AI ASSISTANT CODING
Assignment-11.3

Name : Hasini Irumalla
Hall Ticket No.: 230351286
Batch No.: 05

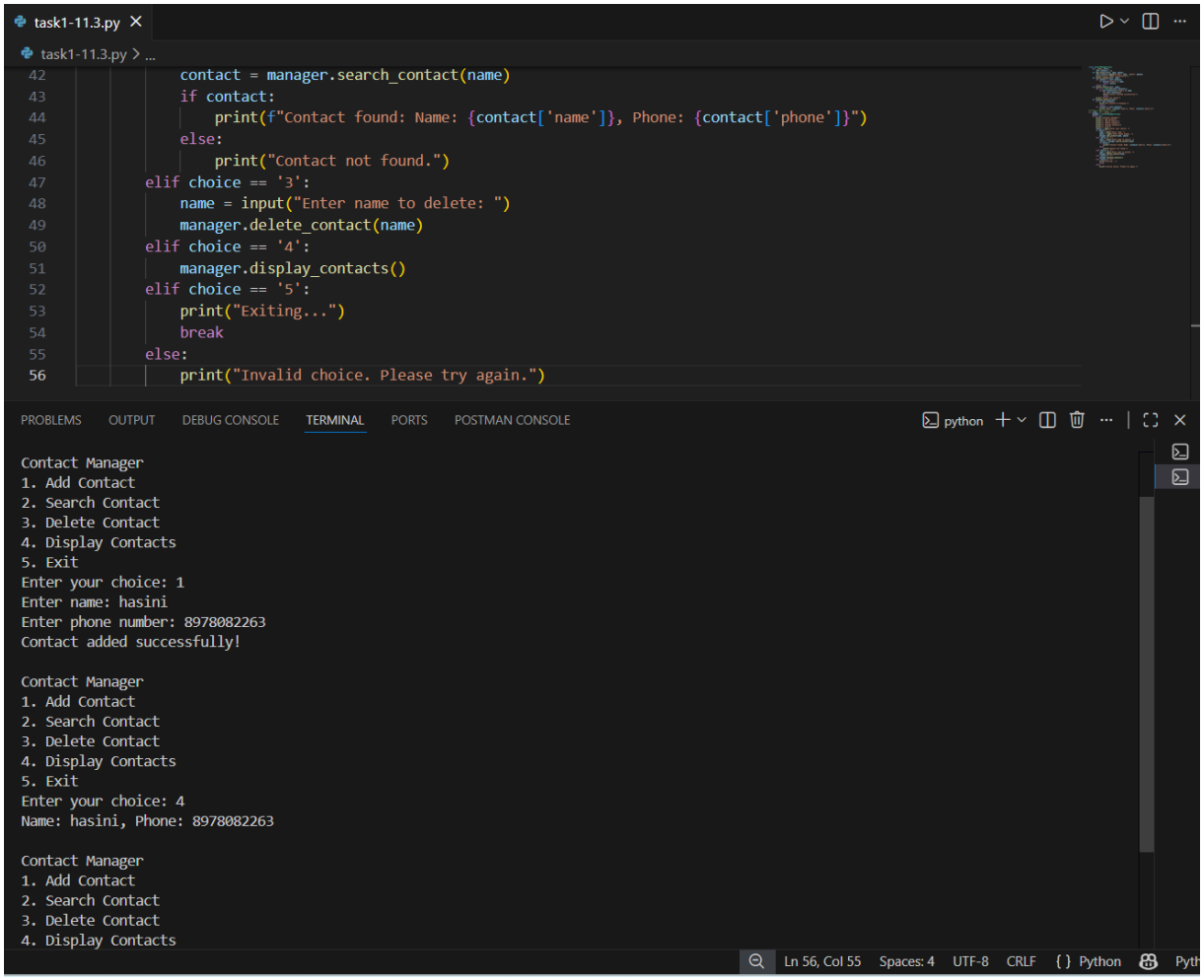
Task 1: Smart Contact Manager (Arrays & Linked Lists) Scenario
SR University's student club requires a simple Contact Manager Application to store members' names and phone numbers. The system should support efficient addition, searching, and deletion of contacts.
Tasks
1. Implement the contact manager 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:
Generate a Python program to implement a Contact Manager using a list (array).
Each contact should store name and phone number.
Include the following methods:
- add_contact(name, phone)
- search_contact(name)
- delete_contact(name)
The program should handle basic errors and print appropriate messages.
Add comments explaining each method**. 1A. Array-Based**
Implementation

```
class ContactManagerArray:
  def __init__(self):
 self.contacts = []
def add_contact(self, name, phone):
self.contacts.append({"name": name, "phone": phone})
print("Contact added successfully!")     def
search_contact(self, name):
 for contact in self.contacts:
```

```
        if contact["name"] == name:
            return contact        return None
def delete_contact(self, name):        for i in
range(len(self.contacts)):            if
self.contacts[i]["name"] == name:
            del self.contacts[i]
print("Contact deleted successfully!")
return
      print("Contact not found!")
def display_contacts(self):
if not self.contacts:
        print("No contacts to display.")            return        for
contact in self.contacts:            print(f"Name: {contact['name']},
Phone: {contact['phone']}")
# Example usage dynamic inputs if
__name__ == "__main__":
  manager = ContactManagerArray()
while True:
      print("\nContact Manager")
print("1. Add Contact")        print("2.
Search Contact")        print("3. Delete
Contact")        print("4. Display
Contacts")        print("5. Exit")
choice = input("Enter your choice: ")
if choice == '1':
        name = input("Enter name: ")
phone = input("Enter phone number: ")
manager.add_contact(name, phone)        elif
choice == '2':
        name = input("Enter name to search: ")
contact = manager.search_contact(name)            if
contact:
            print(f"Contact found: Name: {contact['name']}, Phone: {contact['phone']}")
else:
            print("Contact not found.")
elif choice == '3':
        name = input("Enter name to delete: ")
manager.delete_contact(name)        elif
choice == '4':
        manager.display_contacts()
      elif choice == '5':
print("Exiting...")            break        else:
print("Invalid choice. Please try again.")
```

Output:



Prompt: Generate a Python implementation of a Contact Manager using a singly linked list. Create a Node class to store name and phone number.
Include:
- add_contact(name, phone)
- search_contact(name)
- delete_contact(name)
Use dynamic memory allocation and include proper comments explaining the logic.

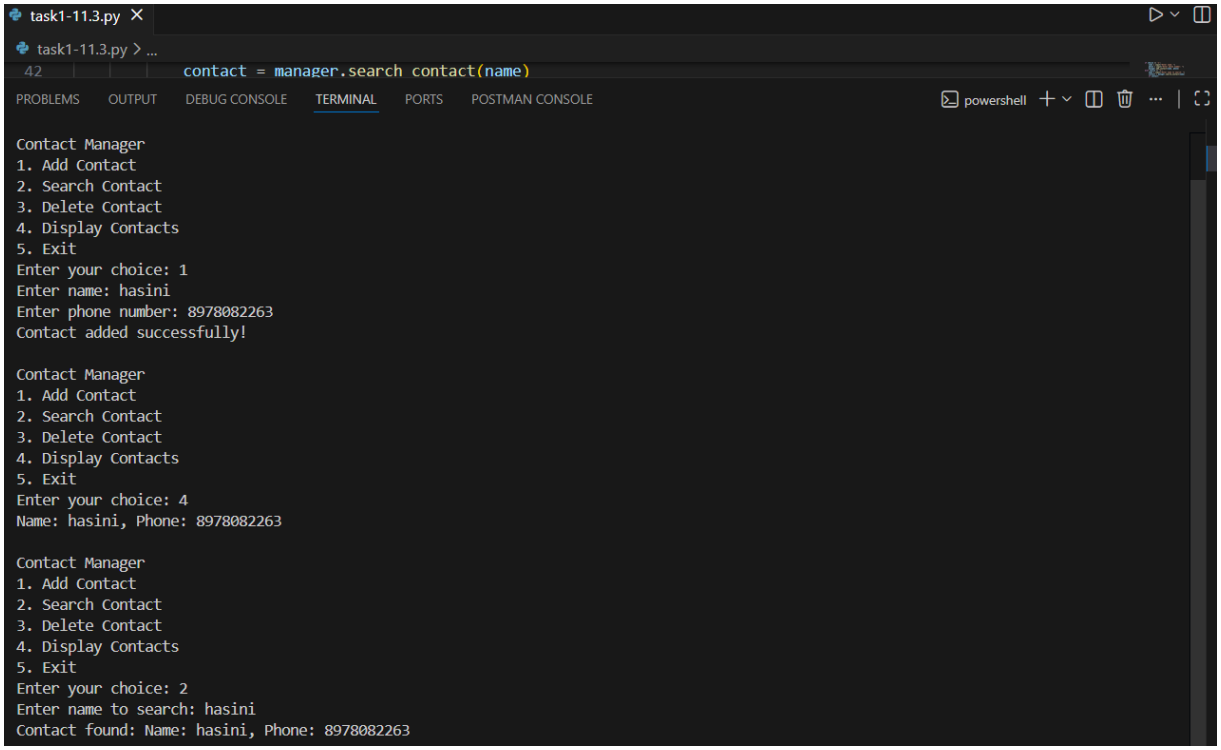
1B. Linked List Implementation

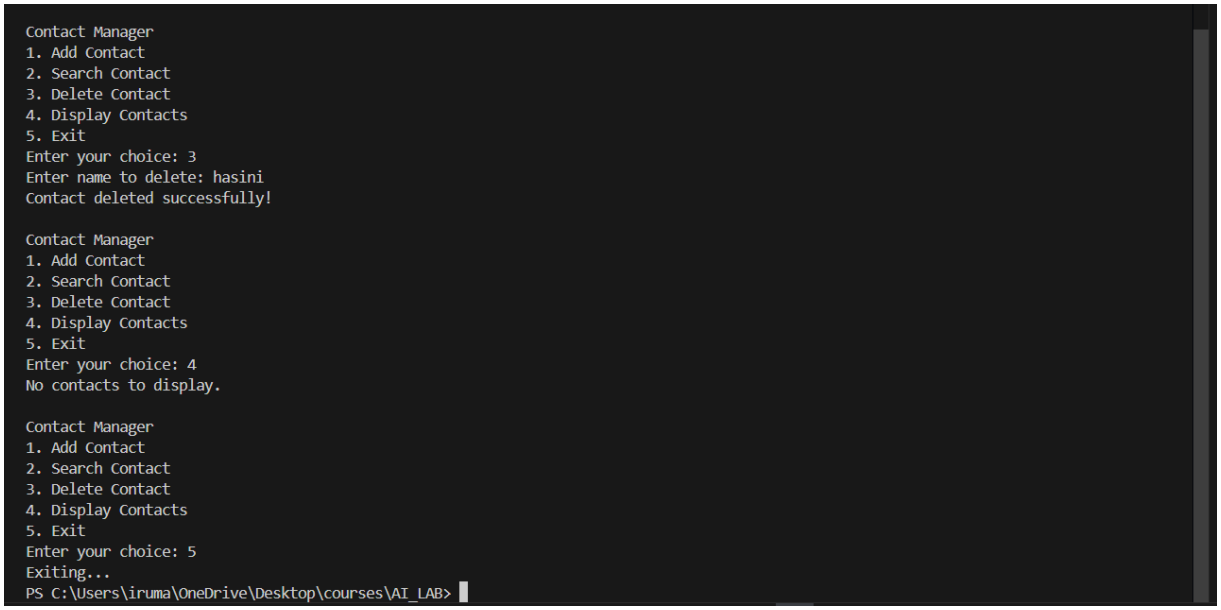
```
class Node:     def __init__(self,
name, phone):
        self.name = name
self.phone = phone          self.next
= None

class ContactManagerLinkedList:
def __init__(self):          self.head
= None
```

```
    def 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 == name:                 return {"name": current.name,
"phone": current.phone}             current = current.next
return None

    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
print("Contact deleted successfully!")
return             prev = current             current =
current.next

        print("Contact not found!") #
Example usage dynamic inputs if
__name__ == "__main__":
    contact_manager = ContactManagerLinkedList()

    while True:
        print("\nContact Manager")
print("1. Add Contact")         print("2.
Search Contact")         print("3.
Delete Contact")         print("4. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':
            name = input("Enter name: ")             phone
= input("Enter phone number: ")
contact_manager.add_contact(name, phone)
elif choice == '2':
```

```
        name = input("Enter name to search: ")
contact = contact_manager.search_contact(name)
if contact:
            print(f"Contact found: Name: {contact['name']}, Phone: {contact['phone']}")
else:
            print("Contact not found!")
elif choice == '3':
        name = input("Enter name to delete: ")
contact_manager.delete_contact(name)
elif choice == '4':             print("Exiting...")
break       else:
          print("Invalid choice! Please try again.")
```

OUTPUT :





Comparison: Array vs Linked List

Operation	Array	Linked List
Insertion	O(1) at end	O(1) at beginning
Search	O(n)	O(n)
Deletion	O(n) (shifting required)	O(n) (no shifting)
Memory	Fixed/Resizing	Dynamic

Explanation:

- Arrays are simple and fast for indexed access.
- Linked Lists are better for frequent insertions and deletions.
- For a dynamic contact manager, Linked List is more flexible.

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: Create a Python program that implements a FIFO queue for managing library book requests.

Use collections.deque.

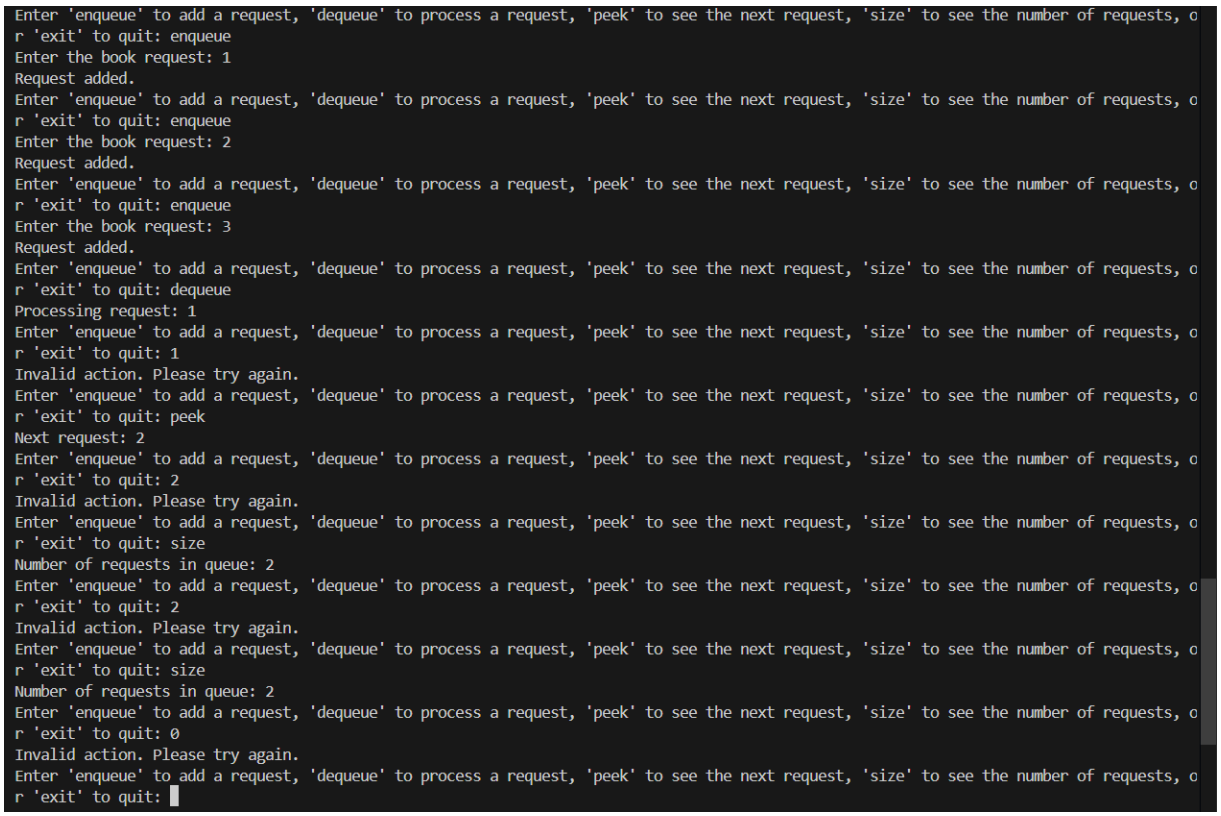
Include:
- enqueue(request)
- dequeue()
Simulate requests from students.
Add comments explaining the working.

2A. Simple Queue (FIFO)

```
from collections import deque

class BookQueue:
def __init__(self):
        self.queue = deque()

    def enqueue(self, request):
self.queue.append(request)          print("Request
added.")

    def dequeue(self):
if self.queue:
            return self.queue.popleft()
return "Queue is empty"      def
peek(self):          if self.queue:
            return self.queue[0]
return "Queue is empty"      def
is_empty(self):
        return len(self.queue) == 0
def size(self):
        return len(self.queue)
# Example usage dyanamic inputs
book_queue = BookQueue() while
True:
    action = input("Enter 'enqueue' to add a request, 'dequeue' to process a request, 'peek' to see
the next request, 'size' to see the number of requests, or 'exit' to quit: ")

    if action == 'enqueue':
        request = input("Enter the book request: ")
book_queue.enqueue(request)      elif action ==
'dequeue':
        print("Processing request:", book_queue.dequeue())
elif action == 'peek':
        print("Next request:", book_queue.peek())
elif action == 'size':
        print("Number of requests in queue:", book_queue.size())
elif action == 'exit':
```

break else: print("Invalid action. Please try again.")

Output:



Prompt: Create a Python program to implement a priority queue for a library system. Faculty requests must be served before student requests.
Use heapq module.
Include:
- enqueue(name, role)
- dequeue()
Test the system with mixed student and faculty requests.
Explain how priority is assigned. **2B.**

Priority Queue (Faculty Priority)

```
import heapq

class BookPriorityQueue:
def __init__(self):
self.queue = []

    def enqueue(self, name, role):
        priority = 0 if role == "Faculty" else 1
heapq.heappush(self.queue, (priority, name))        print("Request
added.")
```

```
   def dequeue(self):
if self.queue:
            return heapq.heappop(self.queue)
return "Queue is empty"     def
display(self):          if self.queue:
           print("Current Queue:")
for priority, name in self.queue:
                role = "Faculty" if priority == 0 else "Student"
print(f"{name} ({role})")          else:
           print("Queue  is  empty")
# Example usage dynamic inputs
if  __name__  ==  "__main__":
pq      =      BookPriorityQueue()
while True:
       action = input("Enter 'enqueue' to add a request, 'dequeue' to process a request, 'display' to
show the queue, or 'exit' to quit: ")          if action == "enqueue":
            name = input("Enter the name of the requester: ")
role = input("Enter the role (Faculty/Student): ")
pq.enqueue(name, role)          elif action == "dequeue":
result = pq.dequeue()               if isinstance(result, tuple):
                print(f"Processing request from {result[1]} ({'Faculty' if result[0] == 0 else
'Student'})")
else:
                print(result)
elif action == "display":
           pq.display()
elif action == "exit":
           break          else:               print("Invalid
action. Please try again.")
```

Output:

```
Enter 'enqueue' to add a request, 'dequeue' to process a request, 'display' to show the queue, or 'exit' to quit: enqueue
Enter the name of the requester: 1
Enter the role (Faculty/Student): faculty
Request added.
Enter 'enqueue' to add a request, 'dequeue' to process a request, 'display' to show the queue, or 'exit' to quit: enqueue
Enter the name of the requester: 2
Enter the role (Faculty/Student): faculty
Request added.
Enter 'enqueue' to add a request, 'dequeue' to process a request, 'display' to show the queue, or 'exit' to quit: enqueue
Enter the name of the requester: 3
Enter the role (Faculty/Student): student
Request added.
Enter 'enqueue' to add a request, 'dequeue' to process a request, 'display' to show the queue, or 'exit' to quit: enqueue
Enter the name of the requester: 4
Enter the role (Faculty/Student): student
Request added.
Enter 'enqueue' to add a request, 'dequeue' to process a request, 'display' to show the queue, or 'exit' to quit: display
Current Queue:
1 (Student)
2 (Student)
3 (Student)
4 (Student)
Enter 'enqueue' to add a request, 'dequeue' to process a request, 'display' to show the queue, or 'exit' to quit: dequeue
Processing request from 1 (Student)
Enter 'enqueue' to add a request, 'dequeue' to process a request, 'display' to show the queue, or 'exit' to quit: display
Current Queue:
2 (Student)
4 (Student)
3 (Student)
Enter 'enqueue' to add a request, 'dequeue' to process a request, 'display' to show the queue, or 'exit' to quit: exit
```

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: Generate a Python program to implement a stack for managing IT help desk tickets.
Include methods:
- push(ticket)
- pop()
- peek()
- is_empty()
- is_full()

Simulate at least five support tickets and demonstrate LIFO behavior.
Add comments explaining stack operations.

Code:

```
class HelpDeskStack:
def __init__(self):
```

```
self.stack = []
self.max_size = 5

   def push(self, ticket):         if
len(self.stack) == self.max_size:
           print("Stack is full!")
else:
           self.stack.append(ticket)
print("Ticket added.")

   def pop(self):         if
self.is_empty():              return "No
tickets to resolve."         return
self.stack.pop()

   def peek(self):         if
self.is_empty():              return "No
tickets available."         return
self.stack[-1]

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

   def is_full(self):
      return len(self.stack) == self.max_size
# Example usage dynamic inputs help_desk
= HelpDeskStack() while True:
   action = input("Enter action (push, pop, peek, exit): ")
if action == "push":         ticket = input("Enter ticket
description: ")         help_desk.push(ticket)     elif action
== "pop":         print(help_desk.pop())     elif action ==
"peek":         print(help_desk.peek())     elif action ==
"exit":
       break
else:
print("Invalid
action. Please
try again.")
```

Output:

```
Enter action (push, pop, peek, exit): push
Enter ticket description: 1
Ticket added.
Enter action (push, pop, peek, exit): push
Enter ticket description: 2
Ticket added.
Enter action (push, pop, peek, exit): push
Enter ticket description: 3
Ticket added.
Enter action (push, pop, peek, exit): peek
3
Enter action (push, pop, peek, exit): pop
3
Enter action (push, pop, peek, exit): peek
2
Enter action (push, pop, peek, exit): pop
2
Enter action (push, pop, peek, exit): peek
1
Enter action (push, pop, peek, exit): push 2
Invalid action. Please try again.
Enter action (push, pop, peek, exit): push
Enter ticket description: 2
Ticket added.
Enter action (push, pop, peek, exit): peek
2
Enter action (push, pop, peek, exit): pop
2
Enter action (push, pop, peek, exit): exit
```

LIFO Demonstration

The last ticket added ("Network Down") will be resolved first.

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: Generate a Python class for implementing a Hash Table with collision handling using chaining.

Include:

- insert(key, value)
- search(key)
- delete(key)

Implement a simple hash function.

Add detailed comments explaining how collision handling works.
Provide example usage and test cases.

Code:

```
class HashTable:     def
__init__(self, size=10):
       self.size = size         self.table = [[]
for _ in range(size)]

   def hash_function(self, key):
return hash(key) % self.size

   def insert(self, key, value):
       index = self.hash_function(key)
self.table[index].append((key, value))         print("Inserted
successfully.")

   def search(self, key):
       index = self.hash_function(key)
for k, v in self.table[index]:
           if k == key:
return v         return "Key
not found"

   def delete(self, key):
       index = self.hash_function(key)         for i,
(k, v) in enumerate(self.table[index]):             if
k == key:
               del self.table[index][i]
return "Deleted successfully"
return "Key not found"
# Example usage dynamic inputs
hash_table = HashTable() while
True:
   print("\n1. Insert")
print("2. Search")     print("3.
Delete")     print("4. Exit")
choice = input("Enter your
choice: ")

   if choice == '1':
       key = input("Enter key: ")
value = input("Enter value: ")
hash_table.insert(key, value)     elif
choice == '2':
```

```
        key = input("Enter key to search: ")
result = hash_table.search(key)
print(f"Search result: {result}")     elif
choice == '3':
        key = input("Enter key to delete: ")
result = hash_table.delete(key)
print(result)     elif choice == '4':
        break     else:         print("Invalid choice.
Please try again.")
```

Output:

```
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 1
Enter key: 1
Enter value: 11
Inserted successfully.

1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 1
Enter key: 2
Enter value: 22
Inserted successfully.

1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 1
Enter key: 3
Enter value: 33
Inserted successfully.

1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 2
Enter key to search: 4
Search result: Key not found
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 2
Enter key to search: 2
Search result: 22

1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 3
Enter key to delete: 1
Deleted successfully

1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 2
Enter key to search: 1
Search result: Key not found

1. Insert
2. Search
3. Delete
4. Exit
Enter your choice: 4
```

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: Suggest the most appropriate data structure for the following campus system features:
- Student Attendance Tracking
- Event Registration System
- Library Book Borrowing
- Bus Scheduling System
- Cafeteria Order Queue

Provide justification in 2-3 sentences for each selection.

Present the answer in table format.

Feature	Mapping Feature	Data Structure	Justification
Attendance Tracking	Hash Table	Fast lookup by student ID	
Event Registration	Linked List	Dynamic registrations	
Library Borrowing	Priority Queue	Faculty priority	
Bus Scheduling	Queue	FIFO boarding	
Cafeteria Orders	Queue	Orders processed in arrival order	

Prompt: Generate a Python implementation for a Student Attendance Tracking system. Use a suitable data structure for fast lookup.
Include:
- mark_attendance(student_id, status)
- check_attendance(student_id) Add comments and example test cases.

Code:

```
class AttendanceSystem:
def __init__(self):
self.records = {}  def
mark_attendance(self,
student_id, status):
    self.records[student_id] = status
print("Attendance marked.")
```

```
    def check_attendance(self, student_id):
        return self.records.get(student_id, "No record found")
# Example usage dynamic inputs attendance_system = AttendanceSystem() while True:
action = input("Enter 'mark' to mark attendance, 'check' to check attendance, or 'exit' to quit:
")     if action ==
'mark':
        student_id = input("Enter student ID: ")         status =
input("Enter attendance status (present/absent): ")
attendance_system.mark_attendance(student_id, status)     elif
action == 'check':
        student_id = input("Enter student ID: ")
print(attendance_system.check_attendance(student_id))     elif
action == 'exit':
        break     else:         print("Invalid action.
Please try again.")
```

Output:

```
Enter 'mark' to mark attendance, 'check' to check attendance, or 'exit' to quit: mark
Enter student ID: 1
Enter attendance status (present/absent): present
Attendance marked.
Enter 'mark' to mark attendance, 'check' to check attendance, or 'exit' to quit: mark
Enter student ID: 2
Enter attendance status (present/absent): present
Attendance marked.
Enter 'mark' to mark attendance, 'check' to check attendance, or 'exit' to quit: mark
Enter student ID: 3
Enter attendance status (present/absent): absent
Attendance marked.
Enter 'mark' to mark attendance, 'check' to check attendance, or 'exit' to quit: mark
Enter student ID: 4
Enter attendance status (present/absent): present
Attendance marked.
Enter 'mark' to mark attendance, 'check' to check attendance, or 'exit' to quit: mark
Enter student ID: 5
Enter attendance status (present/absent): absent
Attendance marked.
Enter 'mark' to mark attendance, 'check' to check attendance, or 'exit' to quit: mark
Enter student ID: 6
Enter attendance status (present/absent): present
Attendance marked.
Enter 'mark' to mark attendance, 'check' to check attendance, or 'exit' to quit: mark
Enter student ID: 7
Enter attendance status (present/absent): present
Attendance marked.
Enter 'mark' to mark attendance, 'check' to check attendance, or 'exit' to quit: check
Enter student ID: 4
present
Enter 'mark' to mark attendance, 'check' to check attendance, or 'exit' to quit: exit
```