ASSIGNMENT-11.3

NAME : THATIKONDA SASYA
BATCH : 20
HTNO:2303A51346

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

```python
class ArrayContactManager:
    def __init__(self):
        self.contacts = []  # list of dictionaries

    def add_contact(self, name, phone):
        self.contacts.append({"name": name, "phone": phone})

    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, contact in enumerate(self.contacts):
            if contact["name"] == name:
                del self.contacts[i]
                return True
        return False


# Example usage
manager = ArrayContactManager()
manager.add_contact("Amit", "9876543210")
manager.add_contact("Rahul", "9123456789")

print(manager.search_contact("Amit"))
manager.delete_contact("Rahul")
print(manager.contacts)
```

```
{'name': 'Amit', 'phone': '9876543210'}
[{'name': 'Amit', 'phone': '9876543210'}]
```

```python
class ContactNode:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
        self.next = None
class LinkedListContactManager:
    def __init__(self):
        self.head = None
    def add_contact(self, name, phone):
        new_node = ContactNode(name, phone)
        new_node.next = self.head
        self.head = new_node
    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
                return True
            prev = current
            current = current.next
        return False
```

```
{'name': 'Amit', 'phone': '9876543210'}
[{'name': 'Amit', 'phone': '9876543210'}]
```

JUSTIFICATION:

The **array-based implementation** is simple and easy to use, making it suitable for small contact lists. However, deletion is inefficient because elements must be shifted after removal, leading to higher time complexity.

The **linked list implementation** provides better flexibility for dynamic memory allocation. Insertion is efficient as it does not require resizing, and deletion only involves pointer adjustments. However, linked lists consume more memory due to additional node pointers and have slower access times.

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

```python
from collections import deque
import heapq
class Request:
    def __init__(self, name, role, book):
        self.name = name
        self.role = role
        self.book = book
    def __str__(self):
        return f"{self.role} - {self.name} requested '{self.book}'"
class LibraryQueue:
    def __init__(self):
        self.queue = deque()
    def enqueue(self, request):
        self.queue.append(request)
    def dequeue(self):
        if not self.queue:
            return None
        return self.queue.popleft()
    def is_empty(self):
        return len(self.queue) == 0
class LibraryPriorityQueue:
    def __init__(self):
        self.pq = []
        self.count = 0
    def get_priority(self, role):
        if role.lower() == "faculty":
            return 0
        return 1
    def enqueue(self, request):
        priority = self.get_priority(request.role)
        heapq.heappush(self.pq, (priority, self.count, request))
        self.count += 1
```

```
    def enqueue(self, request):
        priority = self.get_priority(request.role)
        heapq.heappush(self.pq, (priority, self.count, request))
        self.count += 1
    def dequeue(self):
        if not self.pq:
            return None
        return heapq.heappop(self.pq)[2]
    def is_empty(self):
        return len(self.pq) == 0
requests = [
    Request("Alice", "Student", "Data Structures"),
    Request("Dr. Rao", "Faculty", "Artificial Intelligence"),
    Request("Bob", "Student", "Python Programming"),
    Request("Prof. Mehta", "Faculty", "Machine Learning"),
    Request("Charlie", "Student", "Algorithms")]
print("FIFO Queue Processing:")
fifo = LibraryQueue()
for r in requests:
    fifo.enqueue(r)
while not fifo.is_empty():
    print(fifo.dequeue())
print("\nPriority Queue Processing (Faculty First):")
pq = LibraryPriorityQueue()
for r in requests:
    pq.enqueue(r)
while not pq.is_empty():
    print(pq.dequeue())
```

```
FIFO Queue Processing:
Student - Alice requested 'Data Structures'
Faculty - Dr. Rao requested 'Artificial Intelligence'
Student - Bob requested 'Python Programming'
Faculty - Prof. Mehta requested 'Machine Learning'
Student - Charlie requested 'Algorithms'

Priority Queue Processing (Faculty First):
Faculty - Dr. Rao requested 'Artificial Intelligence'
Faculty - Prof. Mehta requested 'Machine Learning'
Student - Alice requested 'Data Structures'
Student - Bob requested 'Python Programming'
Student - Charlie requested 'Algorithms'
```

JUSTIFICATION:

The system uses a FIFO Queue to process book requests in the order they are received, ensuring fairness.A Priority Queue is added to prioritize faculty requests over student requests as required in the scenario.

Faculty are assigned higher priority (0) and students lower priority (1), so faculty requests are dequeued first.

The enqueue() and dequeue() methods correctly manage insertion and removal of requests in both structures.

Testing with mixed requests proves that FIFO works normally while the Priority Queue correctly gives preference to faculty.

Task 3: Emergency Help Desk (Stack Implementation)

```
class HelpDeskStack:
    def __init__(self, capacity=10):
        self.stack = []
        self.capacity = capacity
    def push(self, ticket):
        if self.is_full():
            print("Stack is full. Cannot add ticket.")
            return
        self.stack.append(ticket)
        print("Ticket Raised:", ticket)
    def pop(self):
        if self.is_empty():
            print("No tickets to resolve.")
            return None
        ticket = self.stack.pop()
        print("Ticket Resolved:", ticket)
        return ticket
    def peek(self):
        if self.is_empty():
            return "No active tickets"
        return self.stack[-1]
    def is_empty(self):
        return len(self.stack) == 0
    def is_full(self):
        return len(self.stack) == self.capacity
helpdesk = HelpDeskStack()
helpdesk.push("Ticket 1: WiFi not working")
helpdesk.push("Ticket 2: System crash")
helpdesk.push("Ticket 3: Printer issue")
helpdesk.push("Ticket 4: Email login error")
helpdesk.push("Ticket 5: Software installation")
print("\nCurrent Top Ticket:", helpdesk.peek())
print("\nResolving Tickets (LIFO Order):")
helpdesk.pop()
helpdesk.pop()
helpdesk.pop()
helpdesk.pop()
helpdesk.pop()
```

```
Ticket Raised: Ticket 1: WiFi not working
Ticket Raised: Ticket 2: System crash
Ticket Raised: Ticket 3: Printer issue
Ticket Raised: Ticket 4: Email login error
Ticket Raised: Ticket 5: Software installation

Current Top Ticket: Ticket 5: Software installation

Resolving Tickets (LIFO Order):
Ticket Resolved: Ticket 5: Software installation
Ticket Resolved: Ticket 4: Email login error
Ticket Resolved: Ticket 3: Printer issue
Ticket Resolved: Ticket 2: System crash
Ticket Resolved: Ticket 1: WiFi not working
```

JUSTIFICATION:

The Emergency Help Desk system uses a stack data structure because ticket escalation follows the LIFO (Last-In, First-Out) principle, meaning the most recently raised ticket is resolved first. The push() operation is used to add new support tickets to the stack as they arrive, while the pop() operation resolves tickets in reverse order of arrival, accurately simulating real escalation handling. The peek() function helps identify the current top priority ticket without removing it from the stack. Additional operations like is_empty() and is_full() ensure proper validation and prevent errors during ticket management. The simulation of five tickets clearly demonstrates correct LIFO behavior, fulfilling the requirement of a functional stack-based ticket management system.

Task 4: Hash Table

```python
class HashTable:
    def __init__(self, size=10):
        # Initialize hash table with empty buckets (lists for chaining)
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        # Hash function to convert key into an index
        return hash(key) % self.size
    def insert(self, key, value):
        # Insert key-value pair into the hash table
        index = self._hash(key)
        bucket = self.table[index]
        # Check if key already exists and update
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                print(f"Updated: ({key}, {value})")
                return
        # If key does not exist, append to bucket (chaining)
        bucket.append((key, value))
        print(f"Inserted: ({key}, {value})")
    def search(self, key):
        # Search for a key in the hash table
        index = self._hash(key)
        bucket = self.table[index]
        for k, v in bucket:
            if k == key:
                print(f"Found: ({key}, {v})")
                return v
        print(f"Key '{key}' not found")
        return None
    def delete(self, key):
        # Delete a key-value pair from the hash table
```

```
    def delete(self, key):
        index = self._hash(key)
        bucket = self.table[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                print(f"Deleted: ({key})")
                return
        print(f"Key '{key}' not found for deletion")
# Testing with collisions (chaining)
ht = HashTable(5)
ht.insert("Alice", "Student")
ht.insert("Bob", "Student")
ht.insert("Charlie", "Faculty")
ht.insert("Eve", "Student")
ht.insert("Mallory", "Faculty")
print()
ht.search("Charlie")
ht.search("Bob")
print()
ht.delete("Bob")
ht.search("Bob")
```

```
Inserted: (Alice, Student)
Inserted: (Bob, Student)
Inserted: (Charlie, Faculty)
Inserted: (Eve, Student)
Inserted: (Mallory, Faculty)

Found: (Charlie, Faculty)
Found: (Bob, Student)

Deleted: (Bob)
Key 'Bob' not found
```

JUSTIFICATION:

The hash table is implemented using key-value pairs to allow efficient insertion, searching, and deletion of data. Collision handling is managed through chaining, where each index stores a list of elements that hash to the same bucket, preventing data loss during collisions. The insert() method adds new elements or updates existing ones, while the search() method retrieves values quickly using the hash function. The delete() method removes the specified key from its bucket without affecting other entries. Overall, the implementation satisfies the objective by demonstrating efficient storage, collision handling, and well-structured hash table operations.

Task 5: Real-Time Application Challenge

```
from collections import deque

class CafeteriaOrderQueue:
    def __init__(self):
        self.orders = deque()

    def place_order(self, student_name, item):
        self.orders.append((student_name, item))
        print(f"Order placed: {student_name} - {item}")

    def process_order(self):
        if not self.orders:
            print("No orders to process.")
            return None
        order = self.orders.popleft()
        print(f"Order processed: {order[0]} - {order[1]}")
        return order

    def display_orders(self):
        if not self.orders:
            print("No pending orders.")
            return
        print("Pending Orders:")
        for order in self.orders:
            print(f"{order[0]} - {order[1]}")
# Example Usage
cafeteria = CafeteriaOrderQueue()
cafeteria.place_order("Amit", "Veg Sandwich")
cafeteria.place_order("Rahul", "Coffee")

cafeteria.display_orders()
cafeteria.process_order()
cafeteria.display_orders()
```

```
Order placed: Amit - Veg Sandwich
Order placed: Rahul - Coffee
Pending Orders:
Amit - Veg Sandwich
Rahul - Coffee
Order processed: Amit - Veg Sandwich
Pending Orders:
Rahul - Coffee
```

JUSTIFICATION:

The queue data structure is ideal for the cafeteria order system because food orders must be handled in the exact sequence in which they are placed. Using a FIFO (First-In-First-Out) approach ensures fairness and prevents order starvation. The deque structure in Python allows efficient insertion and removal operations with O(1) time complexity, making it suitable for real-time environments