SR UNIVERSITY

School of Computer Science and Artificial Intelligence

Lab Assignment # 11.3

Program	**: B. Tech (CSE)**
Specialization	**: -**
Course Title	**: AI Assisted Coding**
Course Code	**: 23CS002PC304**
Semester	**II**
Academic Session	**: 2025-2026**
Name of Student	**: P.Eshwar**
Enrollment No.	**: 2403A51L26**
Batch No.	**51**
Date	**: 24/02/26**

Submission Starts here

Screenshots:

Task Description #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.

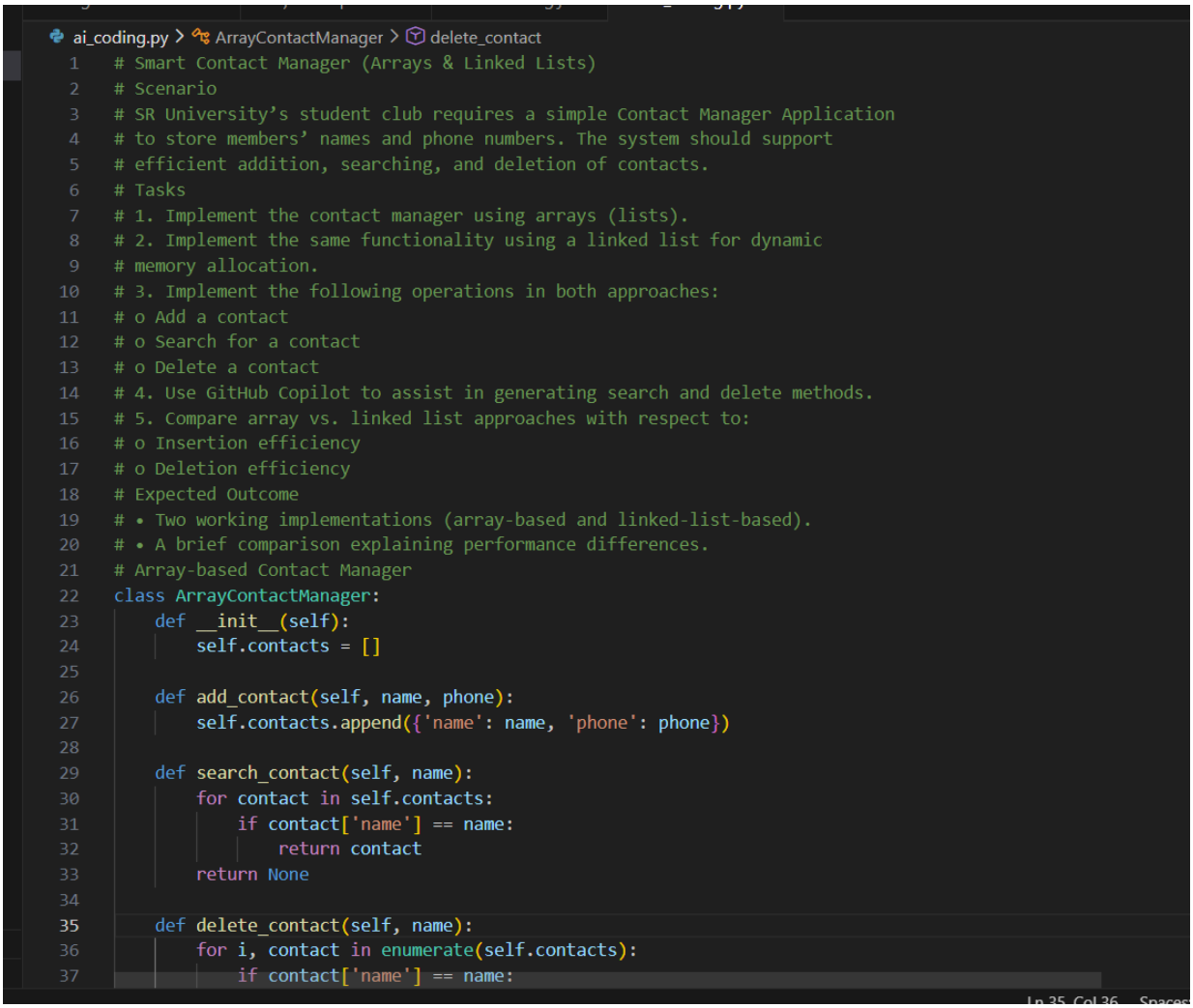
Tasks

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.

SR UNIVERSITY



ai_coding.py > ArrayContactManager > delete_contact

```
22  class ArrayContactManager:
35      def delete_contact(self, name):
36          for i, contact in enumerate(self.contacts):
37              if contact['name'] == name:
38                  del self.contacts[i]
39                  return True
40          return False
41  # Linked List-based Contact Manager
42  class Node:
43      def __init__(self, name, phone):
44          self.name = name
45          self.phone = phone
46          self.next = None
47
48  class LinkedListContactManager:
49      def __init__(self):
50          self.head = None
51
52      def add_contact(self, name, phone):
53          new_node = Node(name, phone)
54          new_node.next = self.head
55          self.head = new_node
56
57      def search_contact(self, name):
58          current = self.head
59          while current:
60              if current.name == name:
61                  return {'name': current.name, 'phone': current.phone}
62              current = current.next
63          return None
64
65      def delete_contact(self, name):
66          current = self.head
67          prev = None
68          while current:
69              if current.name == name:
70                  if prev:
```

```python
class LinkedListContactManager:
    def delete_contact(self, name):
        current = self.head
        prev = None
        while current:
            if current.name == name:
                if prev:
                    prev.next = current.next
                else:
                    self.head = current.next
                return True
            prev = current
            current = current.next
        return False
# 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 at the end).
# Overall, both approaches have similar time complexities for insertion and deletion, but the linked list can be more memory efficient for large dat

# Example Usage
if __name__ == "__main__":
    # Using ArrayContactManager
    array_manager = ArrayContactManager()
    array_manager.add_contact("Alice", "1234567890")
    array_manager.add_contact("Bob", "0987654321")
    print(array_manager.search_contact("Alice"))  # {'name': 'Alice', 'phone': '1234567890'}
    print(array_manager.delete_contact("Bob"))  # True
    print(array_manager.search_contact("Bob"))  # None

    # Using LinkedListContactManager
    linked_list_manager = LinkedListContactManager()
```

Output:

```python
    array_manager = ArrayContactManager()
    array_manager.add_contact("Alice", "1234567890")
    array_manager.add_contact("Bob", "0987654321")
    print(array_manager.search_contact("Alice"))  # {'name': 'Alice', 'phone': '1234567890'}
    print(array_manager.delete_contact("Bob"))  # True
    print(array_manager.search_contact("Bob"))  # None

    # Using LinkedListContactManager
    linked_list_manager = LinkedListContactManager()
    linked_list_manager.add_contact("Charlie", "5555555555")
    linked_list_manager.add_contact("Dave", "4444444444")
    print(linked_list_manager.search_contact("Charlie"))  # {'name': 'Charlie', 'phone': '5555555555'}
    print(linked_list_manager.delete_contact("Dave"))  # True
    print(linked_list_manager.search_contact("Dave"))  # None
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python/ai_coding.py
{'name': 'Alice', 'phone': '1234567890'}
True
None
{'name': 'Charlie', 'phone': '5555555555'}
True
None
PS C:\Users\Eshwar\OneDrive\Desktop\python>
```

EXPLAINATION:

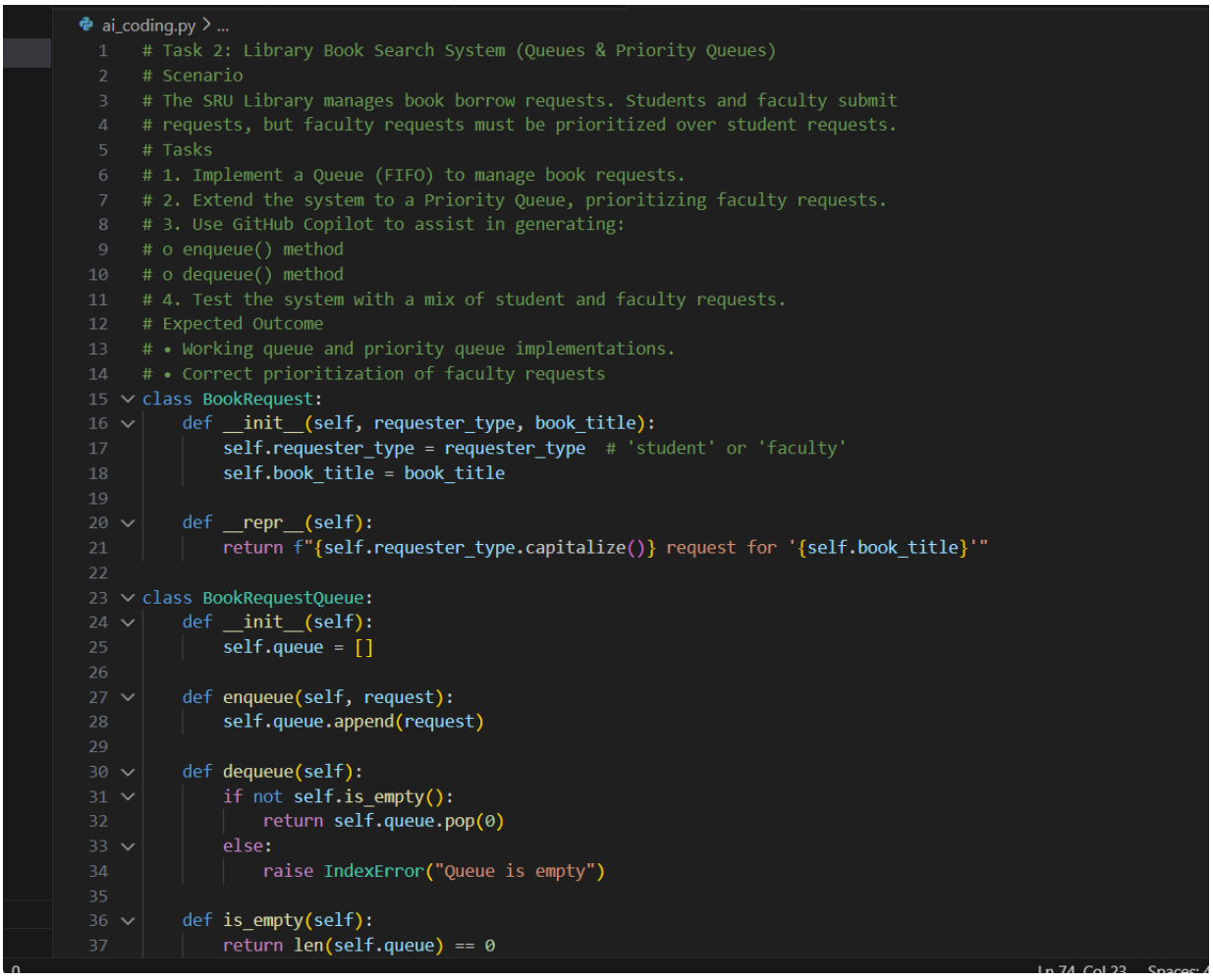
1. Array version is simple and memory efficient.

2. Linked list version is flexible and faster for insertions/deletions (especially at beginning).

3. Both structures take O(n) time for searching.

SR UNIVERSITY

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



ai_coding.py > ...

```
class BookRequestQueue:
        return len(self.queue) == 0
class PriorityBookRequestQueue:
    def __init__(self):
        self.faculty_queue = []
        self.student_queue = []

    def enqueue(self, request):
        if request.requester_type == 'faculty':
            self.faculty_queue.append(request)
        else:
            self.student_queue.append(request)

    def dequeue(self):
        if not self.is_empty():
            if self.faculty_queue:
                return self.faculty_queue.pop(0)
            else:
                return self.student_queue.pop(0)
        else:
            raise IndexError("Queue is empty")

    def is_empty(self):
        return len(self.faculty_queue) == 0 and len(self.student_queue) == 0
# Testing the system
if __name__ == "__main__":
    # Create a priority queue
    priority_queue = PriorityBookRequestQueue()

    # Enqueue some requests
    priority_queue.enqueue(BookRequest('student', 'Introduction to Algorithms'))
    priority_queue.enqueue(BookRequest('faculty', 'Advanced Machine Learning'))
    priority_queue.enqueue(BookRequest('student', 'Data Structures in Python'))
    priority_queue.enqueue(BookRequest('faculty', 'Artificial Intelligence: A Modern Approach'))

    # Dequeue and print requests
    while not priority_queue.is_empty():
```

Output:

```python
# Enqueue some requests
priority_queue.enqueue(BookRequest('student', 'Introduction to Algorithms'))
priority_queue.enqueue(BookRequest('faculty', 'Advanced Machine Learning'))
priority_queue.enqueue(BookRequest('student', 'Data Structures in Python'))
priority_queue.enqueue(BookRequest('faculty', 'Artificial Intelligence: A Modern Approach'))

# Dequeue and print requests
while not priority_queue.is_empty():
    request = priority_queue.dequeue()
    print(request)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/py
Faculty request for 'Advanced Machine Learning'
Faculty request for 'Artificial Intelligence: A Modern Approach'
Student request for 'Introduction to Algorithms'
Student request for 'Data Structures in Python'
PS C:\Users\Eshwar\OneDrive\Desktop\python>
```

EXPLAINATION:

1. Queue → serves requests in arrival order.
2. Priority Queue → serves based on importance.
3. Faculty are assigned **higher priority value**, so they are processed first.
4. A counter is used to maintain order when priorities are equal.

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

SR UNIVERSITY

• Functional stack-based ticket management system.
• Clear demonstration of LIFO behavior.

ai_coding.py > ...

```python
# 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.
class Stack:
    def __init__(self, capacity):
        self.stack = []
        self.capacity = capacity

    def push(self, ticket):
        if len(self.stack) < self.capacity:
            self.stack.append(ticket)
            print(f"Ticket '{ticket}' added to the stack.")
        else:
            print("Stack is full. Cannot add more tickets.")

    def pop(self):
        if not self.is_empty():
            ticket = self.stack.pop()
            print(f"Ticket '{ticket}' resolved and removed from the stack.")
            return ticket
        else:
            print("Stack is empty. No tickets to resolve.")
```

Ln 72, Col

ai_coding.py > ...

```
class Stack:

    def peek(self):
        if not self.is_empty():
            print(f"Next ticket to resolve: '{self.stack[-1]}'")
            return self.stack[-1]
        else:
            print("Stack is empty. No tickets to peek.")
            return None

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

    def is_full(self):
        return len(self.stack) >= self.capacity
# Simulating ticket management
help_desk = Stack(capacity=5)
help_desk.push("Ticket 1: Computer won't start")
help_desk.push("Ticket 2: Software installation issue")
help_desk.push("Ticket 3: Network connectivity problem")
help_desk.push("Ticket 4: Printer not working")
help_desk.push("Ticket 5: Email not syncing")
help_desk.peek()
help_desk.pop()
help_desk.pop()
help_desk.peek()
help_desk.push("Ticket 6: Password reset")
help_desk.peek()
help_desk.pop()
help_desk.pop()
help_desk.pop()
help_desk.pop()  # Attempting to pop from an empty stack
help_desk.is_empty()
help_desk.is_full()
```

Output:

```
Ticket 'Ticket 4: Printer not working' added to the stack.
Ticket 'Ticket 5: Email not syncing' added to the stack.
Next ticket to resolve: 'Ticket 5: Email not syncing'
Ticket 'Ticket 5: Email not syncing' resolved and removed from the stack.
Ticket 'Ticket 4: Printer not working' resolved and removed from the stack.
Next ticket to resolve: 'Ticket 3: Network connectivity problem'
Ticket 'Ticket 6: Password reset' added to the stack.
Next ticket to resolve: 'Ticket 6: Password reset'
Ticket 'Ticket 6: Password reset' resolved and removed from the stack.
Ticket 'Ticket 3: Network connectivity problem' resolved and removed from the stack.
Ticket 'Ticket 2: Software installation issue' resolved and removed from the stack.
Ticket 'Ticket 1: Computer won't start' resolved and removed from the stack.
```

EXPLAINATION:

1. Queue ensures fairness (first come, first served).
2. Priority Queue ensures importance (faculty first).
3. Both structures together simulate a realistic library request system.

SR UNIVERSITY

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

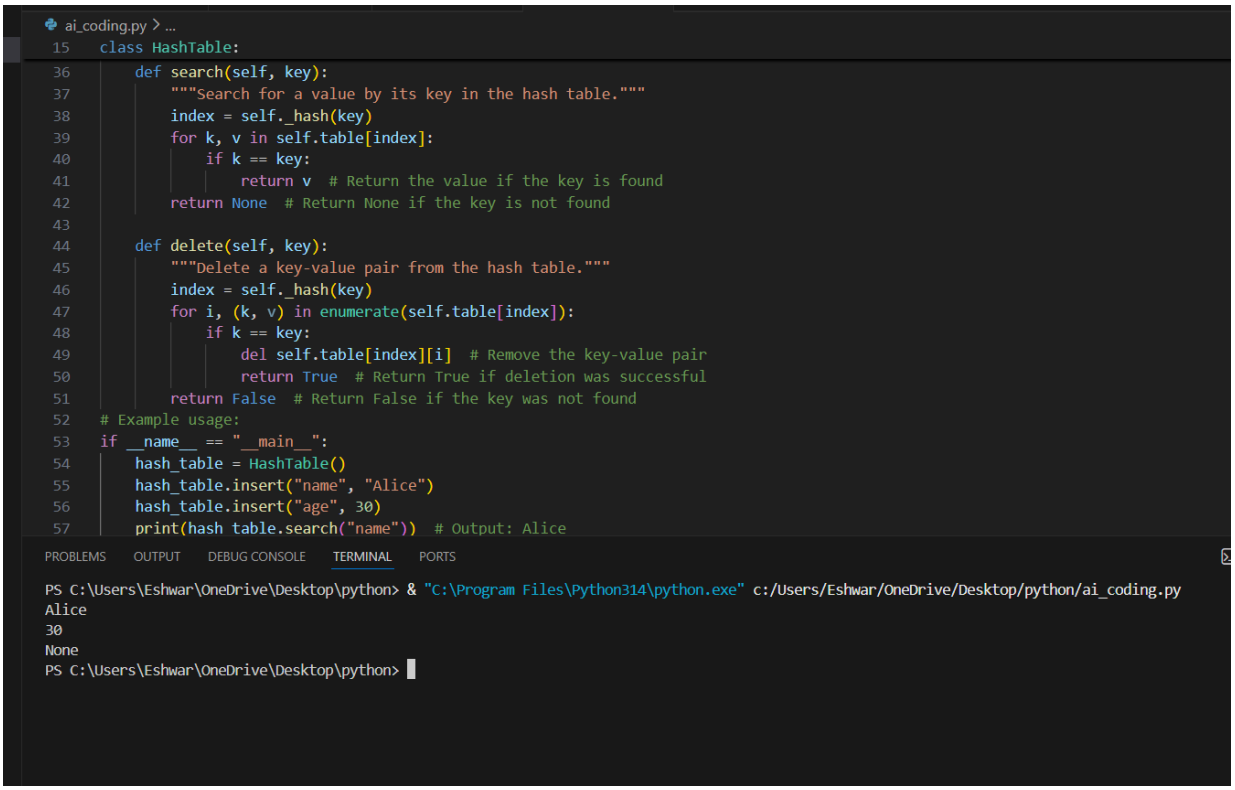
ai_coding.py > ...

```python
# 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
class HashTable:
    def __init__(self, size=10):
        """Initialize the hash table with a specified size."""
        self.size = size
        self.table = [[] for _ in range(size)]  # Create a list of empty lists for chaining

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

    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
        index = self._hash(key)
        # Check if the key already exists and update it
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        # If the key does not exist, add a new key-value pair
        self.table[index].append((key, value))

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



```python
class HashTable:
    def search(self, key):
        """Search for a value by its key in the hash table."""
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v  # Return the value if the key is found
        return None  # Return None if the key is not found

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

```
PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python/ai_coding.py
Alice
30
None
PS C:\Users\Eshwar\OneDrive\Desktop\python>
```

EXPLANATION:

1. Structure

A hash table uses an array of buckets. Each bucket stores a list of key–value pairs. A hash function converts a key into an index where data should be stored.

2. Collision Handling (Chaining)

If two keys map to the same index, they are stored together in that bucket's list instead of replacing each other. This method is called chaining.

3. Operations

- Insert: Hash key → go to bucket → add/update pair.
- Search: Hash key → check bucket list → return value if found.
- Delete: Hash key → find pair in bucket → remove it.

Task 5: Real-Time Application Challenge

Scenario

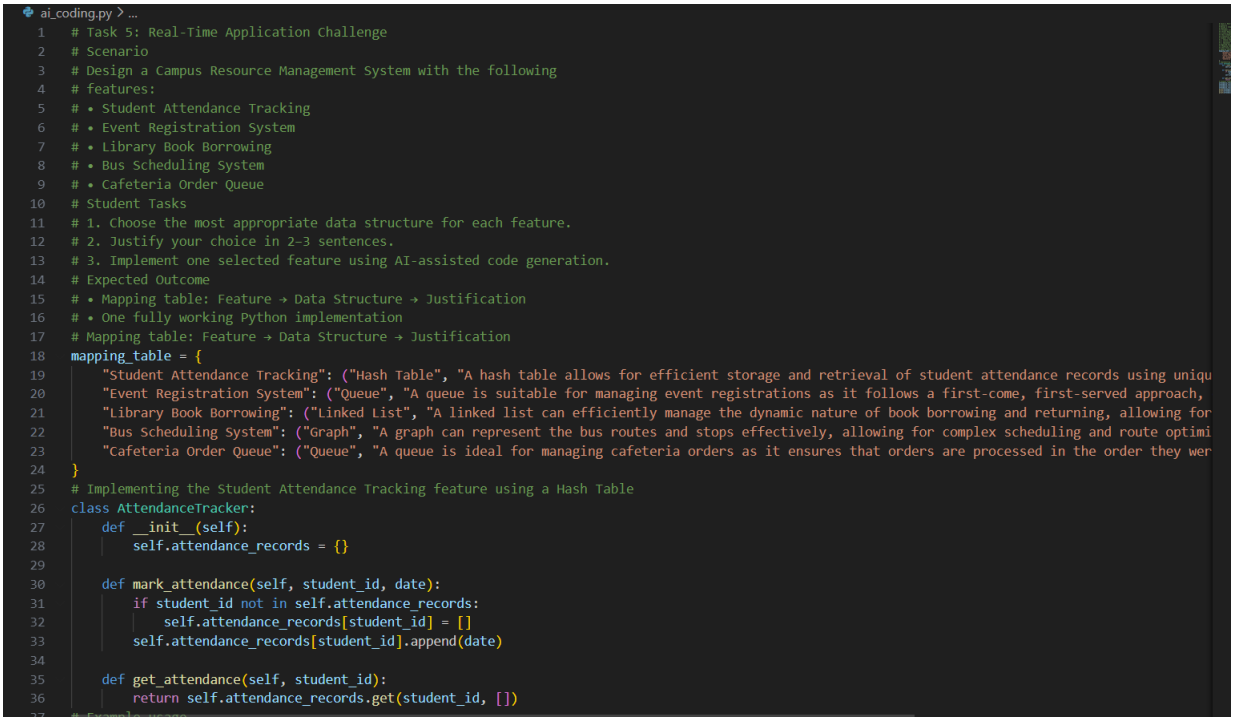
Design a Campus Resource Management System with the following features:

• 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.

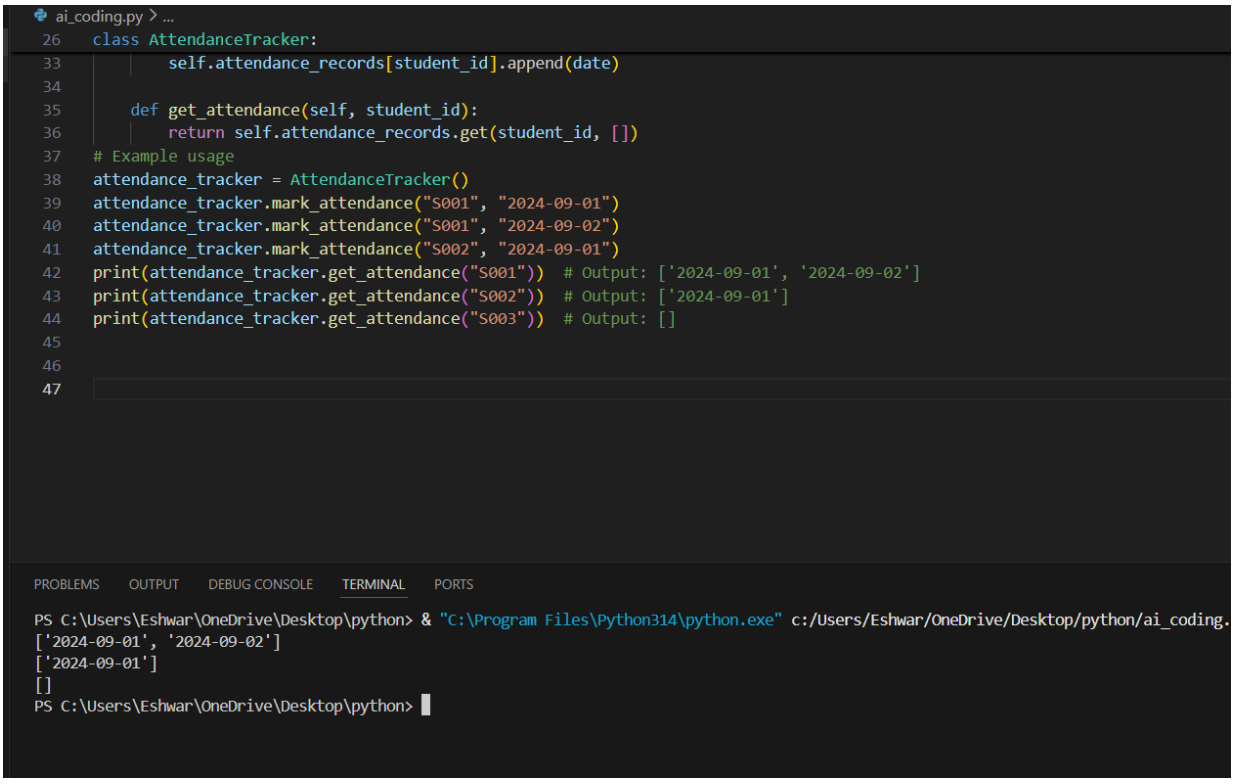
3. Implement one selected feature using AI-assisted code generation.
Expected Outcome
• Mapping table: Feature → Data Structure → Justification
• One fully working Python implementation



```python
# 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
# Mapping table: Feature → Data Structure → Justification
mapping_table = {
    "Student Attendance Tracking": ("Hash Table", "A hash table allows for efficient storage and retrieval of student attendance records using uniqu
    "Event Registration System": ("Queue", "A queue is suitable for managing event registrations as it follows a first-come, first-served approach,
    "Library Book Borrowing": ("Linked List", "A linked list can efficiently manage the dynamic nature of book borrowing and returning, allowing for
    "Bus Scheduling System": ("Graph", "A graph can represent the bus routes and stops effectively, allowing for complex scheduling and route optimi
    "Cafeteria Order Queue": ("Queue", "A queue is ideal for managing cafeteria orders as it ensures that orders are processed in the order they wer
}
# Implementing the Student Attendance Tracking feature using a Hash Table
class AttendanceTracker:
    def __init__(self):
        self.attendance_records = {}

    def mark_attendance(self, student_id, date):
        if student_id not in self.attendance_records:
            self.attendance_records[student_id] = []
        self.attendance_records[student_id].append(date)

    def get_attendance(self, student_id):
        return self.attendance_records.get(student_id, [])
```



```python
class AttendanceTracker:
            self.attendance_records[student_id].append(date)

    def get_attendance(self, student_id):
        return self.attendance_records.get(student_id, [])
# Example usage
attendance_tracker = AttendanceTracker()
attendance_tracker.mark_attendance("S001", "2024-09-01")
attendance_tracker.mark_attendance("S001", "2024-09-02")
attendance_tracker.mark_attendance("S002", "2024-09-01")
print(attendance_tracker.get_attendance("S001"))  # Output: ['2024-09-01', '2024-09-02']
print(attendance_tracker.get_attendance("S002"))  # Output: ['2024-09-01']
print(attendance_tracker.get_attendance("S003"))  # Output: []
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python/ai_coding.
['2024-09-01', '2024-09-02']
['2024-09-01']
[]
PS C:\Users\Eshwar\OneDrive\Desktop\python>
```

EXPLAINATION:
1. A hash table (dictionary in Python) is chosen because it provides O(1) average time complexity for both insertion and retrieval operations.

2. Each student ID is used as a key in the hash table, and the value is a list of attendance dates for that student.
3. This approach allows for efficient tracking of attendance records without needing to scan through all records every time a lookup is performed, making it scalable for large numbers of students.