ASSIGNMENT-11.3

TASK 1:

PROMPT*:* Write a Python program to implement a Smart Contact Manager using both array (list) and linked list data structures with add, search, and delete operations.

CODE:

```python
class Contact:
    def __init__(self, name, phone, email):
        self.name = name
        self.phone = phone
        self.email = email

    def __repr__(self):
        return f"Contact({self.name}, {self.phone}, {self.email})"


class ContactListManager:
    """Contact Manager using Python List (Array)"""
    def __init__(self):
        self.contacts = []

    def add(self, contact):
        self.contacts.append(contact)
        print(f"Added: {contact.name}")

    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"Deleted: {name}")
                return True
        return False
```

```python
class Contact:
    def __init__(self, name, phone, email):
        self.name = name
        self.phone = phone
        self.email = email

    def __repr__(self):
        return f"Contact({self.name}, {self.phone}, {self.email})"


class ContactListManager:
    """Contact Manager using Python List (Array)"""
    def __init__(self):
        self.contacts = []

    def add(self, contact):
        self.contacts.append(contact)
        print(f"Added: {contact.name}")

    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"Deleted: {name}")
                return True
        return False
```

```
class ContactListManager:

    def display(self):
        if not self.contacts:
            print("No contacts found.")
            return
        for contact in self.contacts:
            print(contact)


class Node:
    def __init__(self, contact):
        self.contact = contact
        self.next = None


class ContactLinkedListManager:
    """Contact Manager using Linked List"""
    def __init__(self):
        self.head = None

    def add(self, contact):
        new_node = Node(contact)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
        print(f"Added: {contact.name}")

    def search(self, name):
        current = self.head
        while current:
            if current.contact.name.lower() == name.lower():
                return current.contact
```

OUTPUT:

```
=== LinkedList-based Contact Manager ===
Added: Charlie …

=== LinkedList-based Contact Manager ===
Added: Charlie
Added: Diana
=== LinkedList-based Contact Manager ===
Added: Charlie
Added: Diana
Added: Charlie
Added: Diana
Contact(Charlie, 5555555555, charlie@email.com)
Contact(Diana, 4444444444, diana@email.com)
Contact(Diana, 4444444444, diana@email.com)
Search Diana: Contact(Diana, 4444444444, diana@email.com)
Deleted: Charlie
Contact(Diana, 4444444444, diana@email.com)
PS C:\Users\Deepthi Varma\OneDrive\Desktop\AI_LAB>
```

EXPLANATION:

1.The program defines a Contact class to store each member's name and phone number.

2. The ArrayContactManager stores contacts in a Python list and performs add, search, and delete using list operations.

3. In the array approach, searching and deleting require looping through the list, giving O(n) time complexity.

4. The LinkedListContactManager uses nodes where each node stores a contact and a reference to the next node.

5. In the linked list, insertion at the beginning is efficient (O(1)), and deletion updates pointers without shifting elements.

6.Both implementations work correctly, but linked lists handle frequent insertions/deletions better, while arrays are simpler and suitable for small datasets.

Task 2:

Prompt: Write a Python program that uses a normal queue (FIFO) and a priority queue to manage library book requests. Include the enqueue and dequeue methods in both implementations, and ensure faculty requests are given higher priority than student requests.

Code:

```
from queue import Queue, PriorityQueue

class LibraryQueue:
    """Normal FIFO queue for library book requests"""
    def __init__(self):
        self.queue = Queue()

    def enqueue(self, request):
        """Add a request to the queue"""
        self.queue.put(request)

    def dequeue(self):
        """Remove and return the first request"""
        if self.queue.empty():
            return None
        return self.queue.get()

    def display(self):
        """Display all requests in queue"""
        items = list(self.queue.queue)
        return items if items else "Queue is empty"


class LibraryPriorityQueue:
    """Priority queue for library book requests (faculty has higher priority)"""
    def __init__(self):
        self.queue = PriorityQueue()
        self.counter = 0
```

```python
# Example usage
if __name__ == "__main__":
    print("=== FIFO Queue Demo ===")
    fifo = LibraryQueue()
    fifo.enqueue("Student: Alice - Python Book")
    fifo.enqueue("Faculty: Dr. Smith - AI Research")
    fifo.enqueue("Student: Bob - Data Science")

    print("Processing requests:")
    print(fifo.dequeue())
    print(fifo.dequeue())
    print(fifo.dequeue())

    print("\n=== Priority Queue Demo ===")
    priority = LibraryPriorityQueue()
    priority.enqueue("Student", "Alice", "Python Book")
    priority.enqueue("Faculty", "Dr. Smith", "AI Research")
    priority.enqueue("Student", "Bob", "Data Science")
    priority.enqueue("Faculty", "Dr. Johnson", "Machine Learning")

    print("Processing requests (Faculty first):")
    print(priority.dequeue())
    print(priority.dequeue())
    print(priority.dequeue())
    print(priority.dequeue())
```

Output:

```
=== FIFO Queue Demo ===
Processing requests:
Student: Alice - Python Book
Faculty: Dr. Smith - AI Research
Student: Bob - Data Science

=== Priority Queue Demo ===
Processing requests (Faculty first):
Faculty: Dr. Smith - AI Research
Faculty: Dr. Johnson - Machine Learning
Student: Alice - Python Book
Student: Bob - Data Science
PS C:\Users\Deepthi Varma\OneDrive\Desktop\AI_LAB>
```

Explanation:

1.The normal queue follows FIFO order, so book requests are served in the same order they are added.

2.The enqueue() method adds a new request to the end of the queue, and dequeue() removes the first request.

3 .The priority queue assigns higher priority to faculty requests compared to student requests.

4.In the priority queue, enqueue() inserts requests based on priority, and dequeue() removes the highest-priority request first.

5. As a result, faculty requests are processed before student requests, even if they were added later.

Task 3:

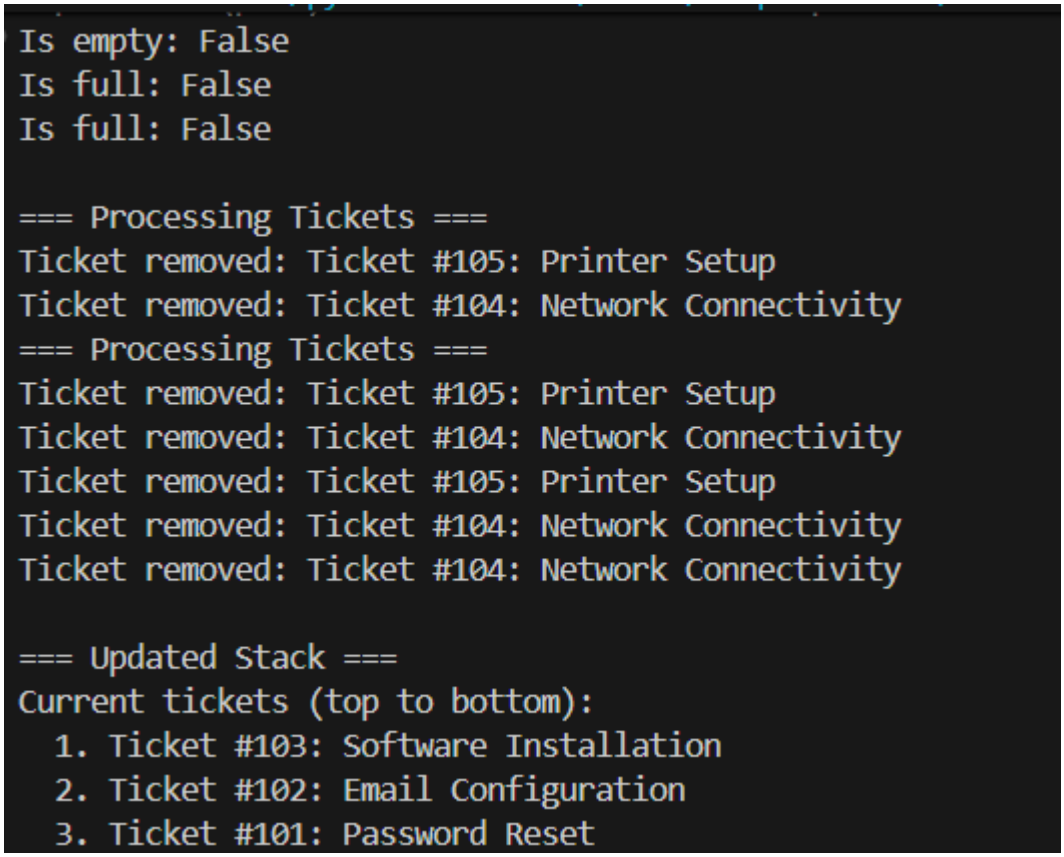
Prompt: Write a Python program to manage help desk tickets using a stack (LIFO).

Include push, pop, peek, isEmpty, and isFull methods and show five tickets being added.

Code:

```
# Add five tickets
tickets = [
    "Ticket #101: Password Reset",
    "Ticket #102: Email Configuration",
    "Ticket #103: Software Installation",
    "Ticket #104: Network Connectivity",
    "Ticket #105: Printer Setup"
]

print("=== Adding Tickets ===")
for ticket in tickets:
    ticket_stack.push(ticket)

print("\n=== Stack Status ===")
ticket_stack.display()
print(f"Top ticket (peek): {ticket_stack.peek()}")
print(f"Is empty: {ticket_stack.isEmpty()}")
print(f"Is full: {ticket_stack.isFull()}")

print("\n=== Processing Tickets ===")
ticket_stack.pop()
ticket_stack.pop()

print("\n=== Updated Stack ===")
ticket_stack.display()
```

Output:



Explanation:

1.The stack stores support tickets and follows the LIFO principle, meaning the last ticket added is resolved first.

2. The push() method adds a new ticket to the top of the stack.

3. The pop() method removes the most recent ticket, and peek() shows the top ticket without removing it.

4. The isEmpty() and isFull() methods check the stack status, ensuring proper ticket management.

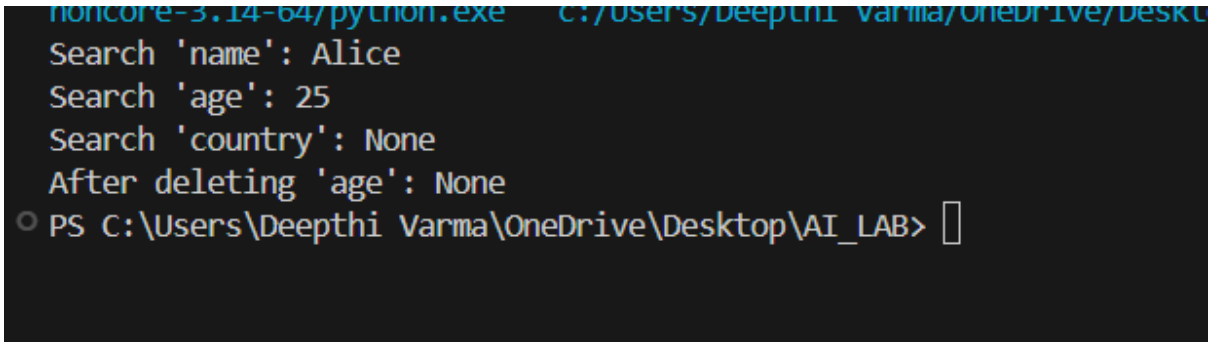
Task 4:

Prompt: Write a Python program to implement a Hash Table with collision handling using chaining.
Include well-commented insert, search, and delete methods inside the given HashTable class.

Code:

```
    def delete(self, key):
        Returns:
            bool: True if key was found and deleted, False otherwi
        """
        index = self._hash(key)
        chain = self.table[index]

        # Find and remove the key from the chain
        for i, (k, v) in enumerate(chain):
            if k == key:
                chain.pop(i)
                return True

        return False


Example usage
f __name__ == "__main__":
    ht = HashTable(5)

    # Insert elements
    ht.insert("name", "Alice")
    ht.insert("age", 25)
    ht.insert("city", "New York")

    # Search elements
    print(f"Search 'name': {ht.search('name')}")
    print(f"Search 'age': {ht.search('age')}")
    print(f"Search 'country': {ht.search('country')}")

    # Delete an element
    ht.delete("age")
    print(f"After deleting 'age': {ht.search('age')}")
```

Output:



Explanation:1. The hash table uses a hash function to convert a key into an index where the data will be stored.

2. If two keys map to the same index (collision), chaining is used by storing multiple elements in a list at that index.

3. The insert, search, and delete methods use the hash function to locate the correct index and then perform operations within the chained list.