ASSIGNMENT - 11.3

PROMPT: Smart Contact Manager (Arrays & Linked Lists)
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.

CODE:

```python
# --- ARRAY BASED IMPLEMENTATION ---
class ArrayContactManager:
    def __init__(self):
        # Python lists are dynamic arrays
        self.contacts = []

    def add_contact(self, name, phone):
        self.contacts.append({'name': name, 'phone': phone})
        print(f"[Array] Added: {name}")

    def search_contact(self, name):
        for contact in self.contacts:
            if contact['name'].lower() == name.lower():
                return f"[Array] Found: {contact['name']} - {contact['phone']}"
        return f"[Array] {name} not found."

    def delete_contact(self, name):
        for i in range(len(self.contacts)):
            if self.contacts[i]['name'].lower() == name.lower():
                removed = self.contacts.pop(i)
                print(f"[Array] Deleted: {removed['name']}")
                return True
        print(f"[Array] Delete failed: {name} not found.")
        return False
```

```python
# --- LINKED LIST IMPLEMENTATION ---
class Node:
    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 = Node(name, phone)
        # Adding to the beginning (O(1) efficiency)
        new_node.next = self.head
        self.head = new_node
        print(f"[Linked List] Added: {name}")

    def search_contact(self, name):
        current = self.head
        while current:
            if current.name.lower() == name.lower():
                return f"[Linked List] Found: {current.name} - {current.phone}"
            current = current.next
        return f"[Linked List] {name} not found."
```

[10]
✓ 0s

```python
    def delete_contact(self, name):
        current = self.head
        prev = None
        while current:
            if current.name.lower() == name.lower():
                if prev:
                    prev.next = current.next
                else:
                    self.head = current.next
                print(f"[Linked List] Deleted: {current.name}")
                return True
            prev = current
            current = current.next
        print(f"[Linked List] Delete failed: {name} not found.")
        return False

# --- EXECUTION / OUTPUT ---
if __name__ == "__main__":
    # Initialize both
    arr_manager = ArrayContactManager()
    ll_manager = LinkedListContactManager()

    # 1. Add Contacts
    for manager in [arr_manager, ll_manager]:
        manager.add_contact("Alice", "123-456")
        manager.add_contact("Bob", "987-654")
        manager.add_contact("Charlie", "555-777")
```

[10]
✓ 0s

```python
    # 2. Search Contacts
    print(arr_manager.search_contact("Bob"))
    print(ll_manager.search_contact("Bob"))
    print(arr_manager.search_contact("Eve")) # Testing non-existent

    print("-" * 30)

    # 3. Delete and Verify
    arr_manager.delete_contact("Alice")
    ll_manager.delete_contact("Alice")

    print(arr_manager.search_contact("Alice")) # Should show not found
```

```
[Array] Added: Alice
[Array] Added: Bob
[Array] Added: Charlie
[Linked List] Added: Alice
[Linked List] Added: Bob
[Linked List] Added: Charlie
------------------------------
[Array] Found: Bob - 987-654
[Linked List] Found: Bob - 987-654
[Array] Eve not found.
------------------------------
[Array] Deleted: Alice
[Linked List] Deleted: Alice
[Array] Alice not found.
```

EXPLANATION:

1. Array-Based (Python Lists)

- **Structure:** Stores contacts in a single, solid block of memory.
- **Search:** Uses a "Linear Search" (checking each index one by one).
- **Insertion/Deletion:** Slow if done at the beginning, because every other contact must be "shifted" over to fill or create a gap.
- **Efficiency:** Best for small lists or adding to the very end.

---

2. Linked List

- **Structure:** Each contact is a "Node" with a pointer (link) to the next one.
- **Search:** Must "walk" the chain from the start until the name is found.
- **Insertion/Deletion:** Very fast. You don't move any data; you just change where the "links" point to skip or add a person.
- **Efficiency:** Best for dynamic data where you frequently add/remove members from the middle or start

2. **PROMPT** : Library Book Search System (Queues & Priority Queues)

The SRU Library manages book borrow requests. Students and faculty submit

requests, but faculty requests must be prioritized over student requests.

CODE:

```python
[11]
✓ 0s
import heapq
from collections import deque

# --- 1. STANDARD QUEUE (FIFO) ---
class LibraryQueue:
    def __init__(self):
        self.queue = deque()

    def enqueue(self, name, user_type):
        self.queue.append((name, user_type))
        print(f"[Queue] Request added: {name} ({user_type})")

    def dequeue(self):
        if not self.queue:
            return "[Queue] No requests to process."
        name, user_type = self.queue.popleft()
        return f"[Queue] Processing: {name} ({user_type})"

# --- 2. PRIORITY QUEUE ---
class PriorityLibraryQueue:
    def __init__(self):
        self.pq = []
        # Priority: Faculty = 0 (Higher), Student = 1 (Lower)
        self.priority_map = {"Faculty": 0, "Student": 1}

    def enqueue(self, name, user_type):
        priority = self.priority_map.get(user_type, 2)
        # heapq is a min-heap, so lower numbers come out first
        heapq.heappush(self.pq, (priority, name))
```

```python
[11]
✓ 0s
    def dequeue(self):
        if not self.pq:
            return "[Priority] No requests to process."
        priority, name = heapq.heappop(self.pq)
        user_type = "Faculty" if priority == 0 else "Student"
        return f"[Priority] Processing: {name} ({user_type})"

# --- TESTING BOTH SYSTEMS ---
if __name__ == "__main__":
    requests = [
        ("Alice", "Student"),
        ("Dr. Smith", "Faculty"),
        ("Bob", "Student"),
        ("Prof. Jones", "Faculty")
    ]

    print("--- Testing Standard FIFO Queue ---")
    lib_q = LibraryQueue()
    for name, utype in requests:
        lib_q.enqueue(name, utype)
    for _ in range(4):
        print(lib_q.dequeue())

    print("\n--- Testing Priority Queue ---")
    p_lib_q = PriorityLibraryQueue()
    for name, utype in requests:
        p_lib_q.enqueue(name, utype)
    for _ in range(4):
        print(p_lib_q.dequeue())
```

OUTPUT:

```
    for _ in range(4):
        print(p_lib_q.dequeue())
```

```
--- Testing Standard FIFO Queue ---
[Queue] Request added: Alice (Student)
[Queue] Request added: Dr. Smith (Faculty)
[Queue] Request added: Bob (Student)
[Queue] Request added: Prof. Jones (Faculty)
[Queue] Processing: Alice (Student)
[Queue] Processing: Dr. Smith (Faculty)
[Queue] Processing: Bob (Student)
[Queue] Processing: Prof. Jones (Faculty)

--- Testing Priority Queue ---
[Priority] Request added: Alice (Student)
[Priority] Request added: Dr. Smith (Faculty)
[Priority] Request added: Bob (Student)
[Priority] Request added: Prof. Jones (Faculty)
[Priority] Processing: Dr. Smith (Faculty)
[Priority] Processing: Prof. Jones (Faculty)
[Priority] Processing: Alice (Student)
[Priority] Processing: Bob (Student)
```

EXPLANATION:

1. Standard Queue (FIFO)

- **Logic:** Operates on the "First-In, First-Out" principle, exactly like a real-world standing line.
- **Processing:** The order of service is determined strictly by the **time of arrival**. If a student arrives before a professor, the student is served first.
- **Use Case:** Best for fair, neutral systems where everyone has equal status.

---

2. Priority Queue

- **Logic:** Assigns a "weight" or "rank" to each request. In our code, Faculty is rank `0` and Student is rank `1`.
- **Processing:** The system always looks for the lowest rank number (highest priority) to serve next. This allows faculty members to "jump the line" regardless of when they arrived.
- **Use Case:** Ideal for systems with **emergency levels** or membership tiers (like the SRU Library).

3.PROMPT: 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.

OUTPUT:

```
[12]
✓ 0s
class TicketStack:
    def __init__(self, capacity=5):
        self.stack = []
        self.capacity = capacity

    def push(self, ticket_id, issue):
        """Adds a new ticket to the top of the stack."""
        if self.is_full():
            print(f"[Stack Full] Cannot add ticket {ticket_id}: {issue}")
            return
        self.stack.append({'id': ticket_id, 'issue': issue})
        print(f"Ticket Raised: {ticket_id} - {issue}")

    def pop(self):
        """Resolves and removes the most recent ticket (LIFO)."""
        if self.is_empty():
            return "No tickets to resolve."
        resolved = self.stack.pop()
        return f"Resolving: {resolved['id']} ({resolved['issue']})"

    def peek(self):
        """Views the ticket at the top without removing it."""
        if self.is_empty():
            return "Stack is empty."
        top = self.stack[-1]
        return f"Next for escalation: {top['id']} - {top['issue']}"
```

```python
    def is_empty(self):
        """Checks if there are no tickets."""
        return len(self.stack) == 0

    def is_full(self):
        """Checks if the stack has reached its limit."""
        return len(self.stack) >= self.capacity

# --- SIMULATION ---
if __name__ == "__main__":
    help_desk = TicketStack(capacity=5)

    # 1. Raising 5 tickets
    tickets = [
        ("T101", "WiFi Down"),
        ("T102", "Login Error"),
        ("T103", "Printer Jam"),
        ("T104", "Software Update"),
        ("T105", "Virus Alert")
    ]

    print("--- Receiving Tickets ---")
    for tid, issue in tickets:
        help_desk.push(tid, issue)

    # 2. Checking the top ticket
    print(f"\nPeek: {help_desk.peek()}")

    # 3. Resolving tickets (LIFO Behavior)
    print("\n--- Resolving Tickets (LIFO) ---")
```

```python
    # 3. Resolving tickets (LIFO Behavior)
    print("\n--- Resolving Tickets (LIFO) ---")
    while not help_desk.is_empty():
        print(help_desk.pop())
```

```
--- Receiving Tickets ---
Ticket Raised: T101 - WiFi Down
Ticket Raised: T102 - Login Error
Ticket Raised: T103 - Printer Jam
Ticket Raised: T104 - Software Update
Ticket Raised: T105 - Virus Alert

Peek: Next for escalation: T105 - Virus Alert

--- Resolving Tickets (LIFO) ---
Resolving: T105 (Virus Alert)
Resolving: T104 (Software Update)
Resolving: T103 (Printer Jam)
Resolving: T102 (Login Error)
Resolving: T101 (WiFi Down)
```

EXPLANATION:

LIFO Principle: The Help Desk operates like a stack of trays. The last ticket placed on top (T105) is the first one picked up to be fixed.

- Pop: Removes the top ticket once resolved.
- Peek: Checks the most urgent ticket without removing it from the system.
- Efficiency: Stacks are extremely fast for these operations ($O(1)$), as you never have to search through the middle; you only ever interact with the very top.

PROMPT 4: Hash Table

To implement a Hash Table and understand collision handling.

CODE:

```python
class HashTable:
    def __init__(self, size=10):
        self.size = size
        # Initialize the table with empty lists (buckets) for chaining
        self.table = [[] for _ in range(self.size)]

    def _hash_function(self, key):
        """Standard hash function using the modulo operator."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Inserts or updates a key-value pair."""
        index = self._hash_function(key)
        # Check if the key already exists in the bucket to update it
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        # If key is new, append it to the chain (collision handling)
        self.table[index].append([key, value])
        print(f"Inserted: '{key}' at Index {index}")

    def search(self, key):
        """Retrieves the value for a given key."""
        index = self._hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
```

```python
    def delete(self, key):
        """Removes a key-value pair from the table."""
        index = self._hash_function(key)
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                del self.table[index][i]
                print(f"Deleted: '{key}'")
                return True
        print(f"Error: '{key}' not found.")
        return False

    def display(self):
        """Prints the internal structure of the table."""
        print("\n--- Current Hash Table State ---")
        for i, bucket in enumerate(self.table):
            print(f"Index {i}: {bucket}")

# --- EXECUTION ---
if __name__ == "__main__":
    ht = HashTable(size=5)

    # 1. Insert data (Some will collide since size is only 5)
    ht.insert("Alice", "123-456")
    ht.insert("Bob", "987-654")
    ht.insert("Charlie", "555-000")
    ht.insert("David", "111-222") # High chance of collision

    # 2. Search
    print(f"\nSearch Result (Bob): {ht.search('Bob')}")
```

```python
# 2. Search
print(f"\nSearch Result (Bob): {ht.search('Bob')}")

# 3. Delete
ht.delete("Alice")

# 4. Final Display
ht.display()
```

```
Inserted: 'Alice' at Index 4
Inserted: 'Bob' at Index 3
Inserted: 'Charlie' at Index 1
Inserted: 'David' at Index 3

Search Result (Bob): 987-654
Deleted: 'Alice'

--- Current Hash Table State ---
Index 0: []
Index 1: [['Charlie', '555-000']]
Index 2: []
Index 3: [['Bob', '987-654'], ['David', '111-222']]
Index 4: []
```

EXPLANATION:

Chaining (Collision Resolution): When two keys (like "Alice" and "David") hash to the same index, we don't overwrite the data. Instead, we store them as a **nested list** (a "chain") at that index.

The Hash Function: The _hash_function takes any string and converts it into a number within the range of our table size using the % (modulo) operator.

Efficiency: * **Average Case:** $O(1)$ for insert, search, and delete. This is why hash tables are the foundation of Python's dict and set.

- **Worst Case:** $O(n)$ if every single key hashes to the exact same index, turning the hash table into one giant linked list.

PROMPT 5: Real-Time Application Challenge

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

CODE:

```
from collections import deque
import time

class CafeteriaSystem:
    def __init__(self):
        # Using deque for O(1) removals from the front
        self.order_queue = deque()

    def place_order(self, student_name, item):
        """Adds a new order to the end of the queue."""
        order = {"name": student_name, "item": item}
        self.order_queue.append(order)
        print(f"✅ Order placed for {student_name}: {item}")

    def serve_next_student(self):
        """Removes and serves the order at the front of the queue."""
        if not self.order_queue:
            print("⚠️ The queue is empty! No orders to serve.")
            return

        served_order = self.order_queue.popleft()
        print(f"🍽️ Serving {served_order['name']}'s {served_order['item']}...")
        time.sleep(1) # Simulating prep time
        print(f"✔️ {served_order['name']} has been served.")

    def view_queue(self):
        """Displays all pending orders."""
        print(f"\n--- Current Queue ({len(self.order_queue)} orders) ---")
```

```
[17]
✓ 2s

campus_cafe.place_order("Bob", "Avocado Toast")
campus_cafe.place_order("Charlie", "Blueberry Muffin")

# 2. View the current state of the line
campus_cafe.view_queue()

# 3. Process the orders in FIFO order
campus_cafe.serve_next_student()
campus_cafe.serve_next_student()

# 4. Final check of the queue
campus_cafe.view_queue()
```

```
✅ Order placed for Alice: Iced Latte
✅ Order placed for Bob: Avocado Toast
✅ Order placed for Charlie: Blueberry Muffin

--- Current Queue (3 orders) ---
1. Alice - Iced Latte
2. Bob - Avocado Toast
3. Charlie - Blueberry Muffin
----------------------------

🍽️ Serving Alice's Iced Latte...
✔️ Alice has been served.
🍽️ Serving Bob's Avocado Toast...
✔️ Bob has been served.

--- Current Queue (1 orders) ---
1. Charlie - Blueberry Muffin
----------------------------
```

EXPLANATION:

This system is designed to handle high-traffic campus operations by prioritizing data integrity and processing speed. By matching specific real-world tasks to optimized data structures, we ensure the software remains responsive even during peak hours, like class transitions or lunch breaks.

Logic Breakdown

- Efficiency: Using structures like Hash Maps for attendance and Queues for the cafeteria ensures that as the student body grows, the system's response time remains nearly instantaneous (constant time complexity).
- Reliability: The use of Sets for event registration prevents "double-booking" errors at the data level, reducing the need for complex manual checks.
- Flow Control: The FIFO (First-In, First-Out) logic implemented in the cafeteria code ensures a fair, chronological service order, which is the standard requirement for any physical or digital waiting line.