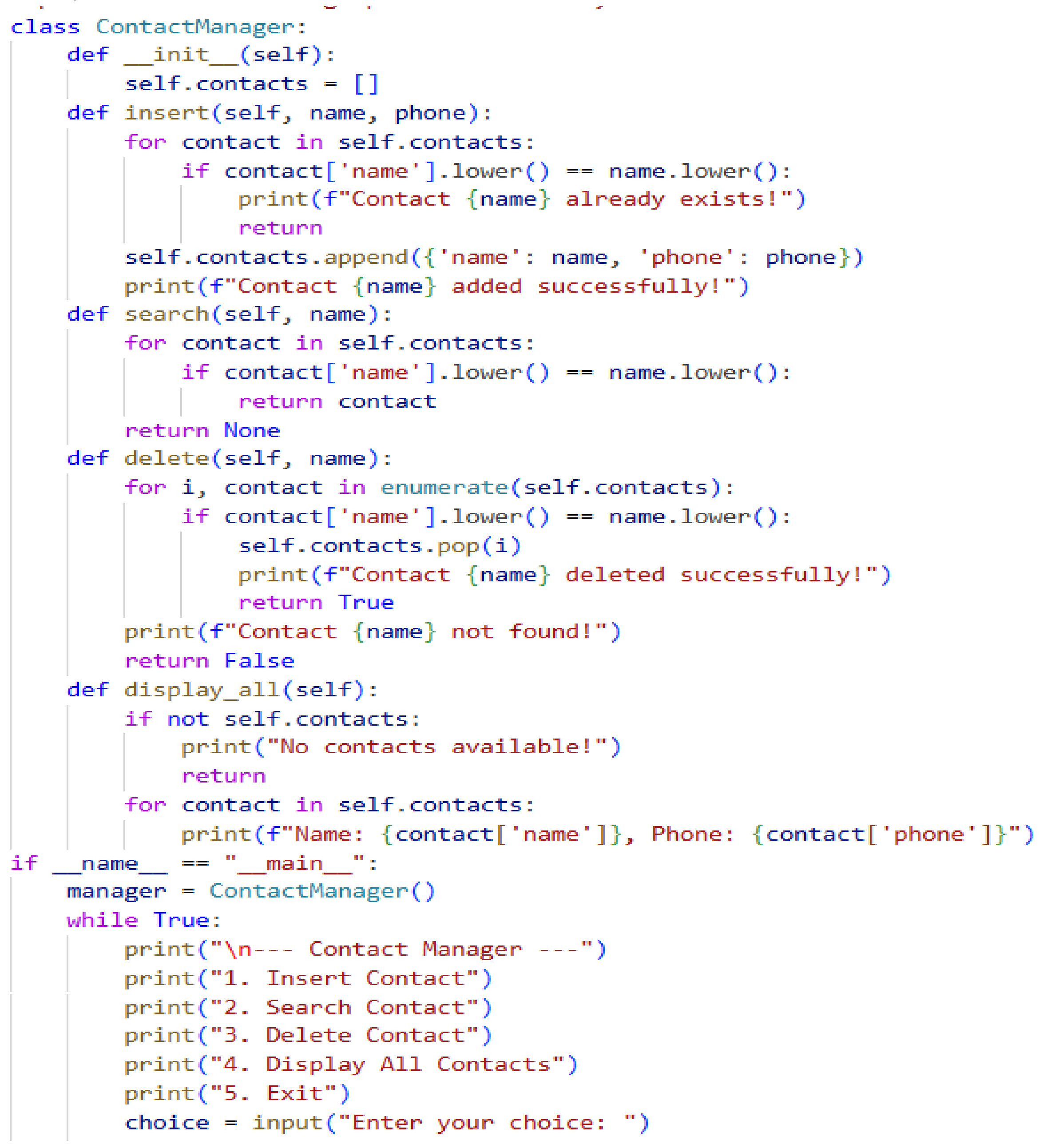
Lab Assignment 11.3

Program	**: B. Tech (CSE)**
Course Title	**: AI Assisted Coding**
Course Code	**: 23CS002PC304**
Semester	**: III**
Academic Session	**: 2025-2026**
Name of Student	**: Sruthi**
Enrollment No.	**: 2403A51L10**
Batch No.	**: 51**
Date	**: 24-02-2026**

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

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. Implement the following operations insert, delete and search a contact and implement the contact manager using arrays and also using linked list.

Array Code:



```
class ContactManager:
    def __init__(self):
        self.contacts = []
    def insert(self, name, phone):
        for contact in self.contacts:
            if contact['name'].lower() == name.lower():
                print(f"Contact {name} already exists!")
                return
        self.contacts.append({'name': name, 'phone': phone})
        print(f"Contact {name} added successfully!")
    def search(self, name):
        for contact in self.contacts:
            if contact['name'].lower() == name.lower():
                return contact
        return None
    def delete(self, name):
        for i, contact in enumerate(self.contacts):
            if contact['name'].lower() == name.lower():
                self.contacts.pop(i)
                print(f"Contact {name} deleted successfully!")
                return True
        print(f"Contact {name} not found!")
        return False
    def display_all(self):
        if not self.contacts:
            print("No contacts available!")
            return
        for contact in self.contacts:
            print(f"Name: {contact['name']}, Phone: {contact['phone']}")
if __name__ == "__main__":
    manager = ContactManager()
    while True:
        print("\n--- Contact Manager ---")
        print("1. Insert Contact")
        print("2. Search Contact")
        print("3. Delete Contact")
        print("4. Display All Contacts")
        print("5. Exit")
        choice = input("Enter your choice: ")
```

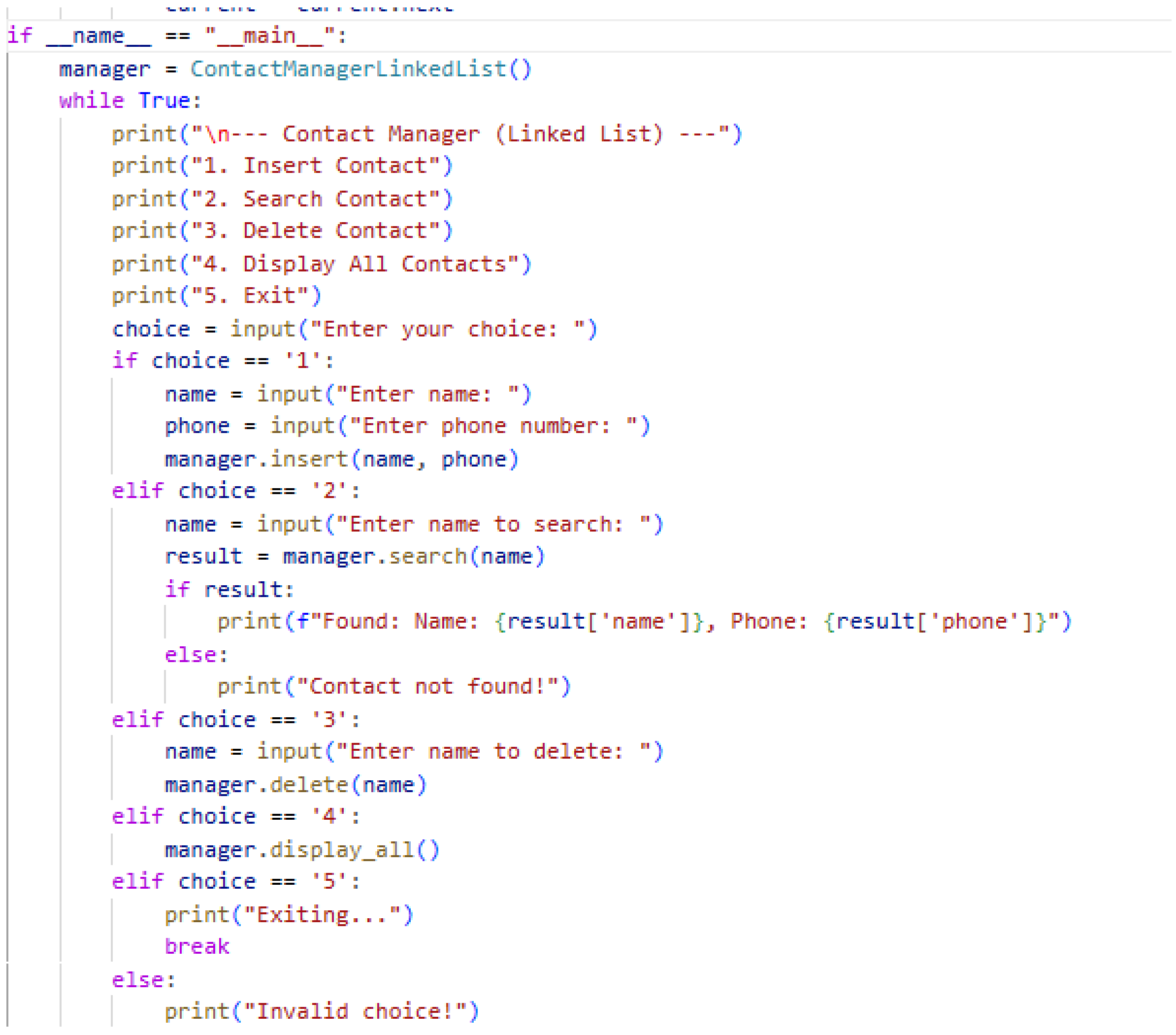
```python
        if choice == '1':
            name = input("Enter name: ")
            phone = input("Enter phone number: ")
            manager.insert(name, phone)
        elif choice == '2':
            name = input("Enter name to search: ")
            result = manager.search(name)
            if result:
                print(f"Found: Name: {result['name']}, Phone: {result['phone']}")
            else:
                print("Contact not found!")
        elif choice == '3':
            name = input("Enter name to delete: ")
            manager.delete(name)
        elif choice == '4':
            manager.display_all()
        elif choice == '5':
            print("Exiting...")
            break
        else:
            print("Invalid choice!")
```

Output:

```
--- Contact Manager ---
1. Insert Contact
2. Search Contact
3. Delete Contact
4. Display All Contacts
5. Exit
Enter your choice: 1
Enter name: pinky
Enter phone number: 1234534223
Contact pinky added successfully!

--- Contact Manager ---
1. Insert Contact
2. Search Contact
3. Delete Contact
4. Display All Contacts
5. Exit
Enter your choice: 1
Enter name: chotu
Enter phone number: 2342354231
Contact chotu added successfully!

--- Contact Manager ---
1. Insert Contact
2. Search Contact
3. Delete Contact
4. Display All Contacts
5. Exit
Enter your choice: 2
Enter name to search: chotu
Found: Name: chotu, Phone: 2342354231

--- Contact Manager ---
1. Insert Contact
2. Search Contact
3. Delete Contact
4. Display All Contacts
5. Exit
Enter your choice: 3
Enter name to delete: pinky
Contact pinky deleted successfully!
```

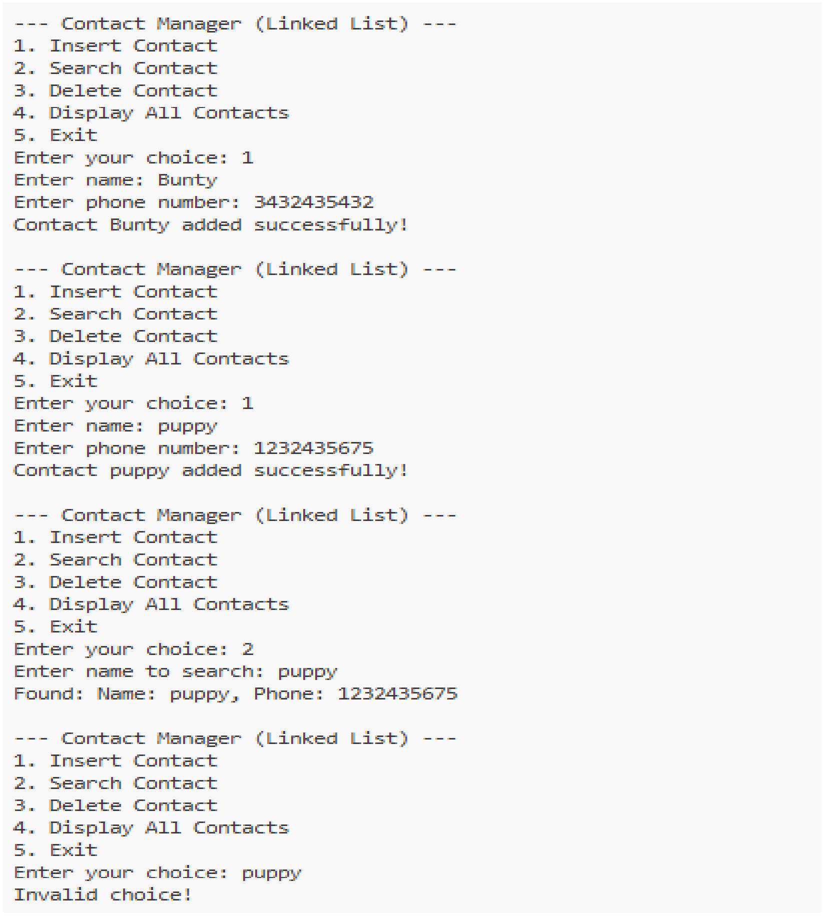
Linked list Code:

```python
class Node:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
        self.next = None
class ContactManagerLinkedList:
    def __init__(self):
        self.head = None
    def insert(self, name, phone):
        current = self.head
        while current:
            if current.name.lower() == name.lower():
                print(f"Contact {name} already exists!")
                return
            current = current.next
        new_node = Node(name, phone)
        new_node.next = self.head
        self.head = new_node
        print(f"Contact {name} added successfully!")
    def search(self, name):
        current = self.head
        while current:
            if current.name.lower() == name.lower():
                return {'name': current.name, 'phone': current.phone}
            current = current.next
        return None
    def delete(self, name):
        if not self.head:
            print(f"Contact {name} not found!")
            return False
        if self.head.name.lower() == name.lower():
            self.head = self.head.next
            print(f"Contact {name} deleted successfully!")
            return True
        current = self.head
        while current.next:
            if current.next.name.lower() == name.lower():
                current.next = current.next.next
                print(f"Contact {name} deleted successfully!")
                return True
            current = current.next
        print(f"Contact {name} not found!")
        return False
    def display_all(self):
        if not self.head:
            print("No contacts available!")
            return
        current = self.head
        while current:
            print(f"Name: {current.name}, Phone: {current.phone}")
            current = current.next
```



```
if __name__ == "__main__":
    manager = ContactManagerLinkedList()
    while True:
        print("\n--- Contact Manager (Linked List) ---")
        print("1. Insert Contact")
        print("2. Search Contact")
        print("3. Delete Contact")
        print("4. Display All Contacts")
        print("5. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            name = input("Enter name: ")
            phone = input("Enter phone number: ")
            manager.insert(name, phone)
        elif choice == '2':
            name = input("Enter name to search: ")
            result = manager.search(name)
            if result:
                print(f"Found: Name: {result['name']}, Phone: {result['phone']}")
            else:
                print("Contact not found!")
        elif choice == '3':
            name = input("Enter name to delete: ")
            manager.delete(name)
        elif choice == '4':
            manager.display_all()
        elif choice == '5':
            print("Exiting...")
            break
        else:
            print("Invalid choice!")
```

Output:



```
--- Contact Manager (Linked List) ---
1. Insert Contact
2. Search Contact
3. Delete Contact
4. Display All Contacts
5. Exit
Enter your choice: 1
Enter name: Bunty
Enter phone number: 3432435432
Contact Bunty added successfully!

--- Contact Manager (Linked List) ---
1. Insert Contact
2. Search Contact
3. Delete Contact
4. Display All Contacts
5. Exit
Enter your choice: 1
Enter name: puppy
Enter phone number: 1232435675
Contact puppy added successfully!

--- Contact Manager (Linked List) ---
1. Insert Contact
2. Search Contact
3. Delete Contact
4. Display All Contacts
5. Exit
Enter your choice: 2
Enter name to search: puppy
Found: Name: puppy, Phone: 1232435675

--- Contact Manager (Linked List) ---
1. Insert Contact
2. Search Contact
3. Delete Contact
4. Display All Contacts
5. Exit
Enter your choice: puppy
Invalid choice!
```

Comparison Table:

Aspect	List	Linked List
Memory	Contiguous	Non-contiguous
Deletion	Slower	Faster
Insertion at beginning	O (n)	O (1)
Access by index	O (1)	Not possible

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

Prompt: The SRU Library manages book borrow requests. Students and faculty submit requests, but faculty requests must be prioritized over student requests. Implement a Queue (FIFO) to manage book requests and extend the system to a Priority Queue, prioritizing faculty requests. Generate a python program that perform enqueue () method and dequeue ().

Queue Code:

```python
class Queue:
    def __init__(self):
        self.queue = []
    def enqueue(self, request):
        self.queue.append(request)
        print(f"Enqueued: {request}")
    def dequeue(self):
        if not self.is_empty():
            removed = self.queue.pop(0)
            print(f"Dequeued: {removed}")
            return removed
        else:
            print("Queue is empty!")
            return None
    def is_empty(self):
        return len(self.queue) == 0
    def display(self):
        print("Current Queue:", self.queue)
print("----- Normal Queue (FIFO) -----")
q = Queue()
q.enqueue("Student - Book A")
q.enqueue("Faculty - Book B")
q.enqueue("Student - Book C")
q.display()
q.dequeue()
q.display()
```

Output:

```
----- Normal Queue (FIFO) -----
Enqueued: Student - Book A
Enqueued: Faculty - Book B
Enqueued: Student - Book C
Current Queue: ['Student - Book A', 'Faculty - Book B', 'Student - Book C']
Dequeued: Student - Book A
Current Queue: ['Faculty - Book B', 'Student - Book C']
```

Priority Queue Code:

```python
class PriorityQueue:
    def __init__(self):
        self.faculty_queue = []
        self.student_queue = []
    def enqueue(self, name, role, book):
        request = f"{role} - {name} requested {book}"
        if role.lower() == "faculty":
            self.faculty_queue.append(request)
            print(f"Enqueued (Faculty Priority): {request}")
        else:
            self.student_queue.append(request)
            print(f"Enqueued (Student): {request}")
    def dequeue(self):
        if self.faculty_queue:
            removed = self.faculty_queue.pop(0)
            print(f"Dequeued (Faculty): {removed}")
            return removed
        elif self.student_queue:
            removed = self.student_queue.pop(0)
            print(f"Dequeued (Student): {removed}")
            return removed
        else:
            print("No requests in queue!")
            return None
    def display(self):
        print("Faculty Queue:", self.faculty_queue)
        print("Student Queue:", self.student_queue)
print("\n----- Priority Queue (Faculty First) -----")
pq = PriorityQueue()
pq.enqueue("Anitha", "Student", "Data Structures")
pq.enqueue("Dr. Rao", "Faculty", "Operating Systems")
pq.enqueue("Kiran", "Student", "Python Programming")
pq.enqueue("Dr. Sharma", "Faculty", "Algorithms")
pq.display()
pq.dequeue()
pq.dequeue()
pq.display()
```

Output:

```
----- Priority Queue (Faculty First) -----
Enqueued (Student): Student - Anitha requested Data Structures
Enqueued (Faculty Priority): Faculty - Dr. Rao requested Operating Systems
Enqueued (Student): Student - Kiran requested Python Programming
Enqueued (Faculty Priority): Faculty - Dr. Sharma requested Algorithms
Faculty Queue: ['Faculty - Dr. Rao requested Operating Systems', 'Faculty - Dr. Sharma requested Algorithms']
Student Queue: ['Student - Anitha requested Data Structures', 'Student - Kiran requested Python Programming']
Dequeued (Faculty): Faculty - Dr. Rao requested Operating Systems
Dequeued (Faculty): Faculty - Dr. Sharma requested Algorithms
Faculty Queue: []
Student Queue: ['Student - Anitha requested Data Structures', 'Student - Kiran requested Python Programming']
```

Explanation:

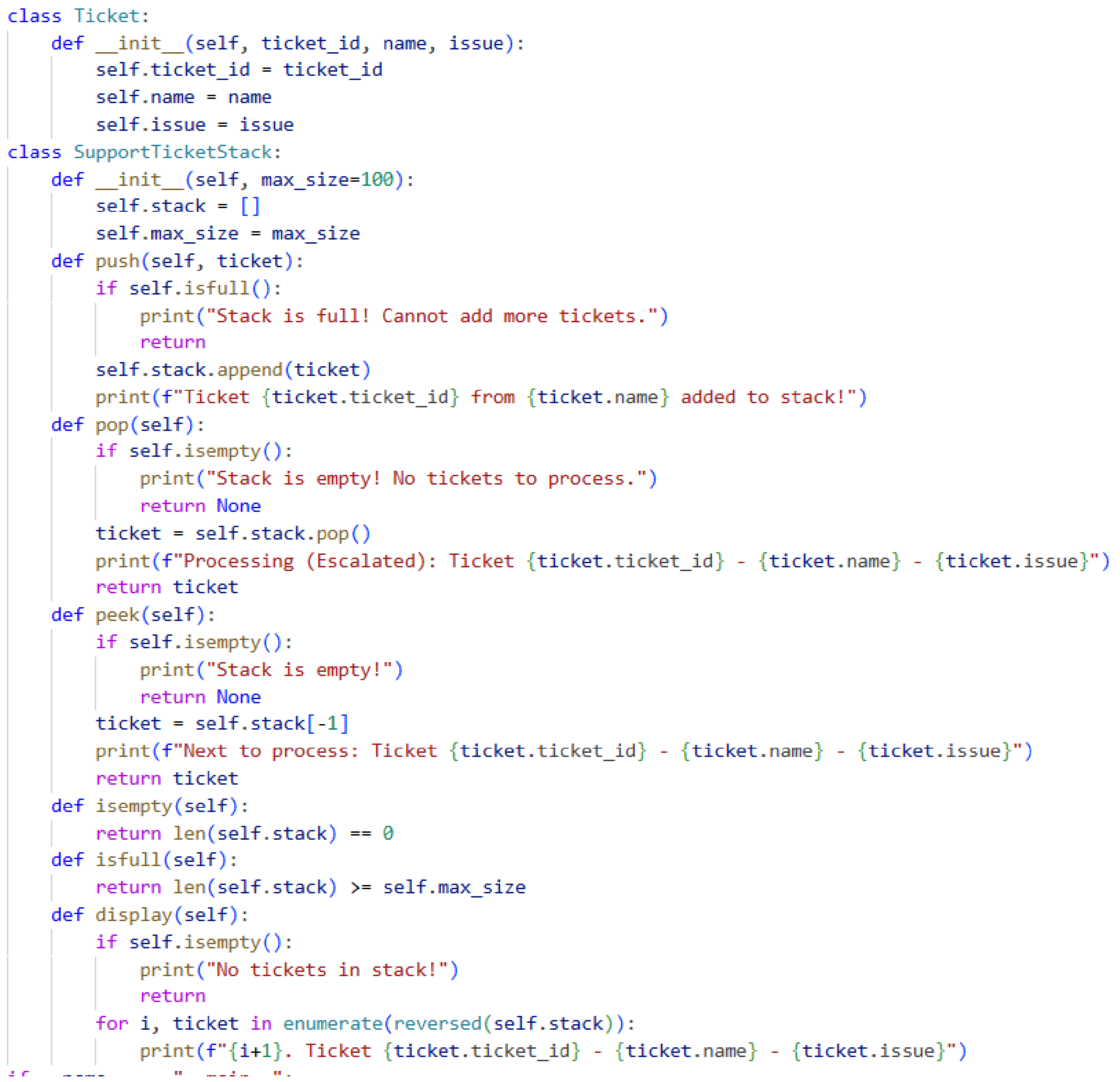
A Queue is a linear data structure that follows the FIFO (First In First Out) principle.
The element inserted first will be removed first.
A Priority Queue removes elements based on priority rather than insertion order.
In this program:

- Faculty → High Priority
- Student → Low Priority

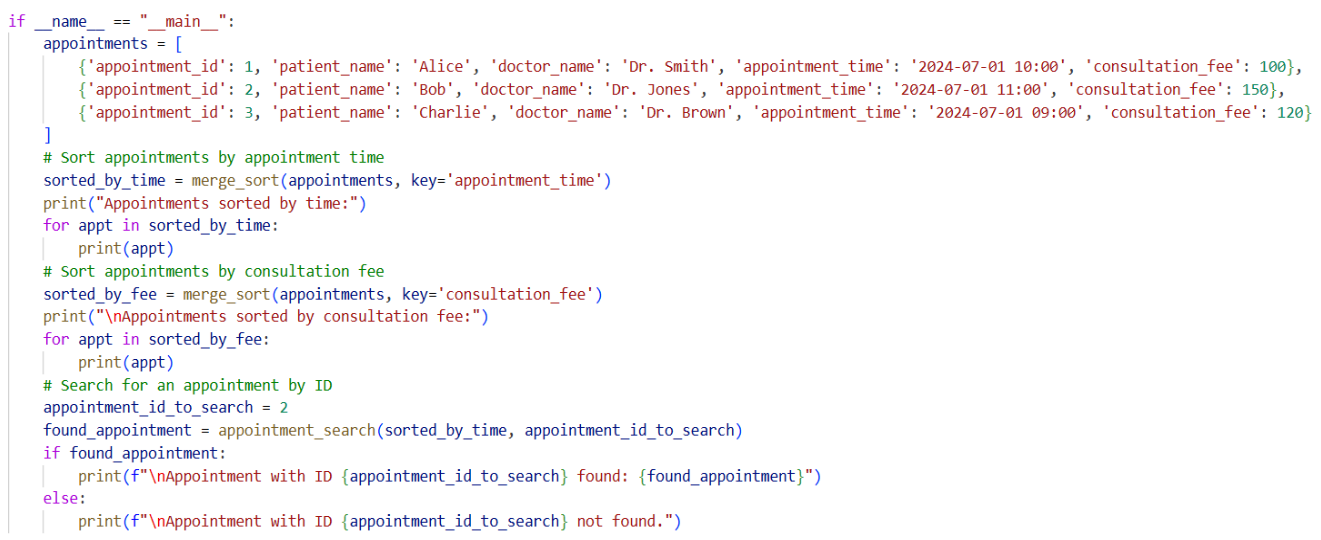
Task 3: Emergency Help Desk (Stack Implementation)

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 and implement the push (), pop (), peek () operations and also perform isempty () and isfull () operations

Code:



```python
class Ticket:
    def __init__(self, ticket_id, name, issue):
        self.ticket_id = ticket_id
        self.name = name
        self.issue = issue
class SupportTicketStack:
    def __init__(self, max_size=100):
        self.stack = []
        self.max_size = max_size
    def push(self, ticket):
        if self.isfull():
            print("Stack is full! Cannot add more tickets.")
            return
        self.stack.append(ticket)
        print(f"Ticket {ticket.ticket_id} from {ticket.name} added to stack!")
    def pop(self):
        if self.isempty():
            print("Stack is empty! No tickets to process.")
            return None
        ticket = self.stack.pop()
        print(f"Processing (Escalated): Ticket {ticket.ticket_id} - {ticket.name} - {ticket.issue}")
        return ticket
    def peek(self):
        if self.isempty():
            print("Stack is empty!")
            return None
        ticket = self.stack[-1]
        print(f"Next to process: Ticket {ticket.ticket_id} - {ticket.name} - {ticket.issue}")
        return ticket
    def isempty(self):
        return len(self.stack) == 0
    def isfull(self):
        return len(self.stack) >= self.max_size
    def display(self):
        if self.isempty():
            print("No tickets in stack!")
            return
        for i, ticket in enumerate(reversed(self.stack)):
            print(f"{i+1}. Ticket {ticket.ticket_id} - {ticket.name} - {ticket.issue}")
```



```python
if __name__ == "__main__":
    appointments = [
        {'appointment_id': 1, 'patient_name': 'Alice', 'doctor_name': 'Dr. Smith', 'appointment_time': '2024-07-01 10:00', 'consultation_fee': 100},
        {'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150},
        {'appointment_id': 3, 'patient_name': 'Charlie', 'doctor_name': 'Dr. Brown', 'appointment_time': '2024-07-01 09:00', 'consultation_fee': 120}
    ]
    # Sort appointments by appointment time
    sorted_by_time = merge_sort(appointments, key='appointment_time')
    print("Appointments sorted by time:")
    for appt in sorted_by_time:
        print(appt)
    # Sort appointments by consultation fee
    sorted_by_fee = merge_sort(appointments, key='consultation_fee')
    print("\nAppointments sorted by consultation fee:")
    for appt in sorted_by_fee:
        print(appt)
    # Search for an appointment by ID
    appointment_id_to_search = 2
    found_appointment = appointment_search(sorted_by_time, appointment_id_to_search)
    if found_appointment:
        print(f"\nAppointment with ID {appointment_id_to_search} found: {found_appointment}")
    else:
        print(f"\nAppointment with ID {appointment_id_to_search} not found.")
```

```python
if __name__ == "__main__":
    stack = SupportTicketStack(max_size=50)
    ticket_counter = 1
    while True:
        print("\n--- IT Help Desk Support Ticket Stack ---")
        print("1. Add Ticket (Push)")
        print("2. Process Ticket (Pop)")
        print("3. View Next Ticket (Peek)")
        print("4. Check if Empty")
        print("5. Check if Full")
        print("6. Display All Tickets")
        print("7. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            name = input("Enter name: ")
            issue = input("Enter issue description: ")
            ticket = Ticket(ticket_counter, name, issue)
            stack.push(ticket)
            ticket_counter += 1
        elif choice == '2':
            stack.pop()
        elif choice == '3':
            stack.peek()
        elif choice == '4':
            print("Stack is empty!" if stack.isempty() else "Stack is not empty!")
        elif choice == '5':
            print("Stack is full!" if stack.isfull() else "Stack is not full!")
        elif choice == '6':
            stack.display()
        elif choice == '7':
            print("Exiting...")
            break
        else:
            print("Invalid choice!")
```

Output:

```
--- IT Help Desk Support Ticket Stack ---
1. Add Ticket (Push)
2. Process Ticket (Pop)
3. View Next Ticket (Peek)
4. Check if Empty
5. Check if Full
6. Display All Tickets
7. Exit
Enter your choice: 1
Enter name: Sruthi
Enter issue description: wifi problem
Ticket 1 from Sruthi added to stack!

--- IT Help Desk Support Ticket Stack ---
1. Add Ticket (Push)
2. Process Ticket (Pop)
3. View Next Ticket (Peek)
4. Check if Empty
5. Check if Full
6. Display All Tickets
7. Exit
Enter your choice: 1
Enter name: Ramu
Enter issue description: login issue
Ticket 2 from Ramu added to stack!

--- IT Help Desk Support Ticket Stack ---
1. Add Ticket (Push)
2. Process Ticket (Pop)
3. View Next Ticket (Peek)
4. Check if Empty
5. Check if Full
6. Display All Tickets
7. Exit
Enter your choice: 2
Processing (Escalated): Ticket 2 - Ramu - login issue
```

Explanation:

This program implements a Stack data structure for managing support tickets using the LIFO (Last in First Out) principle. Tickets are added using push () and processed using pop (), meaning the most recently added ticket is handled first. All stack operations like push, pop, peek, isEmpty, and isFull are implemented efficiently using a Python list.

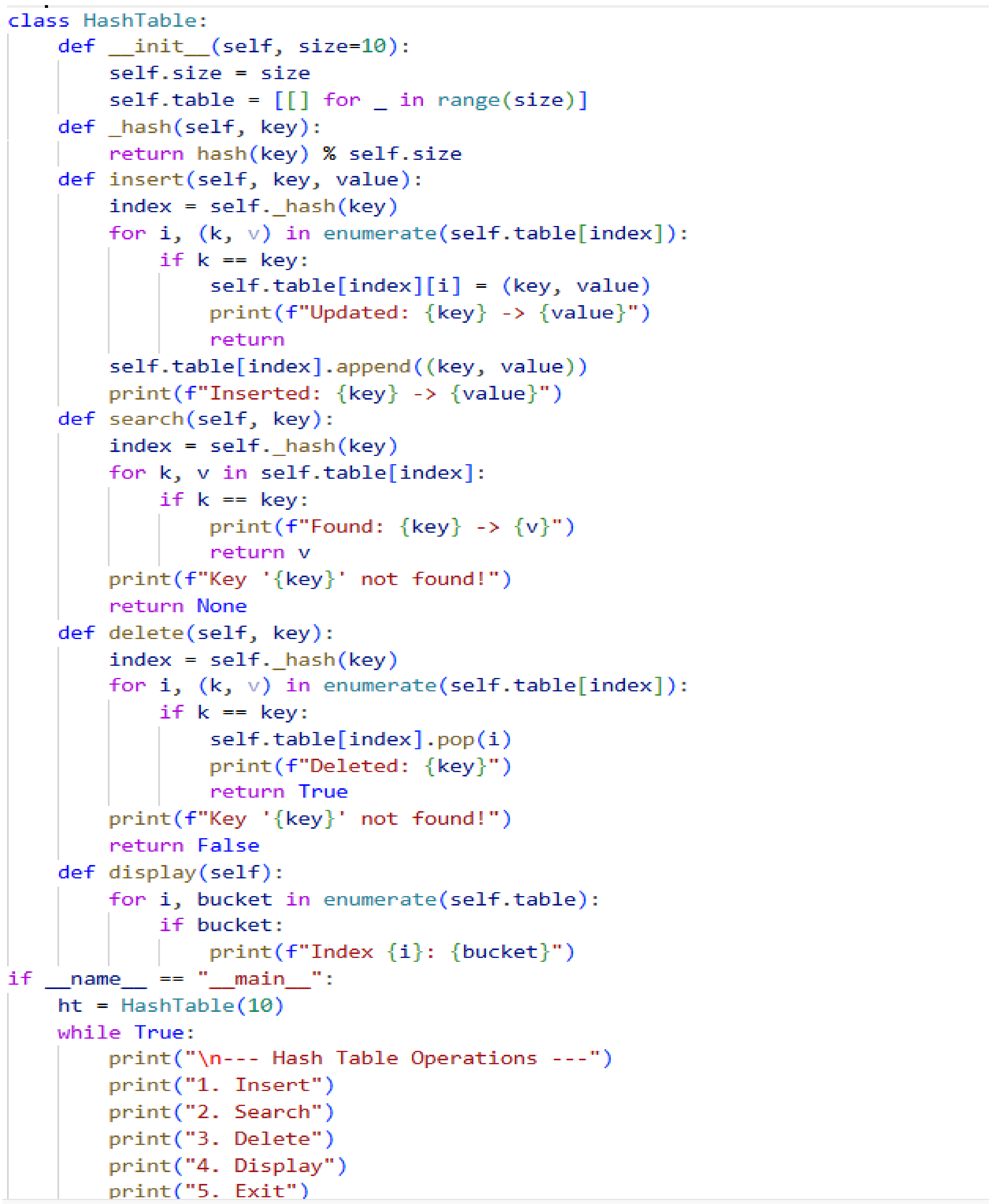
Task 4: Hash Table

Prompt: Implement a Hash table with Insert, Delete, and search operations

Sample code:

```
class HashTable:
    pass
```

Code:



```
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        return hash(key) % self.size
    def insert(self, key, value):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                print(f"Updated: {key} -> {value}")
                return
        self.table[index].append((key, value))
        print(f"Inserted: {key} -> {value}")
    def search(self, key):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                print(f"Found: {key} -> {v}")
                return v
        print(f"Key '{key}' not found!")
        return None
    def delete(self, key):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index].pop(i)
                print(f"Deleted: {key}")
                return True
        print(f"Key '{key}' not found!")
        return False
    def display(self):
        for i, bucket in enumerate(self.table):
            if bucket:
                print(f"Index {i}: {bucket}")
if __name__ == "__main__":
    ht = HashTable(10)
    while True:
        print("\n--- Hash Table Operations ---")
        print("1. Insert")
        print("2. Search")
        print("3. Delete")
        print("4. Display")
        print("5. Exit")
```

```python
    print("5. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        key = input("Enter key: ")
        value = input("Enter value: ")
        ht.insert(key, value)
    elif choice == '2':
        key = input("Enter key to search: ")
        ht.search(key)
    elif choice == '3':
        key = input("Enter key to delete: ")
        ht.delete(key)
    elif choice == '4':
        ht.display()
    elif choice == '5':
        print("Exiting...")
        break
    else:
        print("Invalid choice!")
```

Output:

```
--- Hash Table Operations ---
1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter your choice: 1
Enter key: 23
Enter value: bunty
Inserted: 23 -> bunty

--- Hash Table Operations ---
1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter your choice: 1
Enter key: 234
Enter value: Chotti
Inserted: 234 -> Chotti

--- Hash Table Operations ---
1. Insert
2. Search
3. Delete
4. Display
5. Exit
Enter your choice: 3
Enter key to delete: 1
Key '1' not found!
```

Explanation:

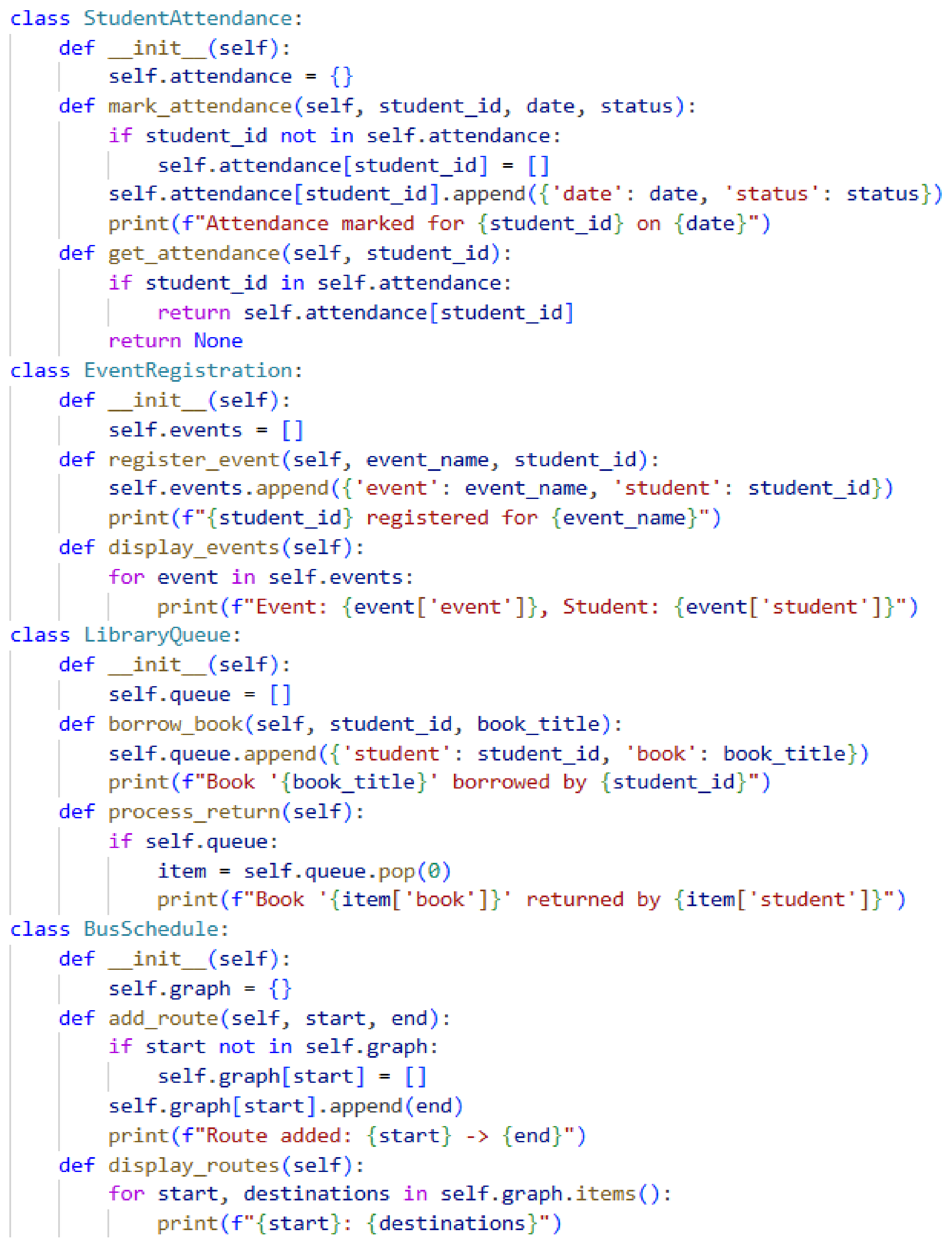
This program implements a Hash Table data structure using separate chaining to handle collisions. Keys are converted into an index using a hash function, and key–value pairs are stored in buckets (lists). It supports insert, search, delete, and display operations with average time complexity of O (1).

Task 5: Real-Time Application Challenge

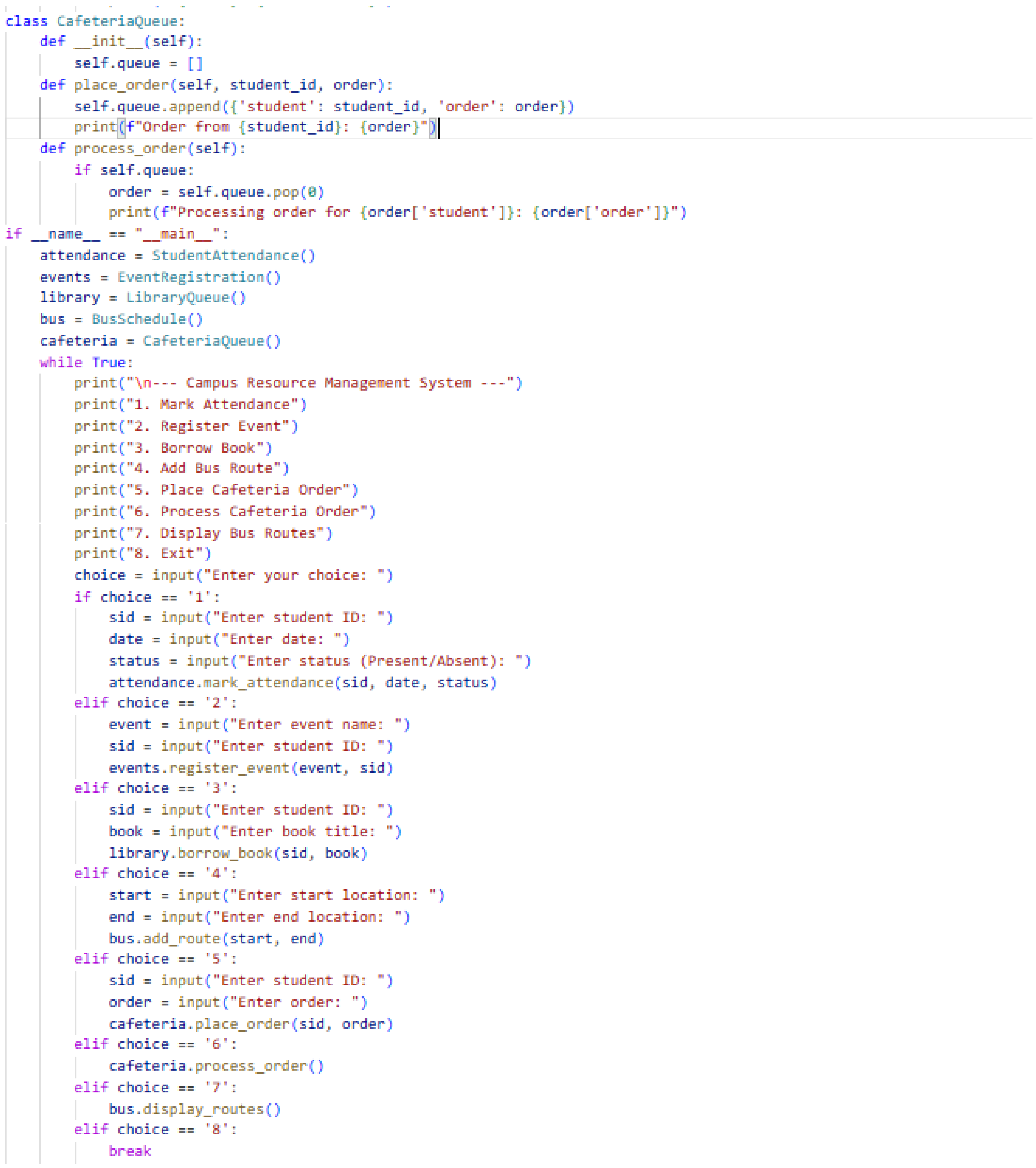
Prompt: Design a Campus Resource Management System with the following features:

• Student Attendance Tracking: Use Hash Table for tracking student attendance
• Event Registration System: Use List for Event Registration System
• Library Book Borrowing: Use Library for Book Borrowing
• Bus Scheduling System: Use Graph for Bus Scheduling System
• Cafeteria Order Queue: Use Queue for Cafeteria Order Queue

Code:



```python
class StudentAttendance:
    def __init__(self):
        self.attendance = {}
    def mark_attendance(self, student_id, date, status):
        if student_id not in self.attendance:
            self.attendance[student_id] = []
        self.attendance[student_id].append({'date': date, 'status': status})
        print(f"Attendance marked for {student_id} on {date}")
    def get_attendance(self, student_id):
        if student_id in self.attendance:
            return self.attendance[student_id]
        return None
class EventRegistration:
    def __init__(self):
        self.events = []
    def register_event(self, event_name, student_id):
        self.events.append({'event': event_name, 'student': student_id})
        print(f"{student_id} registered for {event_name}")
    def display_events(self):
        for event in self.events:
            print(f"Event: {event['event']}, Student: {event['student']}")
class LibraryQueue:
    def __init__(self):
        self.queue = []
    def borrow_book(self, student_id, book_title):
        self.queue.append({'student': student_id, 'book': book_title})
        print(f"Book '{book_title}' borrowed by {student_id}")
    def process_return(self):
        if self.queue:
            item = self.queue.pop(0)
            print(f"Book '{item['book']}' returned by {item['student']}")
class BusSchedule:
    def __init__(self):
        self.graph = {}
    def add_route(self, start, end):
        if start not in self.graph:
            self.graph[start] = []
        self.graph[start].append(end)
        print(f"Route added: {start} -> {end}")
    def display_routes(self):
        for start, destinations in self.graph.items():
            print(f"{start}: {destinations}")
```



```python
class CafeteriaQueue:
    def __init__(self):
        self.queue = []
    def place_order(self, student_id, order):
        self.queue.append({'student': student_id, 'order': order})
        print(f"Order from {student_id}: {order}")
    def process_order(self):
        if self.queue:
            order = self.queue.pop(0)
            print(f"Processing order for {order['student']}: {order['order']}")
if __name__ == "__main__":
    attendance = StudentAttendance()
    events = EventRegistration()
    library = LibraryQueue()
    bus = BusSchedule()
    cafeteria = CafeteriaQueue()
    while True:
        print("\n--- Campus Resource Management System ---")
        print("1. Mark Attendance")
        print("2. Register Event")
        print("3. Borrow Book")
        print("4. Add Bus Route")
        print("5. Place Cafeteria Order")
        print("6. Process Cafeteria Order")
        print("7. Display Bus Routes")
        print("8. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            sid = input("Enter student ID: ")
            date = input("Enter date: ")
            status = input("Enter status (Present/Absent): ")
            attendance.mark_attendance(sid, date, status)
        elif choice == '2':
            event = input("Enter event name: ")
            sid = input("Enter student ID: ")
            events.register_event(event, sid)
        elif choice == '3':
            sid = input("Enter student ID: ")
            book = input("Enter book title: ")
            library.borrow_book(sid, book)
        elif choice == '4':
            start = input("Enter start location: ")
            end = input("Enter end location: ")
            bus.add_route(start, end)
        elif choice == '5':
            sid = input("Enter student ID: ")
            order = input("Enter order: ")
            cafeteria.place_order(sid, order)
        elif choice == '6':
            cafeteria.process_order()
        elif choice == '7':
            bus.display_routes()
        elif choice == '8':
            break
```

Output:

```
--- Campus Resource Management System ---
1. Mark Attendance
2. Register Event
3. Borrow Book
4. Add Bus Route
5. Place Cafeteria Order
6. Process Cafeteria Order
7. Display Bus Routes
8. Exit
Enter your choice: 1
Enter student ID: 23
Enter date: 24
Enter status (Present/Absent): present
Attendance marked for 23 on 24

--- Campus Resource Management System ---
1. Mark Attendance
2. Register Event
3. Borrow Book
4. Add Bus Route
5. Place Cafeteria Order
6. Process Cafeteria Order
7. Display Bus Routes
8. Exit
Enter your choice: 2
Enter event name: Fresher's party
Enter student ID: 21
21 registered for Fresher's party

--- Campus Resource Management System ---
1. Mark Attendance
2. Register Event
3. Borrow Book
4. Add Bus Route
5. Place Cafeteria Order
6. Process Cafeteria Order
7. Display Bus Routes
8. Exit
Enter your choice: 3
Enter student ID: 62
Enter book title: Python
Book 'Python' borrowed by 62
```

```
--- Campus Resource Management System ---
1. Mark Attendance
2. Register Event
3. Borrow Book
4. Add Bus Route
5. Place Cafeteria Order
6. Process Cafeteria Order
7. Display Bus Routes
8. Exit
Enter your choice: 4
Enter start location: College
Enter end location: kazipet
Route added: College -> kazipet

--- Campus Resource Management System ---
1. Mark Attendance
2. Register Event
3. Borrow Book
4. Add Bus Route
5. Place Cafeteria Order
6. Process Cafeteria Order
7. Display Bus Routes
8. Exit
Enter your choice: 5
Enter student ID: 62
Enter order: Samosa
Order from 62: Samosa

--- Campus Resource Management System ---
1. Mark Attendance
2. Register Event
3. Borrow Book
4. Add Bus Route
5. Place Cafeteria Order
6. Process Cafeteria Order
7. Display Bus Routes
8. Exit
Enter your choice: 6
Processing order for 62: Samosa
```

Explanation:

- **Student Attendance Tracking – Hash Table:** A hash table stores attendance using Student ID as the key for quick access. It provides fast insertion and retrieval with O (1) average time complexity.
- **Event Registration System – List**: A list stores registered students in sequential order. It allows easy addition and traversal of participant records.
- **Book Borrowing – Queue:** A queue follows the FIFO principle for fair book allocation.Students are served in the same order they join the waiting list.
- **Bus Scheduling System – Graph:** A graph represents bus stops as nodes and routes as edges. It efficiently models connections between multiple stops.
- **Cafeteria Order Queue – Queue**: A queue processes food orders in arrival order. It ensures fair and systematic service during busy hours.