AIAC-11.3

Name: Siri Vempati

Ht.no:2303A51309

Batch.no:05

Task 1: Smart Contact Manager (Arrays & Linked Lists)

Scenario

SR University's student club requires a simple Contact Manager Application to store members' names and phone numbers. The system should support efficient addition, searching, and deletion of contacts.

Tasks

1. Implement the contact manager using arrays (lists).

2. Implement the same functionality using a linked list for dynamic memory allocation.

3. Implement the following operations in both approaches:

o Add a contact

o Search for a contact

o Delete a contact

4. Use GitHub Copilot to assist in generating search and delete methods.

5. Compare array vs. linked list approaches with respect to:

o Insertion efficiency

o Deletion efficiency

Expected Outcome

• Two working implementations (array-based and linked-list-based).

• A brief comparison explaining performance differences.

Prompt:

SR University's student club requires a simple Contact Manager Application to store members' names and phone numbers. The system should support

efficient addition, searching, and deletion of contacts.

1. Implement the contact manager using arrays (lists).

2. Implement the same functionality using a linked list for dynamic

memory allocation.

3. Implement the following operations in both approaches Add a contact,Search for a contact

Compare array vs. linked list approaches with respect to:Insertion efficiency,Deletion efficiency

Code:

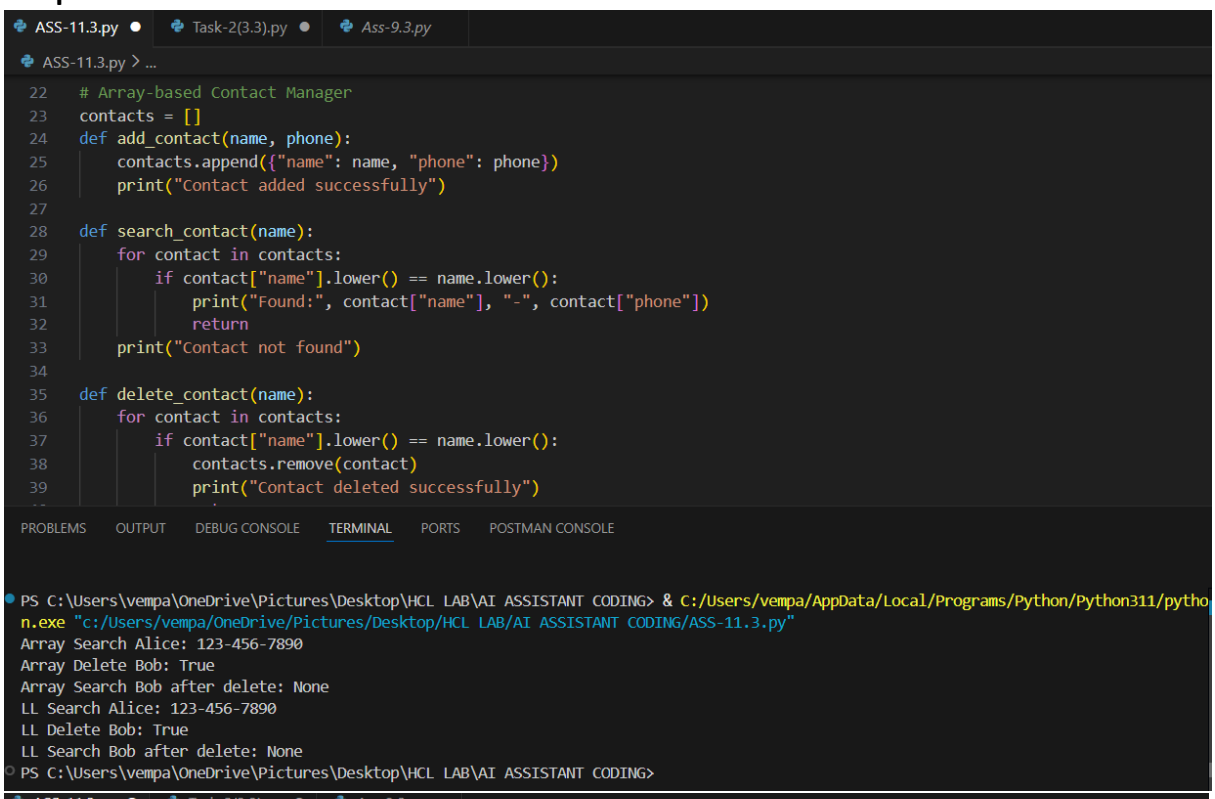
```
# Array-based Contact Manager
contacts = []
def add_contact(name, phone):
    contacts.append({"name": name, "phone": phone})
    print("Contact added successfully")
def search_contact(name):
    for contact in contacts:
        if contact["name"].lower() == name.lower():
            print("Found:", contact["name"], "-", contact["phone"])
            return
    print("Contact not found")
def delete_contact(name):
    for contact in contacts:
        if contact["name"].lower() == name.lower():
            contacts.remove(contact)
            print("Contact deleted successfully")
            return
    print("Contact not found")
def display_contacts():
    print("\nContact List:")
    for c in contacts:
```

```
        print(c["name"], "-", c["phone"])
# Example usage
add_contact("Ravi", "9876543210")
add_contact("Anu", "9123456780")
search_contact("Ravi")
delete_contact("Anu")
display_contacts()
# Linked List Node
class Node:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
        self.next = None

# Linked List Contact Manager
class ContactManager:
    def __init__(self):
        self.head = None

    def add_contact(self, name, phone):
        new_node = Node(name, phone)
        new_node.next = self.head
        self.head = new_node
        print("Contact added successfully")

    def search_contact(self, name):
        current = self.head
        while current:
            if current.name.lower() == name.lower():
```

```
            print("Found:", current.name, "-", current.phone)
            return
        current = current.next
    print("Contact not found")

def delete_contact(self, name):
    current = self.head
    prev = None

    while current:
        if current.name.lower() == name.lower():
            if prev:
                prev.next = current.next
            else:
                self.head = current.next
            print("Contact deleted successfully")
            return
        prev = current
        current = current.next

    print("Contact not found")

def display_contacts(self):
    print("\nContact List:")
    current = self.head
    while current:
        print(current.name, "-", current.phone)
        current = current.next
# Example usage
```

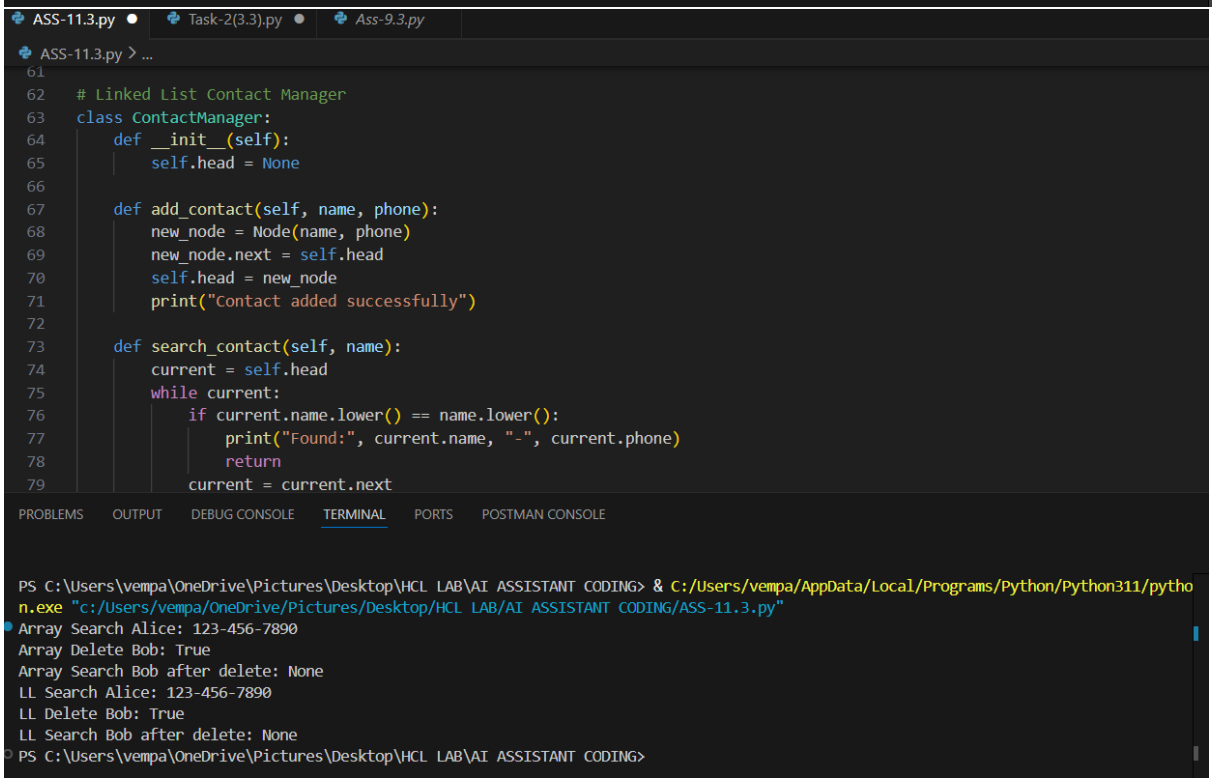
```
cm = ContactManager()
cm.add_contact("Ravi", "9876543210")
cm.add_contact("Anu", "9123456780")
cm.search_contact("Ravi")
cm.delete_contact("Anu")
cm.display_contacts()
```

Output:



```
# Array-based Contact Manager
contacts = []
def add_contact(name, phone):
    contacts.append({"name": name, "phone": phone})
    print("Contact added successfully")

def search_contact(name):
    for contact in contacts:
        if contact["name"].lower() == name.lower():
            print("Found:", contact["name"], "-", contact["phone"])
            return
    print("Contact not found")

def delete_contact(name):
    for contact in contacts:
        if contact["name"].lower() == name.lower():
            contacts.remove(contact)
            print("Contact deleted successfully")
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING> & C:/Users/vempa/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Desktop/HCL LAB/AI ASSISTANT CODING/ASS-11.3.py"
Array Search Alice: 123-456-7890
Array Delete Bob: True
Array Search Bob after delete: None
LL Search Alice: 123-456-7890
LL Delete Bob: True
LL Search Bob after delete: None
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING>
```



```
# Linked List Contact Manager
class ContactManager:
    def __init__(self):
        self.head = None

    def add_contact(self, name, phone):
        new_node = Node(name, phone)
        new_node.next = self.head
        self.head = new_node
        print("Contact added successfully")

    def search_contact(self, name):
        current = self.head
        while current:
            if current.name.lower() == name.lower():
                print("Found:", current.name, "-", current.phone)
                return
            current = current.next
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING> & C:/Users/vempa/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Desktop/HCL LAB/AI ASSISTANT CODING/ASS-11.3.py"
Array Search Alice: 123-456-7890
Array Delete Bob: True
Array Search Bob after delete: None
LL Search Alice: 123-456-7890
LL Delete Bob: True
LL Search Bob after delete: None
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING>
```

Comparison:

Performance Comparison

Insertion Efficiency:

- Array-based: O(1) for appending a contact.

- Linked List-based: O(1) for adding a contact at the head.

Deletion Efficiency:

- Array-based: O(n) in the worst case (if the contact is at the end).

- Linked List-based: O(n) in the worst case (if the contact is atthe end), but O(1) if the contact is at the head.

Conclusion:

Both implementations have similar insertion efficiency, but the linkedlist can be more efficient for deletions if the contact is near the head,while the array may require shifting elements, leading to O(n) time complexity.

Task 2: Library Book Search System (Queues & Priority Queues)

Scenario

The SRU Library manages book borrow requests. Students and faculty submit

requests, but faculty requests must be prioritized over student requests.

Tasks

1. Implement a Queue (FIFO) to manage book requests.

2. Extend the system to a Priority Queue, prioritizing faculty requests.

3. Use GitHub Copilot to assist in generating:

o enqueue() method

o dequeue() method

4. Test the system with a mix of student and faculty requests.

Expected Outcome

• Working queue and priority queue implementations.

• Correct prioritization of faculty requests.

Prompt:

The SRU Library manages book borrow requests. Students and faculty submit

requests, but faculty requests must be prioritized over student requests.

Tasks

1. Implement a Queue (FIFO) to manage book requests.

2. Extend the system to a Priority Queue, prioritizing faculty requests.

3. Use GitHub Copilot to assist in generating:enqueue() method, dequeue() method

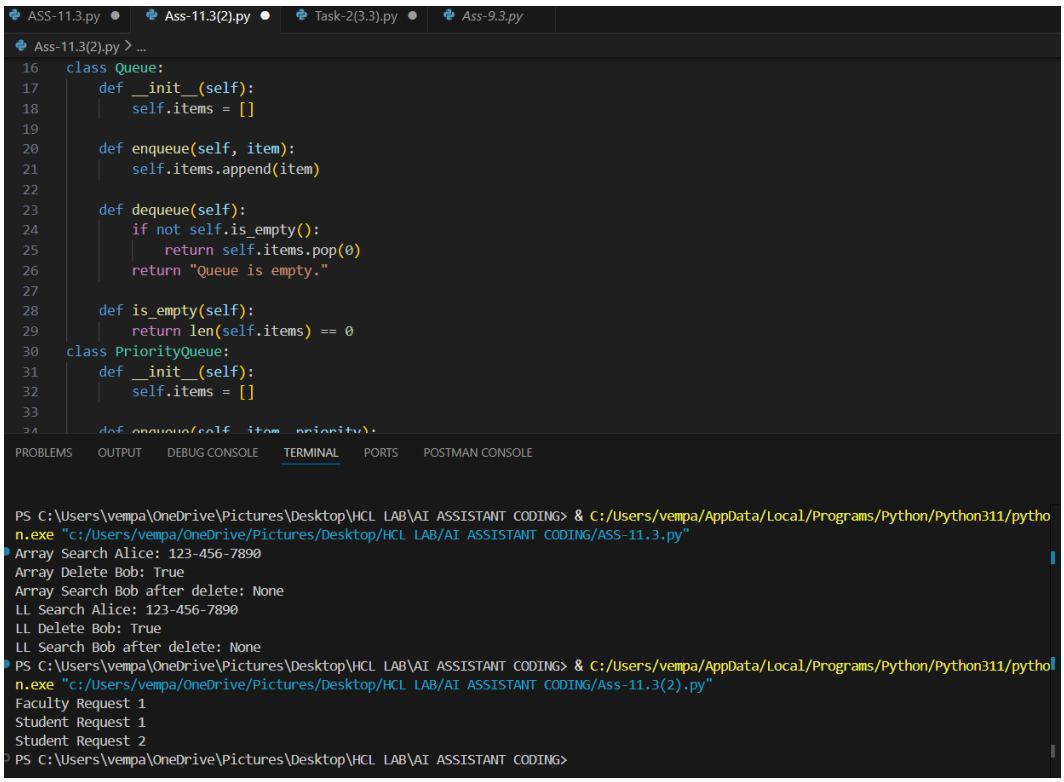
4. Test the system with a mix of student and faculty requests.

Working queue and priority queue implementations,Correct prioritization of faculty requests.

Code:

```
class Queue:
    def __init__(self):
        self.items = []

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

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        return "Queue is empty."

    def is_empty(self):
        return len(self.items) == 0
class PriorityQueue:
    def __init__(self):
        self.items = []

    def enqueue(self, item, priority):
        self.items.append((priority, item))
        self.items.sort(key=lambda x: x[0])  # Sort by priority

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)[1]  # Return the item with the highest priority
        return "Priority Queue is empty."
```

```
    def is_empty(self):
        return len(self.items) == 0
# Example usage
pq = PriorityQueue()
pq.enqueue("Student Request 1", priority=2)
pq.enqueue("Faculty Request 1", priority=1)
pq.enqueue("Student Request 2", priority=2)
print(pq.dequeue())  # Should return "Faculty Request 1"
print(pq.dequeue())  # Should return "Student Request 1"
print(pq.dequeue())  # Should return "Student Request 2"
```



Comparison:

The code implements two classes, Queue and PriorityQueue, to manage book requests in a library system. The Queue class follows the First-In-First-Out (FIFO) principle, allowing items to be enqueued and dequeued in the order they were added. The PriorityQueue class allows for prioritization of requests, where each item is associated with a priority level. When items are enqueued, they are sorted based on their priority, ensuring that higher-priority items are dequeued first. The example usage demonstrates how both classes work, showing the correct order of processing requests based on their type (student vs. faculty).

Task 3: Emergency Help Desk (Stack Implementation)

Scenario

SR University's IT Help Desk receives technical support tickets from students and staff. While tickets are received sequentially, issue escalation follows a Last-In, First-Out (LIFO) approach.

Tasks

1. Implement a Stack to manage support tickets.

2. Provide the following operations:

o push(ticket)

o pop()

o peek()

3. Simulate at least five tickets being raised and resolved.

4. Use GitHub Copilot to suggest additional stack operations such as:

o Checking whether the stack is empty

o Checking whether the stack is full (if applicable)

Expected Outcome

• Functional stack-based ticket management system.

• Clear demonstration of LIFO behavior.

Prompt:

SR University's IT Help Desk receives technical support tickets from students and staff. While tickets are received sequentially, issue escalation follows a Last-In, First-Out (LIFO) approach. implement a Stack to manage support tickets.

Operations like push(ticket),pop(), peek()

Simulate at least five tickets being raised and resolved.

1.Checking whether the stack is empty

2.Checking whether the stack is full (if applicable)

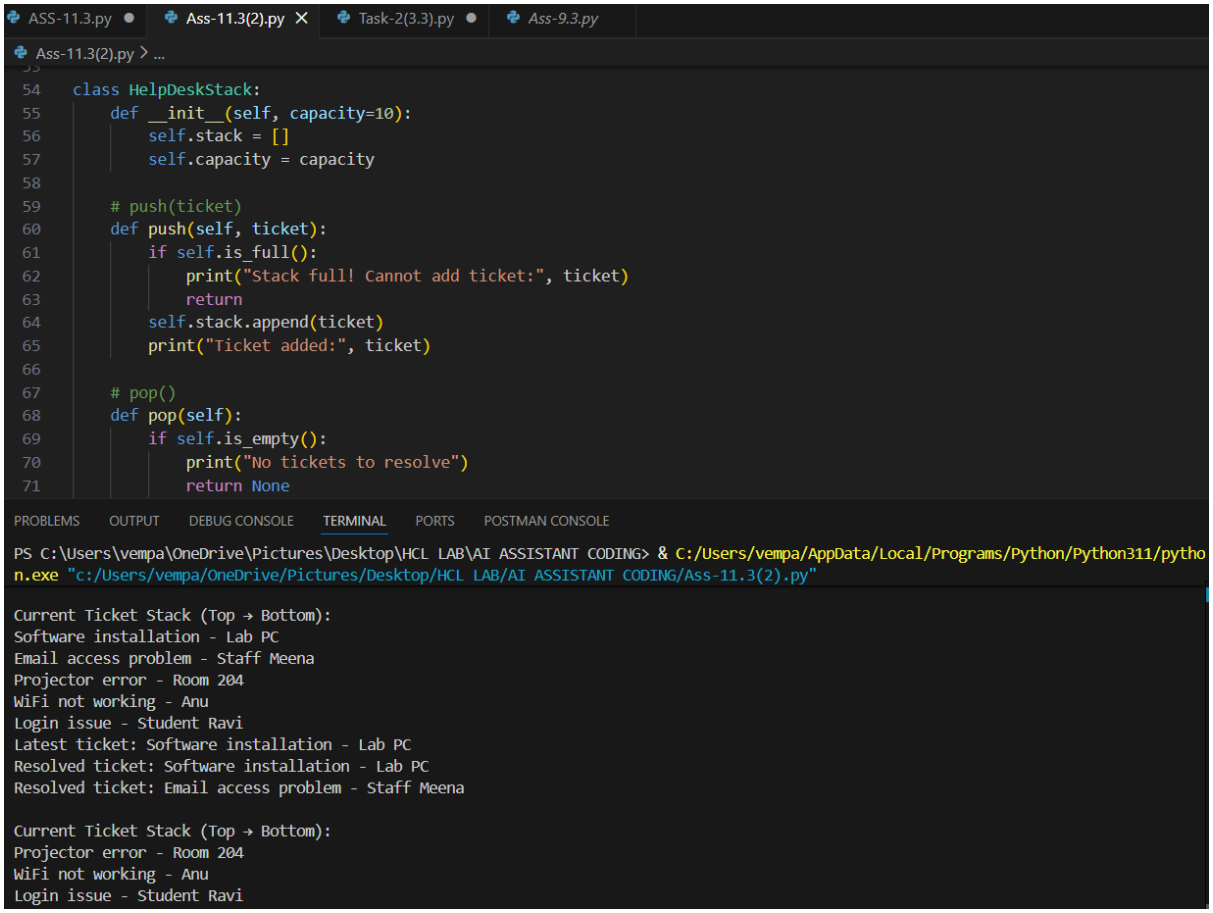
Code:

```
class HelpDeskStack:
    def __init__(self, capacity=10):
```

```
    self.stack = []
    self.capacity = capacity

# push(ticket)
def push(self, ticket):
    if self.is_full():
        print("Stack full! Cannot add ticket:", ticket)
        return
    self.stack.append(ticket)
    print("Ticket added:", ticket)

# pop()
def pop(self):
    if self.is_empty():
        print("No tickets to resolve")
        return None
    ticket = self.stack.pop()
    print("Resolved ticket:", ticket)
    return ticket

# peek()
def peek(self):
    if self.is_empty():
        print("No tickets in stack")
        return None
    print("Latest ticket:", self.stack[-1])
    return self.stack[-1]

# Copilot-style helper: check empty
```

```
    def is_empty(self):
        return len(self.stack) == 0

    # Copilot-style helper: check full
    def is_full(self):
        return len(self.stack) == self.capacity

    # display stack
    def display(self):
        print("\nCurrent Ticket Stack (Top → Bottom):")
        for ticket in reversed(self.stack):
            print(ticket)

helpdesk = HelpDeskStack(capacity=5)
# Raising tickets
helpdesk.push("Login issue - Student Ravi")
helpdesk.push("WiFi not working - Anu")
helpdesk.push("Projector error - Room 204")
helpdesk.push("Email access problem - Staff Meena")
helpdesk.push("Software installation - Lab PC")
helpdesk.display()
# Check latest ticket
helpdesk.peek()
# Resolving tickets (LIFO behavior)
helpdesk.pop()
helpdesk.pop()
helpdesk.display()
```

Output:



Explanation:

1. The HelpDeskStack class implements a stack data structure with a specified capacity.

2. The push() method adds a ticket to the stack, checking for overflow.

3. The pop() method removes and returns the most recent ticket, checking for underflow

4. The peek() method allows viewing the latest ticket without removing it.

5. The is_empty() and is_full() methods assist in managing stack state.

Task 4: Hash Table

Objective

To implement a Hash Table and understand collision handling.

Task Description

Use AI to generate a hash table with:

• Insert

• Search

• Delete

Starter Code

class HashTable:

pass

Expected Outcome

• Collision handling using chaining

• Well-commented methods

Prompt:

To implement a Hash Table and understand collision handling. generate a hash table with:

Insert, Search,Delete

Starter Code

class HashTable:

pass

Collision handling using chaining,Well-commented methods

Code:

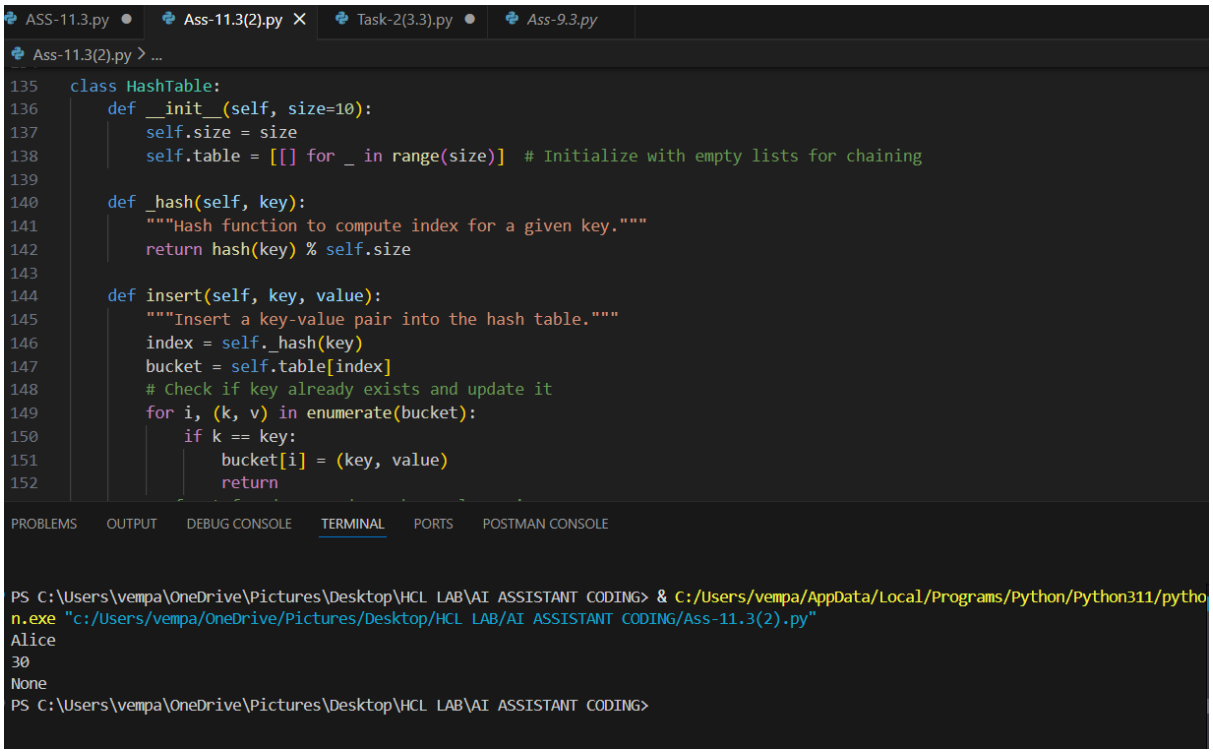
```
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]  # Initialize with empty lists for chaining

    def _hash(self, key):
        """Hash function to compute index for a 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)
        bucket = self.table[index]
        # Check if key already exists and update it
        for i, (k, v) in enumerate(bucket):
            if k == key:
```

```
                bucket[i] = (key, value)
                return
        # If not found, append new key-value pair
        bucket.append((key, value))

    def search(self, key):
        """Search for a value by its key."""
        index = self._hash(key)
        bucket = self.table[index]
        for k, v in bucket:
            if k == key:
                return v
        return None  # Key not found

    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        index = self._hash(key)
        bucket = self.table[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                return True  # Successfully deleted
        return False  # Key not found
# Example usage
hash_table = HashTable()
hash_table.insert("name", "Alice")
hash_table.insert("age", 30)
print(hash_table.search("name"))  # Output: Alice
print(hash_table.search("age"))   # Output: 30
```

hash_table.delete("name")

print(hash_table.search("name")) # Output: None

Output:



Explanation

1. The HashTable class uses chaining to handle collisions.

2. The _hash method computes an index for a given key.

3. The insert method adds a key-value pair, updating if the key already exists.

4. The search method retrieves a value by its key.

5. The delete method removes a key-value pair and returns True if successful, False otherwise.

Task 5: Real-Time Application Challenge

Scenario

Design a Campus Resource Management System with the following

features:

• Student Attendance Tracking

• Event Registration System

• Library Book Borrowing

• Bus Scheduling System

• Cafeteria Order Queue

Student Tasks

1. Choose the most appropriate data structure for each feature.

2. Justify your choice in 2–3 sentences.

3. Implement one selected feature using AI-assisted code generation.

Expected Outcome

• Mapping table: Feature → Data Structure → Justification

• One fully working Python implementation

Prompt:

Design a Campus Resource Management System with the following features:

• Student Attendance Tracking

• Event Registration System

• Library Book Borrowing

• Bus Scheduling System

• Cafeteria Order Queue

Choose the most appropriate data structure for each feature.

2. Justify your choice in 2–3 sentences.

3. Implement one selected feature using AI-assisted code generation

Code:

```
from collections import deque

class CafeteriaQueue:
    def __init__(self):
        self.orders = deque()
    # Add new order
    def place_order(self, student, item):
        self.orders.append({"student": student, "item": item})
        print(f"Order placed: {student} ordered {item}")
```

```
    # Serve next order (FIFO)
    def serve_order(self):
        if not self.orders:
            print("No pending orders")
            return
        order = self.orders.popleft()
        print(f"Serving: {order['student']} - {order['item']}")

    # View next order
    def next_order(self):
        if not self.orders:
            print("No pending orders")
            return
        order = self.orders[0]
        print(f"Next order: {order['student']} - {order['item']}")

    # Display all orders
    def display_orders(self):
        print("\nCurrent Order Queue:")
        for o in self.orders:
            print(o["student"], "-", o["item"])
# Example usage
cafeteria = CafeteriaQueue()
cafeteria.place_order("Ravi", "Sandwich")
cafeteria.place_order("Anu", "Salad")
cafeteria.place_order("Meena", "Pasta")
cafeteria.display_orders()
cafeteria.next_order()
```

cafeteria.serve_order()

cafeteria.display_orders()

Explanation:

1. The CafeteriaQueue class uses a deque from the collections module to implement a

FIFO queue for managing orders.

2. The place_order method adds a new order to the queue, while serve_order removes

The next order to be served. The next_order method allows viewing the upcoming order without removing it.

3. The display_orders method shows all pending orders in the queue. This implementation efficiently manages the order flow in a cafeteria setting.

Output: