

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:

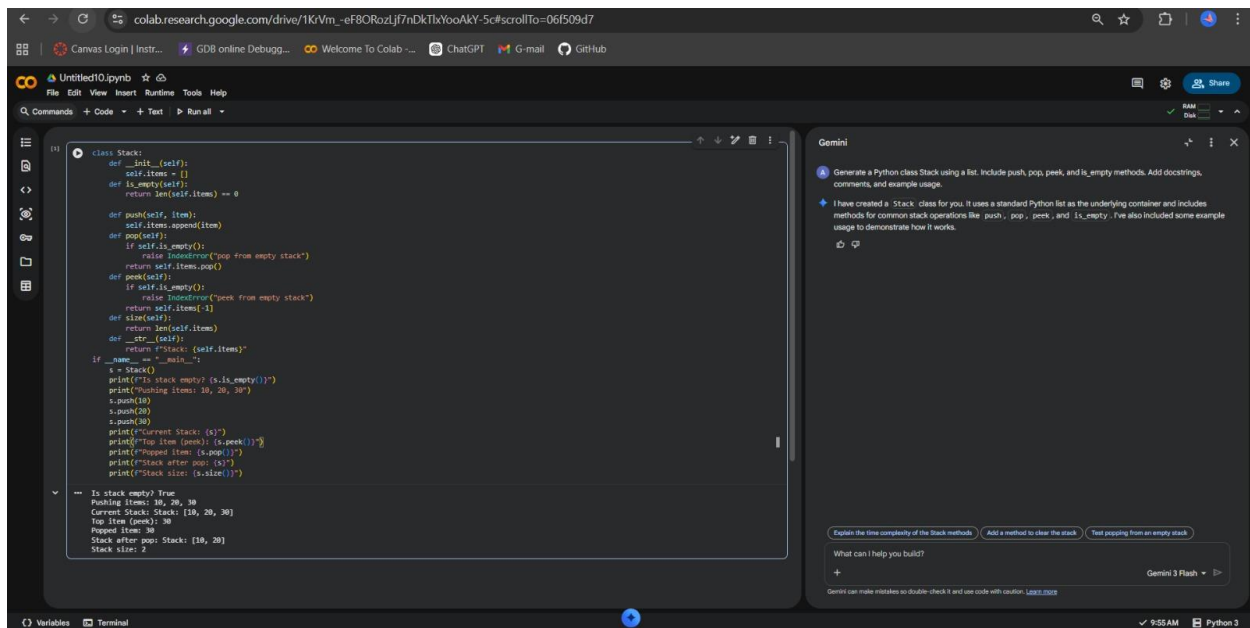
- A functional stack implementation with all required methods and Docstring.

Prompt :

Generate a Python class Stack using a list.

Include push, pop, peek, and is_empty methods.

Add docstrings, comments, and example usage.



Task Description #2 – Queue Implementation

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

Sample Input Code:

```
class Queue:
```

```
pass
```

Expected Output:

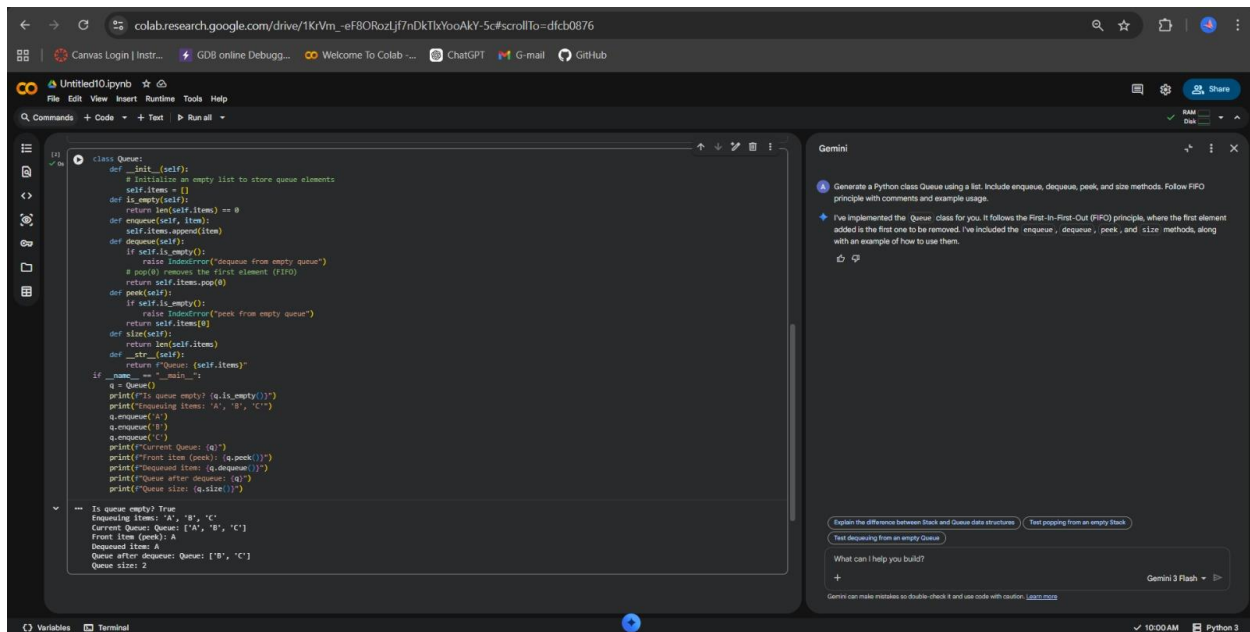
- FIFO-based queue class with enqueue, dequeue, peek, and size methods.

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.



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:

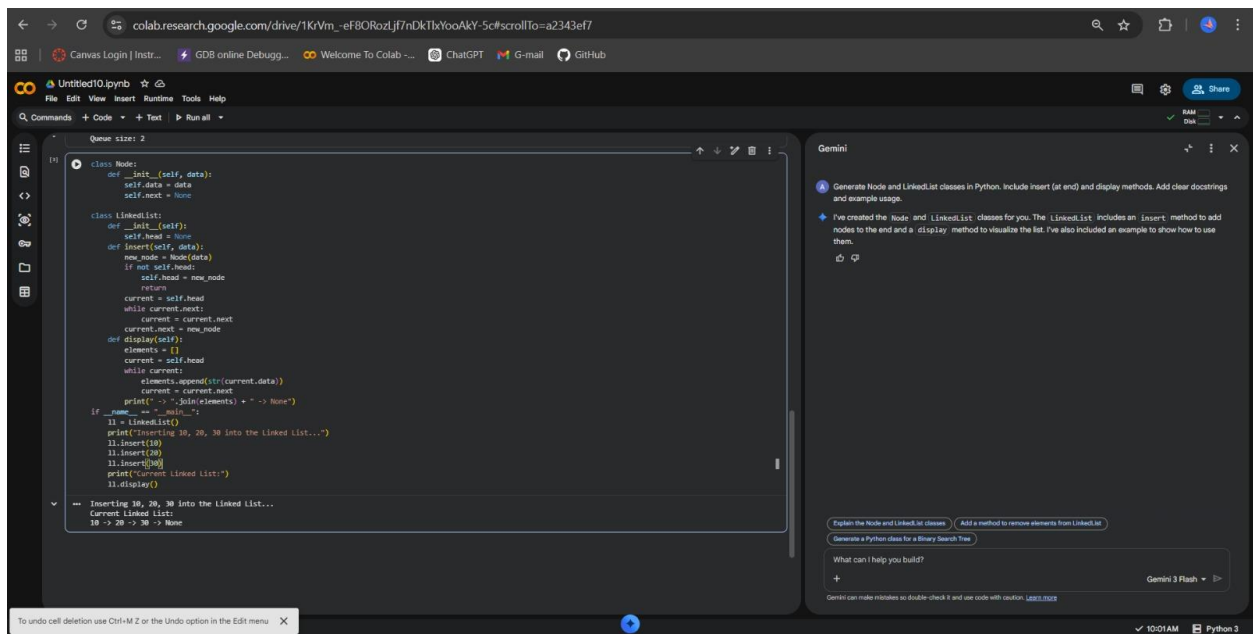
- A working linked list implementation with clear method documentation.

Prompt used :

Generate Node and LinkedList classes in Python.

Include insert (at end) and display methods.

Add clear docstrings and example usage.



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:

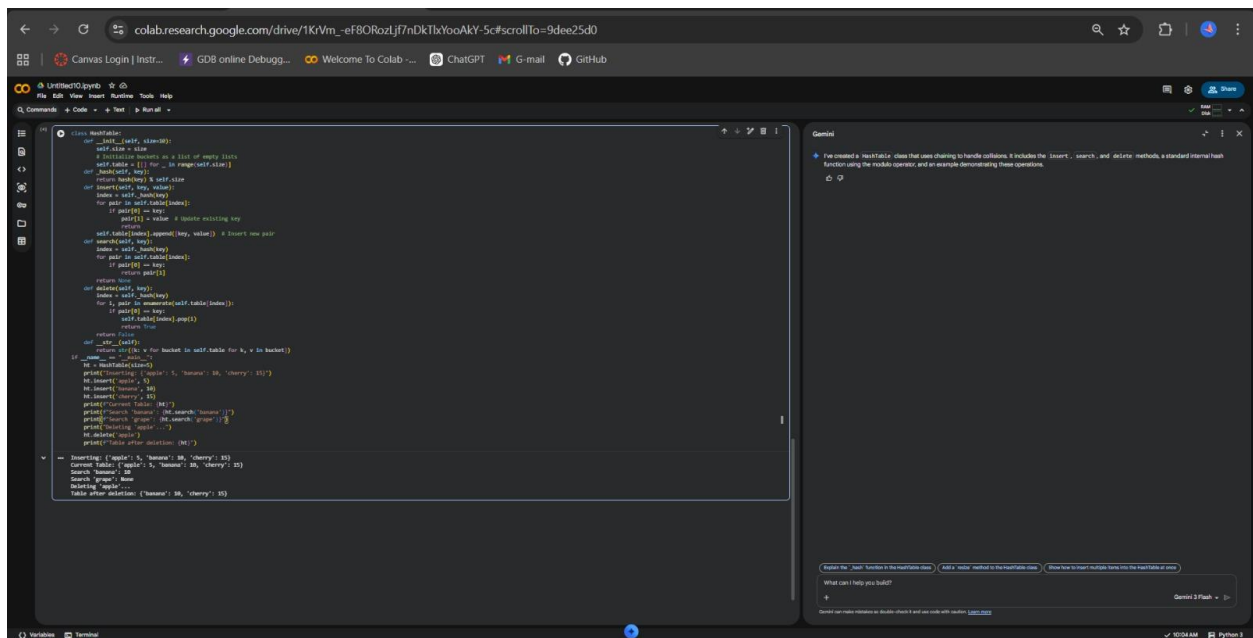
- Collision handling using chaining, with well-commented methods.

Prompt used :

Generate a Python HashTable class using chaining.

Include insert, search, and delete methods.

Add comments, hash function, and example usage.



Task Description #5 – Graph Representation

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

Sample Input Code:

```
class Graph : pass
```

Expected Output:

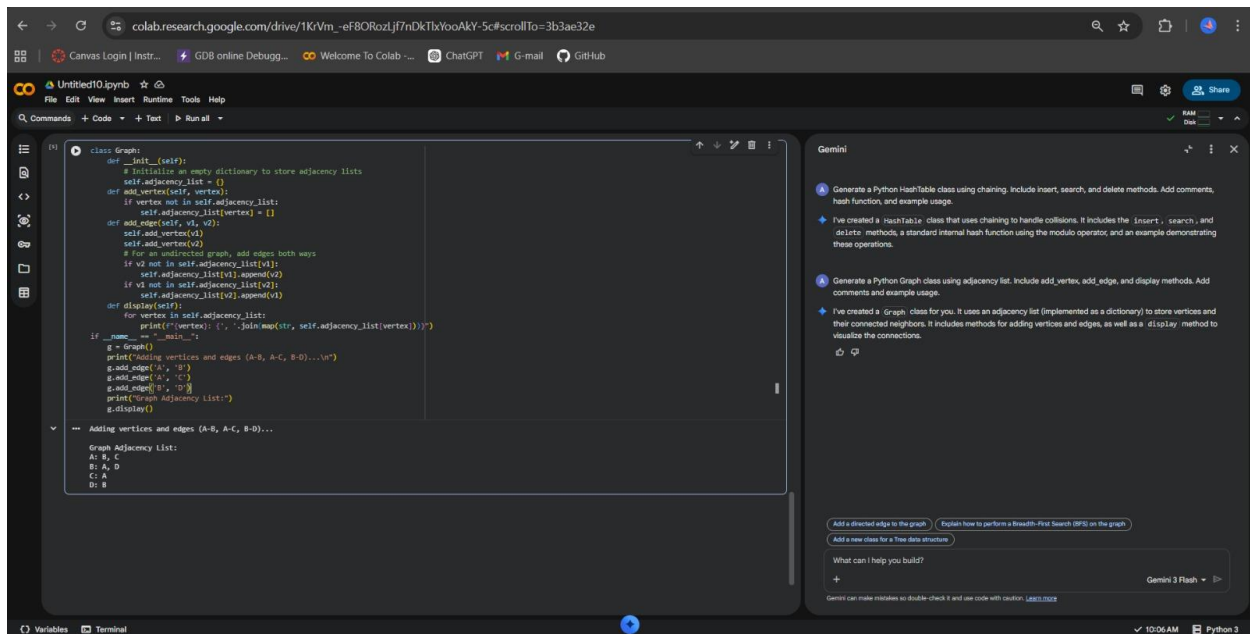
- Graph with methods to add vertices, add edges, and display connections.

Prompt Used :

Generate a Python Graph class using adjacency list.

Include add_vertex, add_edge, and display methods.

Add comments and example usage.



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.

Smart Hospital Management System: Data Structure Mapping

Feature	Recommended Data Structure	Justification
1. Patient Check-in System	Queue (FIFO)	Standard check-ins follow a first-come, first-served basis to ensure fairness. A Queue manages the flow of patients in the exact order they arrive at the registration desk.
2. Emergency Case Handling	Priority Queue	Critical cases must be treated before minor injuries regardless of arrival time. A Priority Queue allows the system to always serve the patient with the highest medical urgency.
3. Medical Records Storage	Hash Table	Retrieving a patient's history quickly is vital in healthcare. A Hash Table provides near-constant access (O(1)) through keys using a unique Patient ID as the key.
4. Doctor Appointment Scheduling	Binary Search Tree (BST)	Appointments need to be stored in a way that allows for efficient searching, insertion, and range queries (e.g., finding all slots between 2 PM and 4 PM). A balanced BST is suitable for this.
5. Hospital Room Navigation	Graph	A hospital layout consists of rooms (nodes) and hallways (edges). A Graph structure allows the system to use algorithms like BFS or Dijkstra's to find the shortest path between rooms.

```

import heapq

class EmergencyRoom:
    """
    Manages hospital emergency cases using a Priority Queue.
    Uses the heapq module to maintain patients based on medical urgency.
    """
    def __init__(self):
        # Stores patients as tuples: (severity, name)
        # heapq implements a min-heap, so lower severity value = higher priority.
        self.patients = []

    def add_patient(self, name, severity):
        """
        Adds a patient to the priority queue.
        Time Complexity: O(log N) where N is the number of patients.
        """
        heapq.heappush(self.patients, (severity, name))
        print(f"[Admitted] Patient: {name}, Severity Level: {severity}")

    def treat_patient(self):
        """
        Removes and returns the patient with the highest priority (lowest severity).
        Time Complexity: O(log N) due to heap re-adjustment.
        """
        if not self.patients:
            print("No patients currently in the emergency queue.")
            return None

        severity, name = heapq.heappop(self.patients)
        print(f"[Treating] Now attending to: {name} (Severity: {severity})")
        return (name, severity)

    def display_patients(self):
        """
        Displays the current patients in the queue, ordered by priority.
        """
        print("Waiting Patients (Ordered by Priority):")
        for severity, name in sorted(self.patients, key=lambda x: x[0]):
            print(f"- {name} (Severity: {severity})")

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

    # Adding patients with different severity levels
    # 1 is critical, 3 is minor
    er.add_patient("Alice Brown", 3)
    er.add_patient("Charlie Davis", 1)
    er.add_patient("Bob Miller", 2)

    # 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.

- 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.
- 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.

What can I help you build?

Gemini 3 Flash

```

if not self.patients:
    print("The emergency room is empty.")
    return

print("\n--- Waiting Patients (Ordered by Priority) ---")
# We sort to show the order they will be treated in
for severity, name in sorted(self.patients, key=lambda x: x[0]):
    print(f"- {name} (Severity: {severity})")
print("-----")

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

    # Adding patients with different severity levels
    # 1 is critical, 3 is minor
    er.add_patient("Alice Brown", 3)
    er.add_patient("Charlie Davis", 1)
    er.add_patient("Bob Miller", 2)

    # 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()
  
```

--- [Admitted] Patient: Alice Brown, Severity Level: 3
 [Admitted] Patient: Charlie Davis, Severity Level: 1
 [Admitted] Patient: Bob Miller, Severity Level: 2

 --- Waiting Patients (Ordered by Priority) ---
 - Charlie Davis (Severity: 1)
 - Bob Miller (Severity: 2)
 - Alice Brown (Severity: 3)

 [Treating] Now attending to: Charlie Davis (Severity: 1)
 [Treating] Now attending to: Bob Miller (Severity: 2)

 --- Waiting Patients (Ordered by Priority) ---
 - Alice Brown (Severity: 3)

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.

- 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.
- 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.

What can I help you build?

Gemini 3 Flash

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 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)

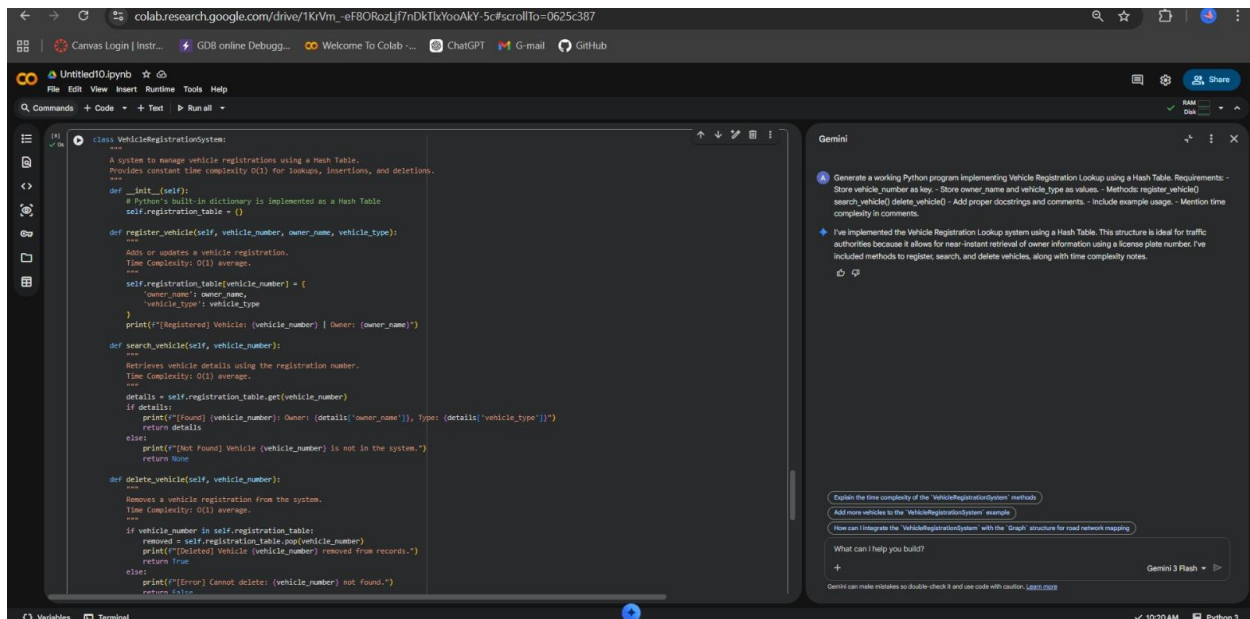
Generate a working Python program implementing Vehicle Registration Lookup

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.



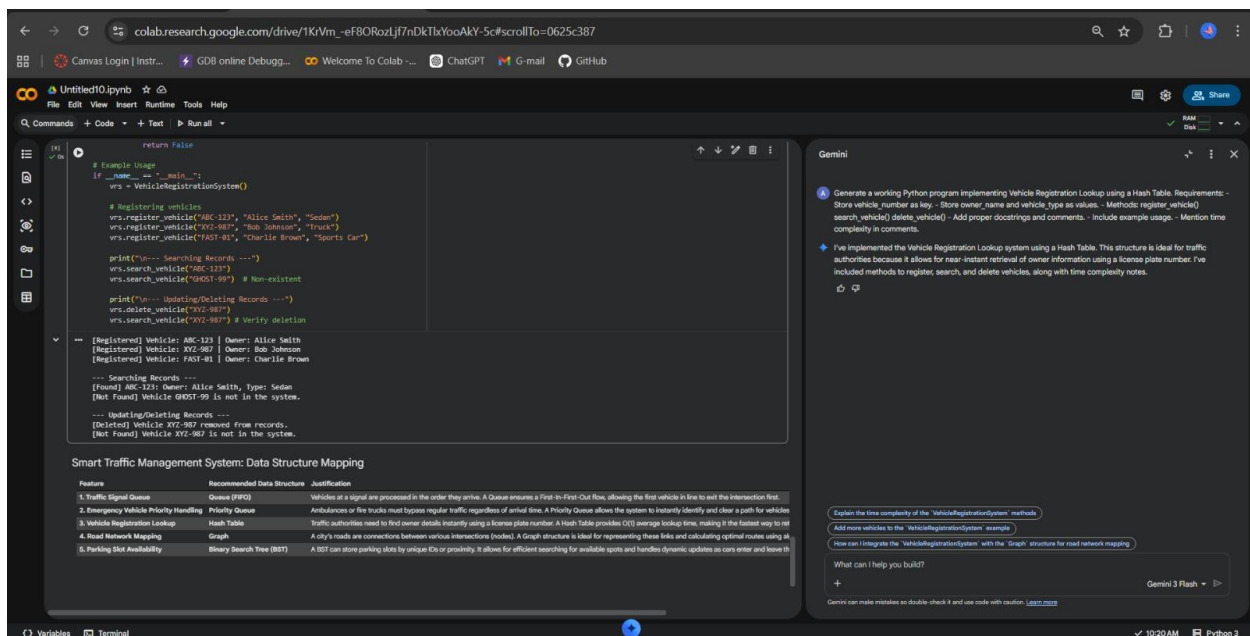
The screenshot shows a Jupyter Notebook with a class `VehicleRegistrationSystem`. The class has a `__init__` method that initializes a `self.registration_table` as a dictionary. It also has methods `register_vehicle`, `search_vehicle`, and `delete_vehicle`. Each method includes a comment indicating its time complexity: `O(1)` average for `register_vehicle` and `delete_vehicle`, and `O(1)` average for `search_vehicle`.

```
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: {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 Jupyter Notebook with example usage of the `VehicleRegistrationSystem` class and a table titled "Smart Traffic Management System: Data Structure Mapping".

```
# Example Usage
if __name__ == '__main__':
    wrs = VehicleRegistrationSystem()

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

    print("\n--- Searching Records ---")
    wrs.search_vehicle("ABC-123")
    wrs.search_vehicle("QWERT-999") # Non-existent

    print("\n--- Updating/Deleting Records ---")
    wrs.delete_vehicle("XYZ-987") # Verify deletion
    wrs.search_vehicle("XYZ-987")
```

Feature	Recommended Data Structure	Justification
1. Traffic Signal Control	Queue (FIFO)	Vehicles at a signal are processed in the order they arrive. A Queue ensures a First-In-First-Out flow, allowing the first vehicle in line to reach the intersection first.
2. Emergency Vehicle Priority Handling	Priority Queue	Ambulance or fire trucks must bypass regular traffic regardless of arrival time. A Priority Queue allows the system to instantly identify and clear a path for vehicles.
3. Vehicle Registration Lookup	Hash Table	Traffic authorities need to find owner details instantly using a license plate number. A Hash Table provides O(1) average lookup time, making it the fastest way to retrieve data.
4. Road Network Mapping	Graph	A city's roads are connections between various intersections (nodes). A Graph structure is ideal for representing these links and calculating optimal routes using algorithms like Dijkstra's.
5. Parking Slot Availability	Binary Search Tree (BST)	A city can store parking slots by unique IDs or proximity. A BST can store parking slots by unique IDs or proximity. It allows for efficient searching for available spots and handles dynamic updates as cars enter and leave the area.

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 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_eF8ORozlj7nDkTbYooAKY-5c#scrollTo=1b50d3d7

Canvas Login | Instr... GDB online Debugg... Welcome To Colab ... ChatGPT G-mail GitHub

Untitled0.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

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 the entire list.
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 (FIFO) approach, ensuring that the customer who placed their order first is served 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.
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 algorithms like Dijkstra's to find the most efficient route.

```
from collections import deque
class OrderProcessingSystem:
    """
    A system to manage customer orders using a Queue (FIFO).
    Ensures that orders are processed in the sequence they are placed.
    """
    def __init__(self):
        # deque is preferred over list for O(1) pops from the left
        self.order_queue = deque()
    def place_order(self, order_id, customer_name):
        """
        Adds a new order to the end of the queue.
        Time Complexity: O(1)
        """
        order = {'order_id': order_id, 'customer_name': customer_name}
        self.order_queue.append(order)
        print(f"[Placed] Order ID: {order_id} | Customer: {customer_name}")
    def process_order(self):
        """
        Removes and processes the oldest order from the front of the queue.
        Time Complexity: O(1)
        """
        if not self.order_queue:
            print("(System) No pending orders to process.")
            return None
        processed_order = self.order_queue.popleft()
        print(f"[Processing] Order ID: {processed_order['order_id']} for {processed_order['customer_name']}")
        return processed_order
    def display_orders(self):
        """
        Displays all pending orders in the queue.
        Time Complexity: O(N)
        """
        if not self.order_queue:
```

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 docstrings 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, docstrings, 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 3 Flash

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Variables Terminal

10:24 AM Python 3

colab.research.google.com/drive/1KVm_eF8ORozlj7nDkTbYooAKY-5c#scrollTo=1b50d3d7

Canvas Login | Instr... GDB online Debugg... Welcome To Colab ... ChatGPT G-mail GitHub

Untitled0.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

Smart E-Commerce Platform: Data Structure Mapping

```
if not self.order_queue:
    print("(System) The order queue is empty.")
    return
print(f"--- Pending Orders (FIFO) ---")
for order in self.order_queue:
    print(f"Order #{order['order_id']} ({order['customer_name']})")
print("-----")
# Example Usage
if __name__ == "__main__":
    ops = OrderProcessingSystem()
    # Placing multiple orders
    ops.place_order("1801", "David Wilson")
    ops.place_order("1802", "Emma Thompson")
    ops.place_order("1803", "Frank Castle")
    # Show Current Queue
    ops.display_orders()
    # Processing orders in sequence
    ops.process_order() # Processes 1801
    ops.process_order() # Processes 1802
    # Final status check
    ops.display_orders()

-- [Placed] Order ID: 1801 | Customer: David Wilson
[Placed] Order ID: 1802 | Customer: Emma Thompson
[Placed] Order ID: 1803 | Customer: Frank Castle

--- Pending Orders (FIFO) ---
- Order #1801 (David Wilson)
- Order #1802 (Emma Thompson)
- Order #1803 (Frank Castle)
-----

[Processing] Order ID: 1801 for David Wilson
[Processing] Order ID: 1802 for Emma Thompson

--- Pending Orders (FIFO) ---
- Order #1803 (Frank Castle)
-----
```

Double-click (or enter) to edit

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 docstrings 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, docstrings, 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 3 Flash

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Variables Terminal

10:24 AM Python 3