID: {order[0]} For {customer_name}" for order in oos.order_queue])
    print([f"- Pending Order (FIFO) ---" for order in oos.order_queue])
    print([f"- Order #1003 (Frank Castle)" for order in oos.order_queue])

```

Gemini

A Generate a working Python program implementing Order Processing System using a Queue. Requirements: - Each order contains order_id and customer_name. - Methods: place_order() process_order() display_order(). - Follow FIFO principle. - Add doctests and comments. - Include example usage. - Mention time complexity in comments.

I have implemented the Order Processing System using a Queue. This follows the First-In-First-Out (FIFO) principle, ensuring that the earliest orders are processed first. I've included the required methods, doctests, and complexity analysis.

Test the OrderProcessingSystem with more complex scenarios Explain the time complexity of process_order in OrderProcessingSystem Add a new feature to the VehicleRegistrationSystem

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Gemini 3 Flash ▾

Variables Terminal 10:24AM Python 3