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	**:**
Semester	**II**
Academic Session	**: 2025-2026**
Name of Student	**: Akash Reddy**
Enrollment No.	**: 2403A51L30**
Batch No.	**: 51**
Date	**:24-02-2026**

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

Prompt:Create a Python program that implements a Contact Manager using:

1) Array (Python list)
   - Class: ArrayContactManager
   - Methods:
     add_contact(name, phone)
     search_contact(name)
     delete_contact(name)
     display_contacts()
   - Store contacts as list of dictionaries.

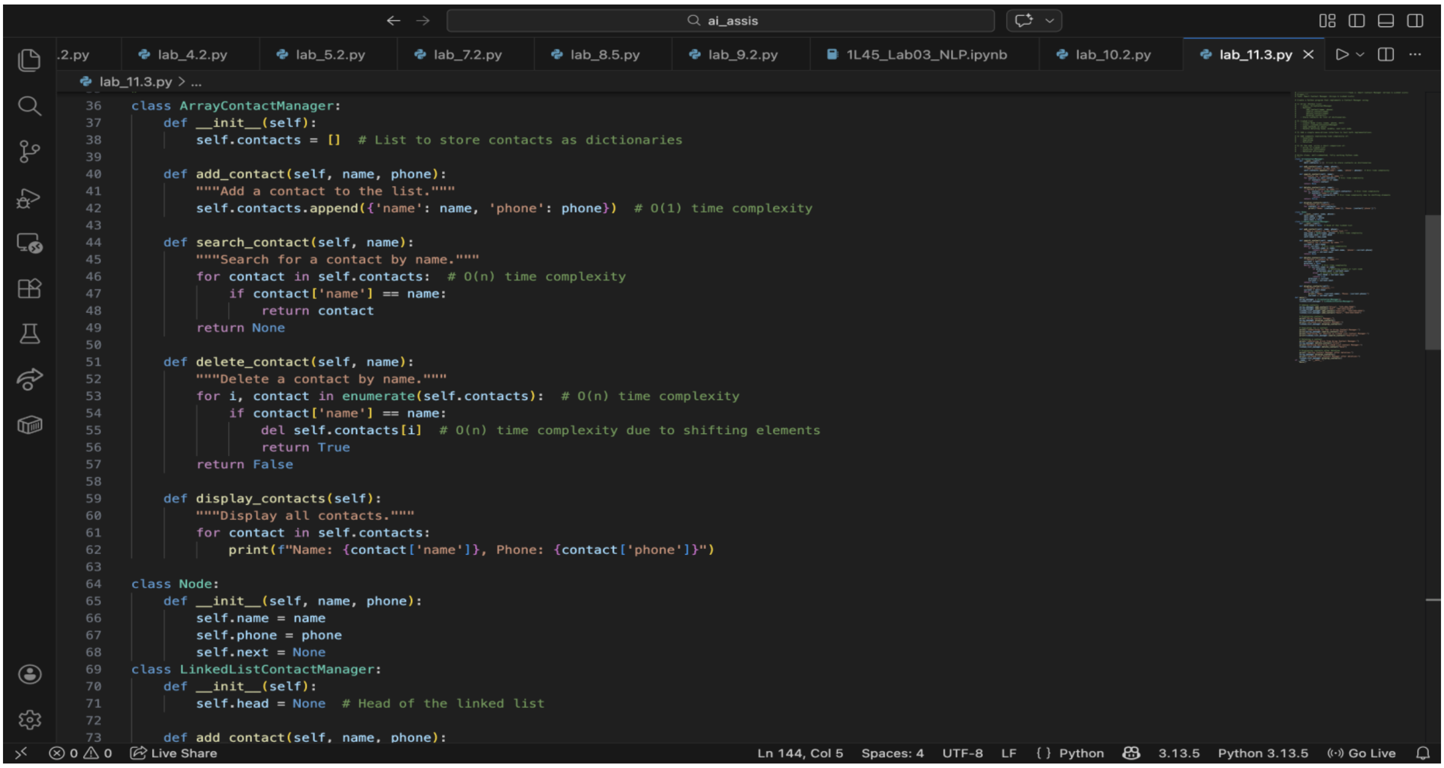
2) Linked List
   - Create Node class (name, phone, next)
   - Class: LinkedListContactManager
   - Same methods as above
   - Handle deleting head, middle, and last node.

3) Add a simple menu-driven interface to test both implementations.

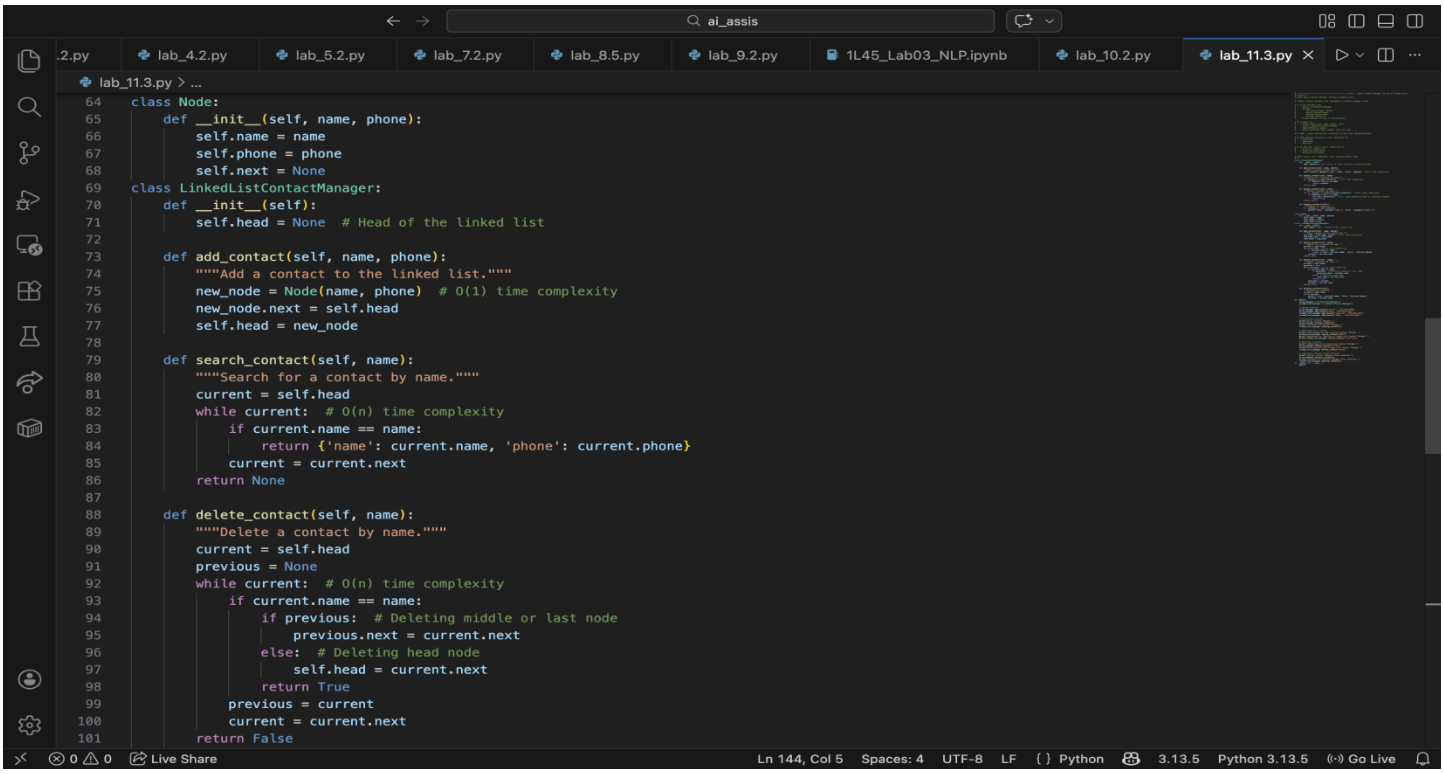
4) Add comments explaining time complexity of:
   - Insertion
   - Searching
   - Deletion

5) At the end, write a short comparison of:
   - Array vs Linked List
   - Insertion efficiency
   - Deletion efficiency

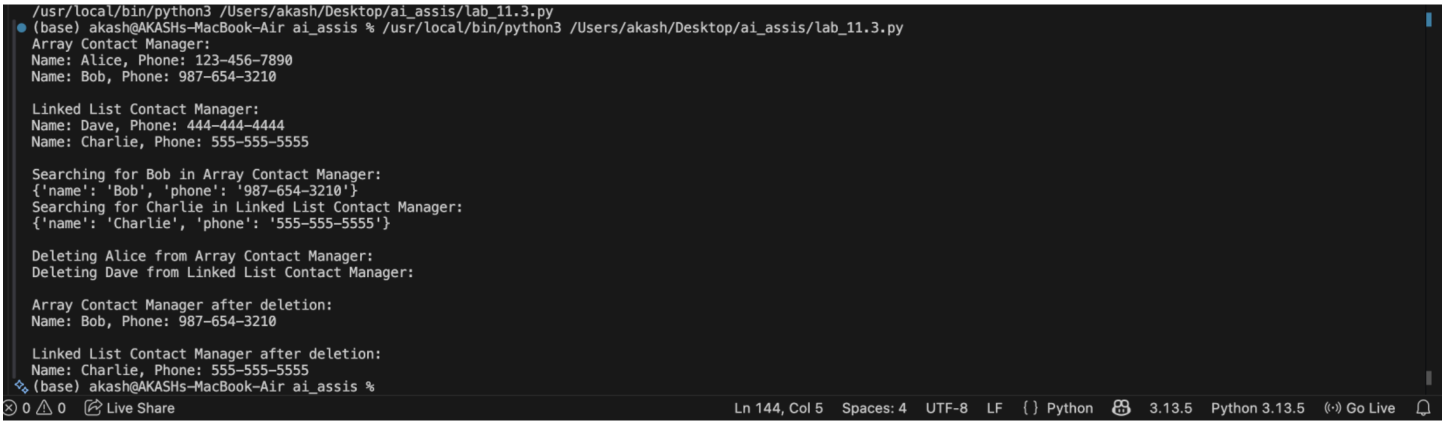
Output:



```python
class ArrayContactManager:
    def __init__(self):
        self.contacts = []  # List to store contacts as dictionaries

    def add_contact(self, name, phone):
        """Add a contact to the list."""
        self.contacts.append({'name': name, 'phone': phone})  # O(1) time complexity

    def search_contact(self, name):
        """Search for a contact by name."""
        for contact in self.contacts:  # O(n) time complexity
            if contact['name'] == name:
                return contact
        return None

    def delete_contact(self, name):
        """Delete a contact by name."""
        for i, contact in enumerate(self.contacts):  # O(n) time complexity
            if contact['name'] == name:
                del self.contacts[i]  # O(n) time complexity due to shifting elements
                return True
        return False

    def display_contacts(self):
        """Display all contacts."""
        for contact in self.contacts:
            print(f"Name: {contact['name']}, Phone: {contact['phone']}")

class Node:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
        self.next = None
class LinkedListContactManager:
    def __init__(self):
        self.head = None  # Head of the linked list

    def add_contact(self, name, phone):
```



```python
class Node:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
        self.next = None
class LinkedListContactManager:
    def __init__(self):
        self.head = None  # Head of the linked list

    def add_contact(self, name, phone):
        """Add a contact to the linked list."""
        new_node = Node(name, phone)  # O(1) time complexity
        new_node.next = self.head
        self.head = new_node

    def search_contact(self, name):
        """Search for a contact by name."""
        current = self.head
        while current:  # O(n) time complexity
            if current.name == name:
                return {'name': current.name, 'phone': current.phone}
            current = current.next
        return None

    def delete_contact(self, name):
        """Delete a contact by name."""
        current = self.head
        previous = None
        while current:  # O(n) time complexity
            if current.name == name:
                if previous:  # Deleting middle or last node
                    previous.next = current.next
                else:  # Deleting head node
                    self.head = current.next
                return True
            previous = current
            current = current.next
        return False
```



```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_11.3.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_11.3.py
Array Contact Manager:
Name: Alice, Phone: 123-456-7890
Name: Bob, Phone: 987-654-3210

Linked List Contact Manager:
Name: Dave, Phone: 444-444-4444
Name: Charlie, Phone: 555-555-5555

Searching for Bob in Array Contact Manager:
{'name': 'Bob', 'phone': '987-654-3210'}
Searching for Charlie in Linked List Contact Manager:
{'name': 'Charlie', 'phone': '555-555-5555'}

Deleting Alice from Array Contact Manager:
Deleting Dave from Linked List Contact Manager:

Array Contact Manager after deletion:
Name: Bob, Phone: 987-654-3210

Linked List Contact Manager after deletion:
Name: Charlie, Phone: 555-555-5555
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation: Array-based contact manager stores contacts in a Python list, making insertion easy but deletion slower due to shifting elements (O(n)).

Linked list-based contact manager uses dynamic memory allocation, allowing efficient deletions without shifting, but searching still takes O(n) time.

Task 2: Library Book Search System (Queues & Priority Queues):
Prompt:
Create a Python program that:

1) Implements a normal Queue (FIFO) to manage book requests.
   - Class: BookRequestQueue
   - Methods:
       enqueue(name, role)   # role = "Student" or "Faculty"
       dequeue()
       display_queue()

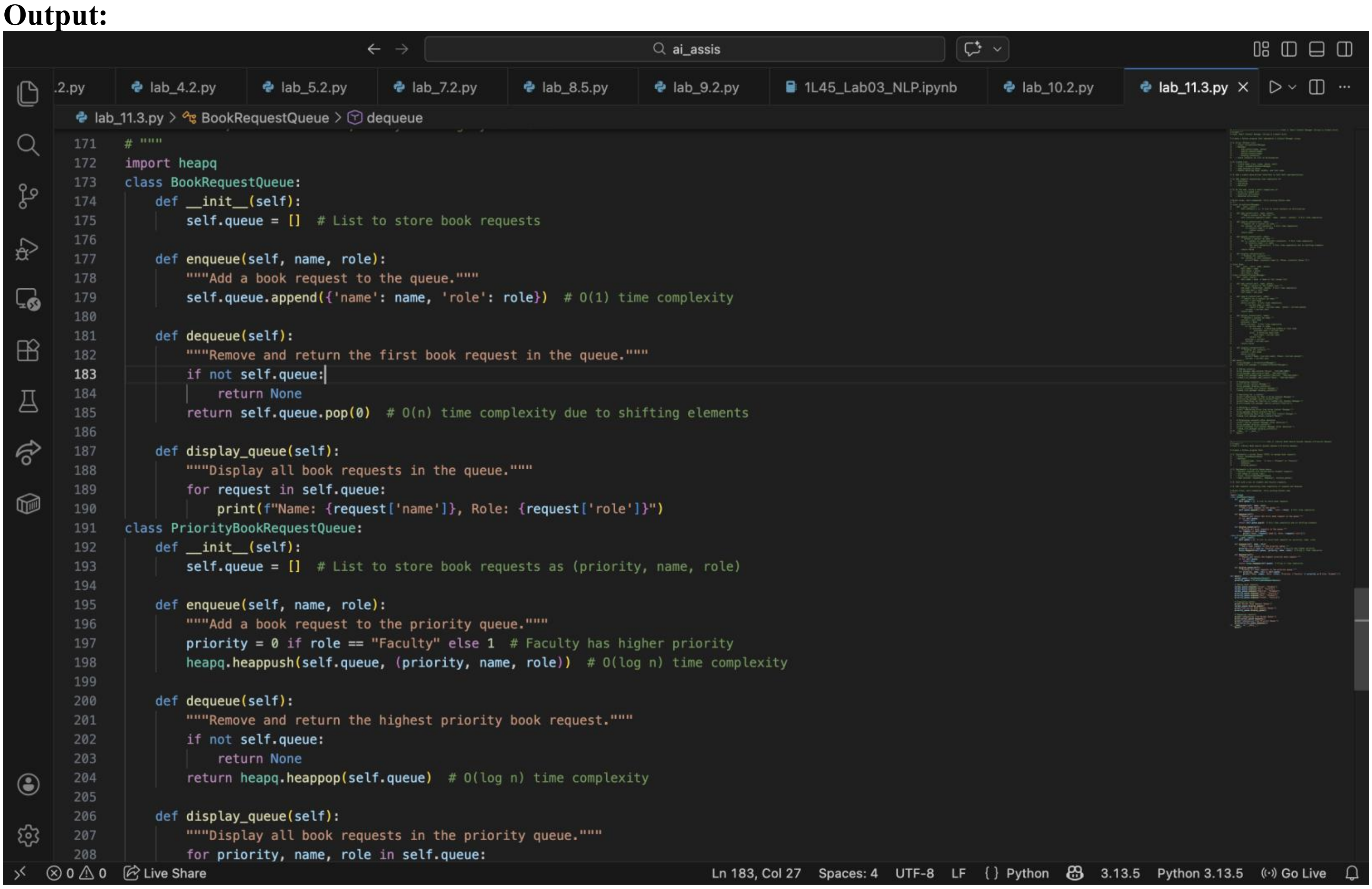
2) Implements a Priority Queue where:
   - Faculty requests are served before Student requests.
   - Use heapq or custom logic.
   - Class: PriorityBookRequestQueue
   - Same methods: enqueue(), dequeue(), display_queue()

3) Test with a mix of student and faculty requests.

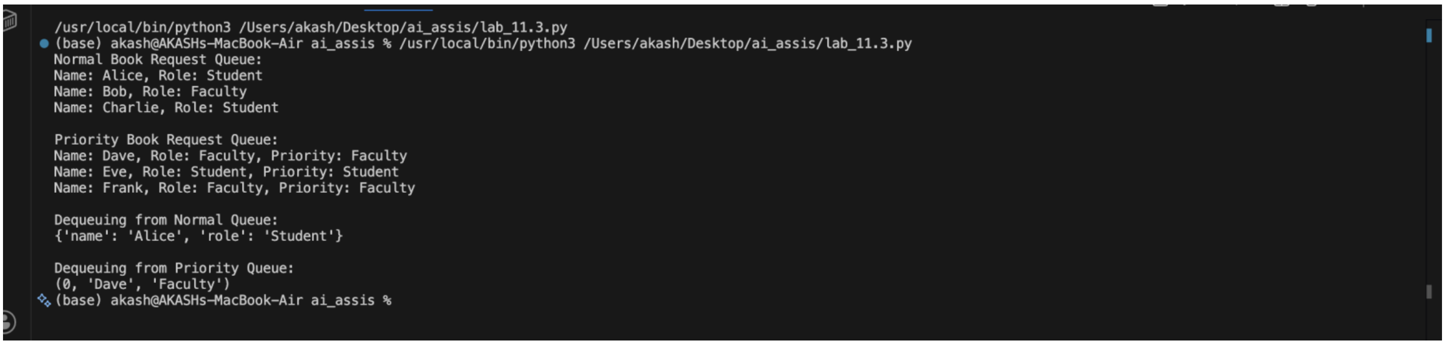
4) Add comments explaining time complexity of enqueue and dequeue.

Write clean, well-commented, fully working Python code.

Output:



```python
# """
import heapq
class BookRequestQueue:
    def __init__(self):
        self.queue = []  # List to store book requests

    def enqueue(self, name, role):
        """Add a book request to the queue."""
        self.queue.append({'name': name, 'role': role})  # O(1) time complexity

    def dequeue(self):
        """Remove and return the first book request in the queue."""
        if not self.queue:
            return None
        return self.queue.pop(0)  # O(n) time complexity due to shifting elements

    def display_queue(self):
        """Display all book requests in the queue."""
        for request in self.queue:
            print(f"Name: {request['name']}, Role: {request['role']}")
class PriorityBookRequestQueue:
    def __init__(self):
        self.queue = []  # List to store book requests as (priority, name, role)

    def enqueue(self, name, role):
        """Add a book request to the priority queue."""
        priority = 0 if role == "Faculty" else 1  # Faculty has higher priority
        heapq.heappush(self.queue, (priority, name, role))  # O(log n) time complexity

    def dequeue(self):
        """Remove and return the highest priority book request."""
        if not self.queue:
            return None
        return heapq.heappop(self.queue)  # O(log n) time complexity

    def display_queue(self):
        """Display all book requests in the priority queue."""
        for priority, name, role in self.queue:
```



```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_11.3.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_11.3.py
Normal Book Request Queue:
Name: Alice, Role: Student
Name: Bob, Role: Faculty
Name: Charlie, Role: Student

Priority Book Request Queue:
Name: Dave, Role: Faculty, Priority: Faculty
Name: Eve, Role: Student, Priority: Student
Name: Frank, Role: Faculty, Priority: Faculty

Dequeuing from Normal Queue:
{'name': 'Alice', 'role': 'Student'}

Dequeuing from Priority Queue:
(0, 'Dave', 'Faculty')
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation:A normal Queue (FIFO) processes book requests in the order they arrive, without considering user type.

A Priority Queue ensures faculty requests are served before student requests, giving higher priority to faculty members.

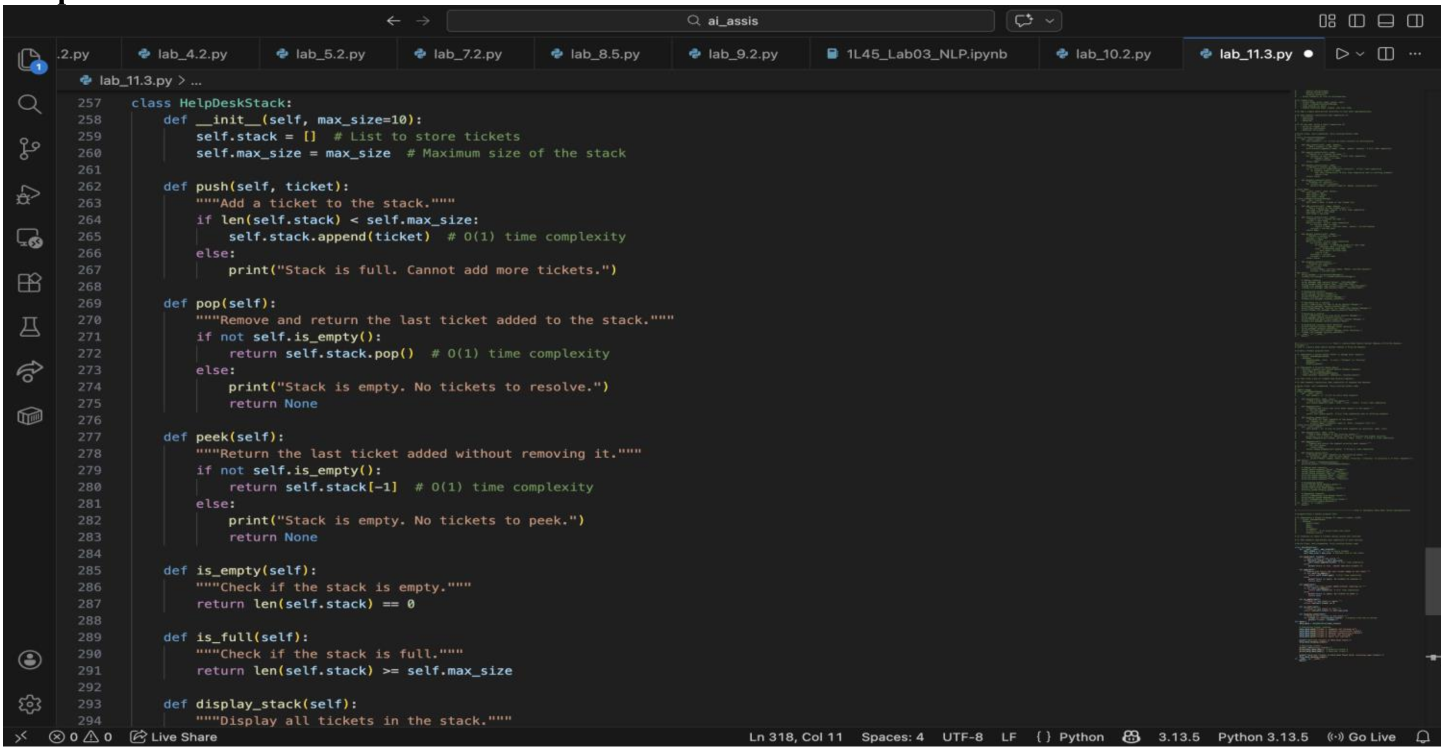
Task 3: Emergency Help Desk (Stack Implementation):
Prompt: Create a Python program that:

1) Implements a Stack to manage IT support tickets (LIFO).
- Class: HelpDeskStack
- Methods:
push(ticket)
pop()
peek()
is_empty()
is_full() # if using fixed size stack
display_stack()

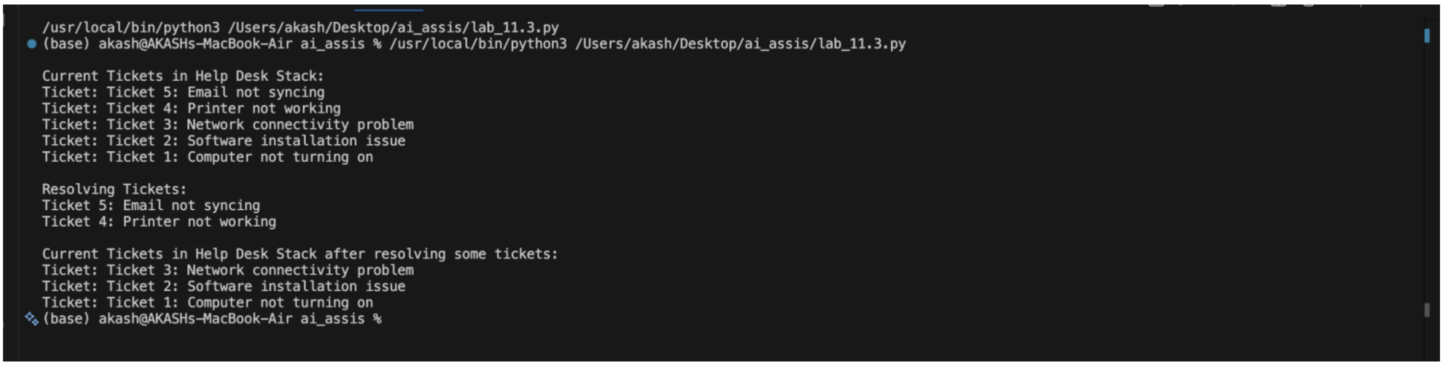
2) Simulate at least 5 tickets being raised and resolved.

3) Add comments explaining time complexity of push and pop.

Write clean, well-commented, fully working Python code.
Output:



```python
class HelpDeskStack:
    def __init__(self, max_size=10):
        self.stack = []  # List to store tickets
        self.max_size = max_size  # Maximum size of the stack

    def push(self, ticket):
        """Add a ticket to the stack."""
        if len(self.stack) < self.max_size:
            self.stack.append(ticket)  # O(1) time complexity
        else:
            print("Stack is full. Cannot add more tickets.")

    def pop(self):
        """Remove and return the last ticket added to the stack."""
        if not self.is_empty():
            return self.stack.pop()  # O(1) time complexity
        else:
            print("Stack is empty. No tickets to resolve.")
            return None

    def peek(self):
        """Return the last ticket added without removing it."""
        if not self.is_empty():
            return self.stack[-1]  # O(1) time complexity
        else:
            print("Stack is empty. No tickets to peek.")
            return None

    def is_empty(self):
        """Check if the stack is empty."""
        return len(self.stack) == 0

    def is_full(self):
        """Check if the stack is full."""
        return len(self.stack) >= self.max_size

    def display_stack(self):
        """Display all tickets in the stack."""
```



```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_11.3.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_11.3.py

Current Tickets in Help Desk Stack:
Ticket: Ticket 5: Email not syncing
Ticket: Ticket 4: Printer not working
Ticket: Ticket 3: Network connectivity problem
Ticket: Ticket 2: Software installation issue
Ticket: Ticket 1: Computer not turning on

Resolving Tickets:
Ticket 5: Email not syncing
Ticket 4: Printer not working

Current Tickets in Help Desk Stack after resolving some tickets:
Ticket: Ticket 3: Network connectivity problem
Ticket: Ticket 2: Software installation issue
Ticket: Ticket 1: Computer not turning on
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation:A Stack follows the Last-In, First-Out (LIFO) principle, meaning the most recently raised ticket is resolved first.

Push adds a new ticket to the stack, while pop removes the latest ticket, clearly demonstrating LIFO behavior

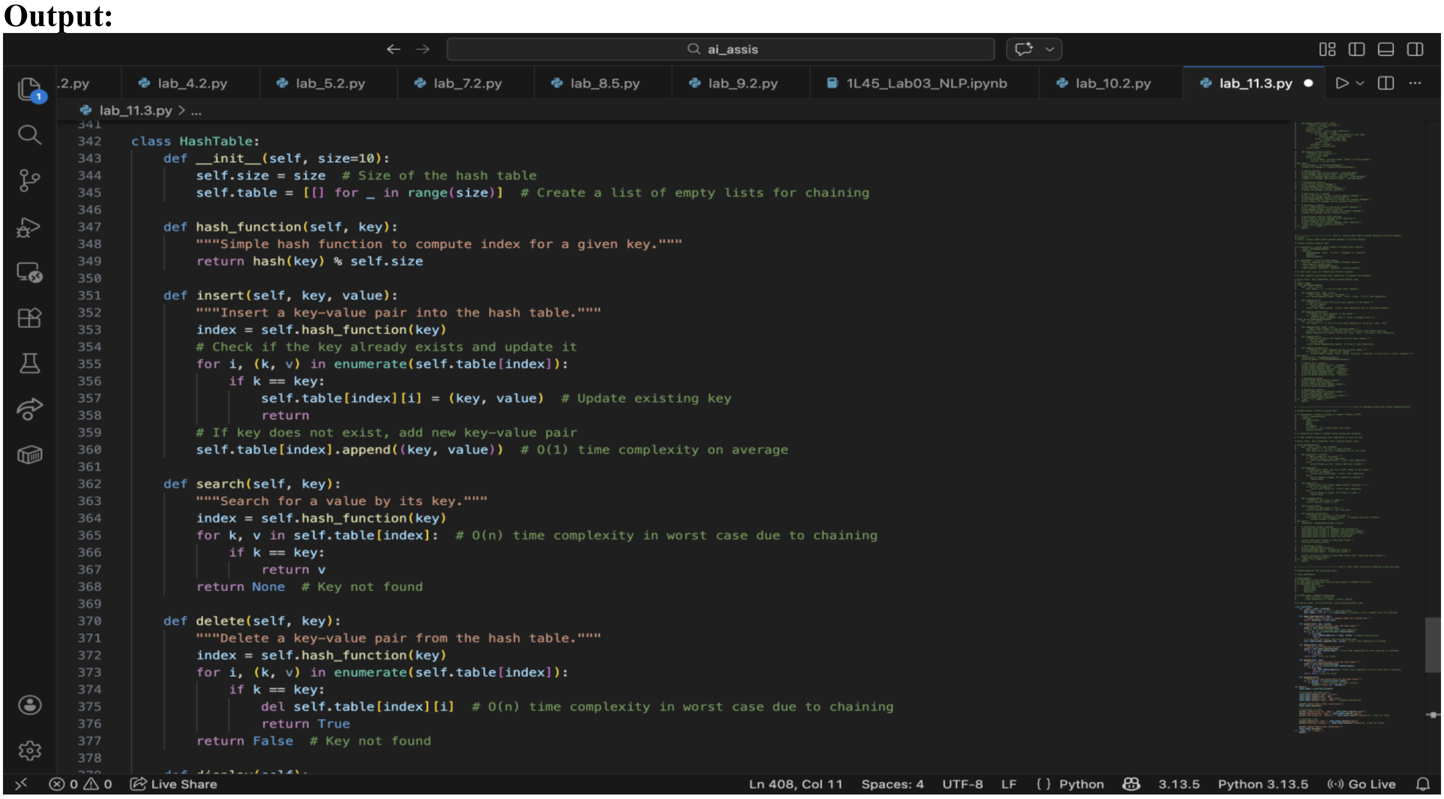
Task 4: Hash Table:
Prompt: Complete the following class:

class HashTable:

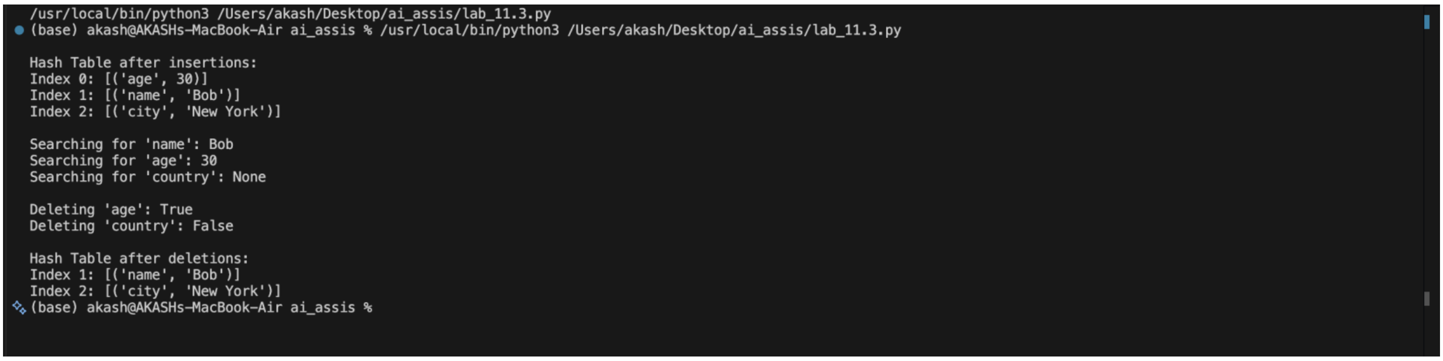
Requirements:
1) Implement a hash function.
2) Use separate chaining (list at each index) to handle collisions.
3) Implement methods:
insert(key, value)
search(key)
delete(key)
display()

4) Add proper comments explaining:
- How collision is handled
- Time complexity of insert, search, delete

5) Write clean, well-structured, fully working Python code.
Output:



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

    def hash_function(self, key):
        """Simple 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_function(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 key does not exist, add new key-value pair
        self.table[index].append((key, value))  # O(1) time complexity on average

    def search(self, key):
        """Search for a value by its key."""
        index = self.hash_function(key)
        for k, v in self.table[index]:  # O(n) time complexity in worst case due to chaining
            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_function(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]  # O(n) time complexity in worst case due to chaining
                return True
        return False  # Key not found
```



```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_11.3.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_11.3.py

Hash Table after insertions:
Index 0: [('age', 30)]
Index 1: [('name', 'Bob')]
Index 2: [('city', 'New York')]

Searching for 'name': Bob
Searching for 'age': 30
Searching for 'country': None

Deleting 'age': True
Deleting 'country': False

Hash Table after deletions:
Index 1: [('name', 'Bob')]
Index 2: [('city', 'New York')]
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation:A Hash Table stores key-value pairs using a hash function to determine the index for storage.

When collisions occur, separate chaining stores multiple elements at the same index using a list.

Task 5: Real-Time Application Challenge:

Prompt:1) Create a mapping table:
Feature → Suitable Data Structure → Justification (2-3 sentences)

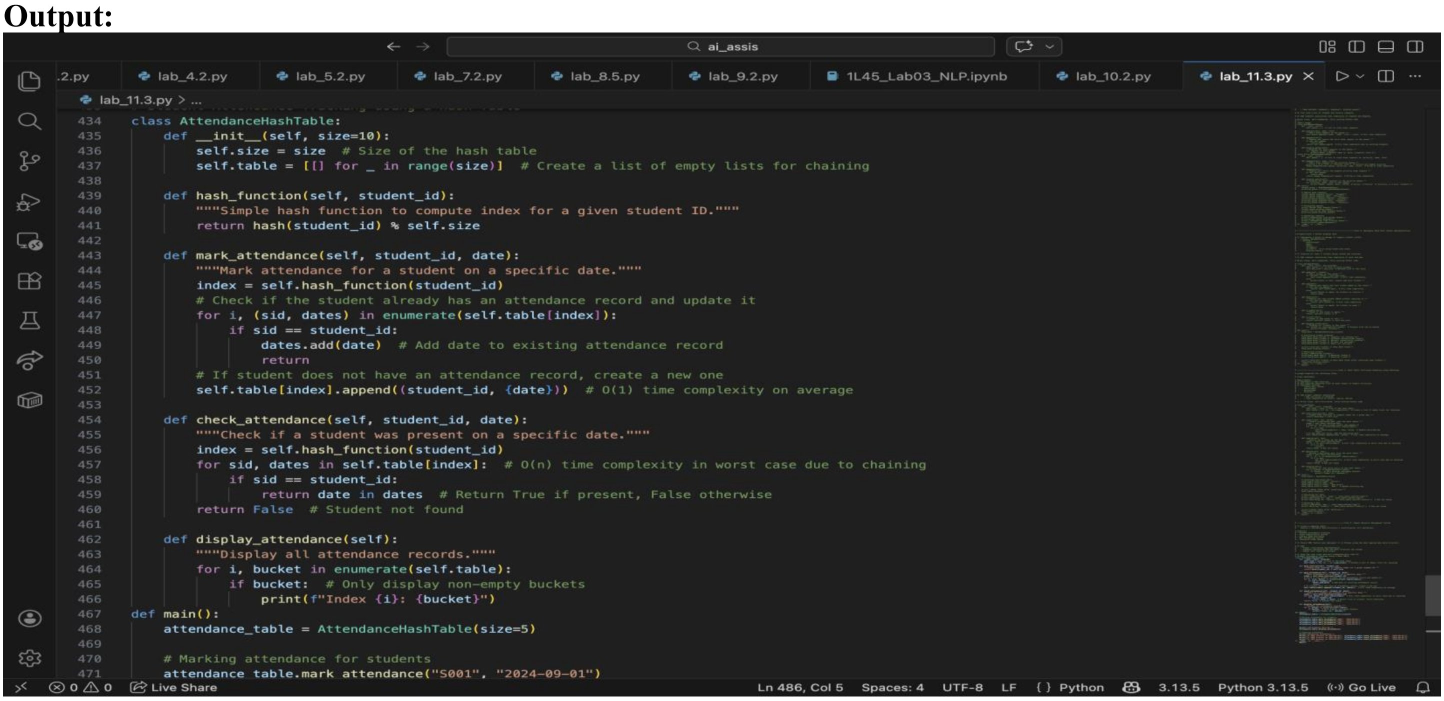
Features:
- Student Attendance Tracking
- Event Registration System
- Library Book Borrowing
- Bus Scheduling System
- Cafeteria Order Queue

2) Choose ONE feature and implement it in Python using the most appropriate data structure.

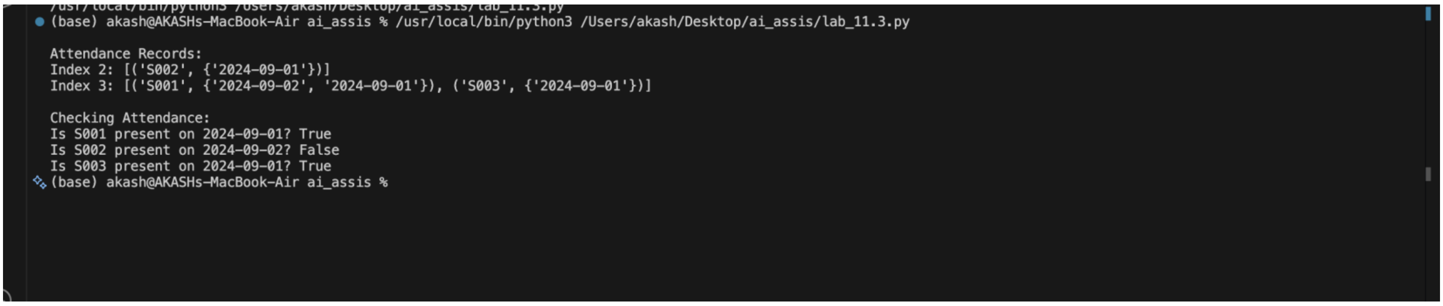
3) Add:
- Proper class-based implementation
- Comments explaining why the data structure was chosen
- Sample test cases with output

4) Keep the code clean and well-commented.

Output:



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

    def hash_function(self, student_id):
        """Simple hash function to compute index for a given student ID."""
        return hash(student_id) % self.size

    def mark_attendance(self, student_id, date):
        """Mark attendance for a student on a specific date."""
        index = self.hash_function(student_id)
        # Check if the student already has an attendance record and update it
        for i, (sid, dates) in enumerate(self.table[index]):
            if sid == student_id:
                dates.add(date)  # Add date to existing attendance record
                return
        # If student does not have an attendance record, create a new one
        self.table[index].append((student_id, {date}))  # O(1) time complexity on average

    def check_attendance(self, student_id, date):
        """Check if a student was present on a specific date."""
        index = self.hash_function(student_id)
        for sid, dates in self.table[index]:  # O(n) time complexity in worst case due to chaining
            if sid == student_id:
                return date in dates  # Return True if present, False otherwise
        return False  # Student not found

    def display_attendance(self):
        """Display all attendance records."""
        for i, bucket in enumerate(self.table):
            if bucket:  # Only display non-empty buckets
                print(f"Index {i}: {bucket}")
def main():
    attendance_table = AttendanceHashTable(size=5)

    # Marking attendance for students
    attendance_table.mark_attendance("S001", "2024-09-01")
```



```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_11.3.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_11.3.py

Attendance Records:
Index 2: [('S002', {'2024-09-01'})]
Index 3: [('S001', {'2024-09-02', '2024-09-01'}), ('S003', {'2024-09-01'})]

Checking Attendance:
Is S001 present on 2024-09-01? True
Is S002 present on 2024-09-02? False
Is S003 present on 2024-09-01? True
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation: Different campus features require different data structures based on how data is stored and accessed (e.g., Queue for orders, Hash Table for attendance lookup).

Choosing the correct data structure improves efficiency, performance, and real-time system behavior.