Assignment-11.3

Ht.No: 2303A51584

Batch: 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

Prompt:

Implement a Smart Contact Manager using singly linked list building from scratch in python. Each node contains:

name

phone number

next pointer.

implement the methods:

addContact,

searchContact (case-insensitive search by name),

deleteContact (should correctly handle deleting head, middle, or last node)

Return true if deletion is successful, otherwise false.

Code:

smart_contact_manager.py > ContactNode

```python
"""
Smart Contact Manager using a singly linked list
Provides functionality to add, search, and delete contacts
"""
class ContactNode:
    """Node class for the singly linked list"""

    def __init__(self, name, phone_number):
        """
        Constructor to create a new contact node
        Args:
            name: the contact's name
            phone_number: the contact's phone number
        """
        self.name = name
        self.phone_number = phone_number
        self.next = None

    def __str__(self):
        """String representation of the contact"""
        return f"Name: {self.name}, Phone: {self.phone_number}"


class SmartContactManager:
    """Smart Contact Manager using a singly linked list"""

    def __init__(self):
        """Constructor to initialize an empty contact list"""
        self.head = None

    def add_contact(self, name, phone_number):
        """
        Add a new contact to the linked list
        Contacts are added at the beginning of the list for O(1) time complexity

        Args:
            name: the contact's name
```

```
class SmartContactManager:
    def add_contact(self, name, phone_number):
            phone_number: the contact's phone number
        """
        if not name or not str(name).strip() or not phone_number or not str(phone_number).strip():
            print("Error: Name and phone number cannot be empty.")
            return

        name = str(name).strip()
        phone_number = str(phone_number).strip()
        new_node = ContactNode(name, phone_number)

        # Add at the beginning of the list
        new_node.next = self.head
        self.head = new_node

        print(f"Contact added successfully: {new_node}")

    def search_contact(self, name):
        """
        Search for a contact by name (case-insensitive)

        Args:
            name: the name to search for

        Returns:
            the ContactNode if found, None otherwise
        """
        if not name or not str(name).strip():
            print("Error: Name cannot be empty.")
            return None

        search_name = str(name).strip().lower()
        current = self.head

        while current is not None:
            # Case-insensitive comparison
```

```
def search_contact(self, name):
        # case-insensitive comparison
        if current.name.lower() == search_name:
            return current
        current = current.next

    return None

def delete_contact(self, name):
    """
    Delete a contact by name (case-insensitive)
    Handles deletion of head, middle, and last nodes correctly

    Args:
        name: the name of the contact to delete

    Returns:
        True if deletion was successful, False otherwise
    """
    if not name or not str(name).strip():
        print("Error: Name cannot be empty.")
        return False

    delete_name = str(name).strip().lower()

    # Case 1: Deleting the head node
    if self.head is not None and self.head.name.lower() == delete_name:
        self.head = self.head.next
        print(f"Contact deleted successfully: {name}")
        return True

    # Case 2 & 3: Deleting a middle or last node
    current = self.head
    previous = None

    while current is not None:
        if current.name.lower() == delete_name:
```

```
def delete_contact(self, name):
    while current is not None:
        if current.name.lower() == delete_name:
            # Found the node to delete
            if previous is not None:
                previous.next = current.next
                print(f"Contact deleted successfully: {name}")
                return True
        previous = current
        current = current.next

    # Contact not found
    print(f"Contact not found: {name}")
    return False

def display_all_contacts(self):
    """Display all contacts in the list"""
    if self.head is None:
        print("No contacts available.")
        return

    print("\n=== All Contacts ===")
    current = self.head
    count = 1

    while current is not None:
        print(f"{count}. {current}")
        current = current.next
        count += 1

    print("====================\n")

def get_total_contacts(self):
    """
    Get the total number of contacts
```

```python
def get_total_contacts(self):
    """
    Get the total number of contacts

    Returns:
        number of contacts in the list
    """
    count = 0
    current = self.head

    while current is not None:
        count += 1
        current = current.next

    return count

def get_phone_number(self, name):
    """
    Get a contact's phone number by name (case-insensitive)

    Args:
        name: the contact's name

    Returns:
        the phone number, or None if not found
    """
    contact = self.search_contact(name)
    if contact is not None:
        return contact.phone_number
    return None


f search_demo(manager, name):
"""Helper function to demonstrate search functionality"""
```

```python
def search_demo(manager, name):
    """Helper function to demonstrate search functionality"""
    contact = manager.search_contact(name)
    if contact is not None:
        print(f"Found: {contact}")
    else:
        print(f"Not found: {name}")


def main():
    """Demo/Main function to test the Smart Contact Manager"""
    manager = SmartContactManager()

    print("=== Smart Contact Manager Demo ===\n")

    # Demo 1: Add contacts
    print("--- Adding Contacts ---")
    manager.add_contact("Alice Johnson", "555-1234")
    manager.add_contact("Bob Smith", "555-5678")
    manager.add_contact("Charlie Brown", "555-9999")
    manager.add_contact("Diana Prince", "555-4444")

    manager.display_all_contacts()

    # Demo 2: Search contacts (case-insensitive)
    print("--- Searching Contacts (Case-Insensitive) ---")
    search_demo(manager, "alice johnson")
    search_demo(manager, "ALICE JOHNSON")
    search_demo(manager, "bob smith")
    search_demo(manager, "John Doe")

    # Demo 3: Get phone number
    print("--- Getting Phone Numbers ---")
    phone1 = manager.get_phone_number("charlie brown")
    print(f"Charlie Brown's phone: {phone1}")

    phone2 = manager.get_phone_number("NonExistent")
```

```python
def main():
    result1 = manager.delete_contact("diana prince")
    print(f"Deletion successful: {result1}")

    # Delete middle node
    print("\nDeleting middle node (Bob Smith):")
    result2 = manager.delete_contact("BOB SMITH")
    print(f"Deletion successful: {result2}")

    # Delete last node
    print("\nDeleting last node (Alice Johnson):")
    result3 = manager.delete_contact("alice johnson")
    print(f"Deletion successful: {result3}")

    # Try to delete non-existent contact
    print("\nTrying to delete non-existent contact:")
    result4 = manager.delete_contact("John Doe")
    print(f"Deletion successful: {result4}")

    manager.display_all_contacts()

    # Demo 5: Total contacts count
    print(f"Total contacts remaining: {manager.get_total_contacts()}\n")

    # Demo 6: Additional tests
    print("--- Additional Tests ---")
    manager.add_contact("Eve Wilson", "555-7777")
    manager.add_contact("Frank Miller", "555-8888")

    manager.display_all_contacts()

    print(f"Total contacts: {manager.get_total_contacts()}")


if __name__ == "__main__":
    main()
```

Output:

```
E:\3-2\AI Assisted Coding>C:/Users/hp/AppData/Local/Programs/Python/Python311/python.exe
er.py"
=== Smart Contact Manager Demo ===

--- Adding Contacts ---
Contact added successfully: Name: Alice Johnson, Phone: 555-1234
Contact added successfully: Name: Bob Smith, Phone: 555-5678
Contact added successfully: Name: Charlie Brown, Phone: 555-9999
Contact added successfully: Name: Diana Prince, Phone: 555-4444

=== All Contacts ===
1. Name: Diana Prince, Phone: 555-4444
2. Name: Charlie Brown, Phone: 555-9999
3. Name: Bob Smith, Phone: 555-5678
4. Name: Alice Johnson, Phone: 555-1234
====================

--- Searching Contacts (Case-Insensitive) ---
Found: Name: Alice Johnson, Phone: 555-1234
Found: Name: Alice Johnson, Phone: 555-1234
Found: Name: Bob Smith, Phone: 555-5678
Not found: John Doe
--- Getting Phone Numbers ---
Charlie Brown's phone: 555-9999
NonExistent's phone: None

--- Deleting Contacts ---

Deleting head node (Diana Prince):
Contact deleted successfully: diana prince
Deletion successful: True

Deleting middle node (Bob Smith):
Contact deleted successfully: BOB SMITH
Deletion successful: True

Deleting last node (Alice Johnson):
Contact deleted successfully: alice johnson
Deletion successful: True
```

```
Found: Name: Bob Smith, Phone: 555-5678
Not found: John Doe
--- Getting Phone Numbers ---
Charlie Brown's phone: 555-9999
NonExistent's phone: None

--- Deleting Contacts ---

Deleting head node (Diana Prince):
Contact deleted successfully: diana prince
Deletion successful: True

Deleting middle node (Bob Smith):
Contact deleted successfully: BOB SMITH
Deletion successful: True

Deleting last node (Alice Johnson):
Contact deleted successfully: alice johnson
Deletion successful: True

Trying to delete non-existent contact:
Contact not found: John Doe
Deletion successful: False

=== All Contacts ===
1. Name: Charlie Brown, Phone: 555-9999
====================

Total contacts remaining: 1

--- Additional Tests ---
Contact added successfully: Name: Eve Wilson, Phone: 555-7777
Contact added successfully: Name: Frank Miller, Phone: 555-8888

=== All Contacts ===
1. Name: Frank Miller, Phone: 555-8888
2. Name: Eve Wilson, Phone: 555-7777
3. Name: Charlie Brown, Phone: 555-9999
====================

Total contacts: 3
```

Using Arrays:

smart_contact_manager.py > SmartContactManager

```python
"""
Smart Contact Manager using Arrays/Lists
Provides functionality to add, search, and delete contacts
"""


class Contact:
    """Contact class to represent a contact with name and phone number"""

    def __init__(self, name, phone_number):
        """
        Constructor to create a new contact

        Args:
            name: the contact's name
            phone_number: the contact's phone number
        """
        self.name = name
        self.phone_number = phone_number

    def __str__(self):
        """String representation of the contact"""
        return f"Name: {self.name}, Phone: {self.phone_number}"


class SmartContactManager:
    """Smart Contact Manager using an array (list) to store contacts"""

    def __init__(self):
        """Constructor to initialize an empty contact list"""
        self.contacts = []

    def add_contact(self, name, phone_number):
        """
        Add a new contact to the array

        Args:
            name: the contact's name
            phone_number: the contact's phone number
        """
        if not name or not str(name).strip() or not phone_number or not str(phone_number).strip():
            print("Error: Name and phone number cannot be empty.")
            return

        name = str(name).strip()
        phone_number = str(phone_number).strip()

        # Create new contact and add to array
        new_contact = Contact(name, phone_number)
        self.contacts.append(new_contact)

        print(f"Contact added successfully: {new_contact}")

    def search_contact(self, name):
        """
        Search for a contact by name (case-insensitive)

        Args:
            name: the name to search for

        Returns:
            the Contact object if found, None otherwise
        """
        if not name or not str(name).strip():
            print("Error: Name cannot be empty.")
            return None

        search_name = str(name).strip().lower()

        # Linear search through the array
        for contact in self.contacts:
            if contact.name.lower() == search_name:
                return contact

        return None

    def delete_contact(self, name):
        """
        Delete a contact by name (case-insensitive)
        Handles deletion of head, middle, or last contact correctly

```

```python
def search_demo(manager, name):
    """Helper function to demonstrate search functionality"""
    contact = manager.search_contact(name)
    if contact is not None:
        print(f"Found: {contact}")
    else:
        print(f"Not found: {name}")


def main():
    """Demo/Main function to test the Smart Contact Manager"""
    manager = SmartContactManager()

    print("=== Smart Contact Manager Demo ===\n")

    # Demo 1: Add contacts
    print("--- Adding Contacts ---")
    manager.add_contact("Alice Johnson", "555-1234")
    manager.add_contact("Bob Smith", "555-5678")
    manager.add_contact("Charlie Brown", "555-9999")
    manager.add_contact("Diana Prince", "555-4444")

    manager.display_all_contacts()

    # Demo 2: Search contacts (case-insensitive)
    print("--- Searching Contacts (Case-Insensitive) ---")
    search_demo(manager, "alice johnson")
    search_demo(manager, "ALICE JOHNSON")
    search_demo(manager, "bob smith")
    search_demo(manager, "John Doe")

    # Demo 3: Get phone number
    print("--- Getting Phone Numbers ---")
    phone1 = manager.get_phone_number("charlie brown")
    print(f"Charlie Brown's phone: {phone1}")

    phone2 = manager.get_phone_number("NonExistent")
    print(f"NonExistent's phone: {phone2}\n")

    # Demo 4: Delete contacts
    print("--- Deleting Contacts ---")

    # Delete first contact (head position)
    print("\nDeleting first contact (Alice Johnson):")
    result1 = manager.delete_contact("alice johnson")
    print(f"Deletion successful: {result1}")

    # Delete middle contact
    print("\nDeleting middle contact (Bob Smith):")
    result2 = manager.delete_contact("BOB SMITH")
    print(f"Deletion successful: {result2}")

    # Delete last contact
    print("\nDeleting last contact (Charlie Brown):")
    result3 = manager.delete_contact("charlie brown")
    print(f"Deletion successful: {result3}")

    # Try to delete non-existent contact
    print("\nTrying to delete non-existent contact:")
    result4 = manager.delete_contact("John Doe")
    print(f"Deletion successful: {result4}")

    manager.display_all_contacts()

    # Demo 5: Total contacts count
    print(f"Total contacts remaining: {manager.get_total_contacts()}\n")

    # Demo 6: Additional tests
    print("--- Additional Tests ---")
    manager.add_contact("Eve Wilson", "555-7777")
    manager.add_contact("Frank Miller", "555-8888")

    manager.display_all_contacts()

    print(f"Total contacts: {manager.get_total_contacts()}")


if __name__ == "__main__":
    main()
```

```
Total contacts: 3

E:\3-2\AI Assisted Coding>C:/Users/hp/AppData/Local/Programs/Python/Python311/python.exe "e:/3-2/AI Assisted Coding/smart_contact_manager.py"
=== Smart Contact Manager Demo ===

--- Adding Contacts ---
Contact added successfully: Name: Alice Johnson, Phone: 555-1234
Contact added successfully: Name: Bob Smith, Phone: 555-5678
Contact added successfully: Name: Charlie Brown, Phone: 555-9999
Contact added successfully: Name: Diana Prince, Phone: 555-4444

=== All Contacts ===
1. Name: Alice Johnson, Phone: 555-1234
2. Name: Bob Smith, Phone: 555-5678
3. Name: Charlie Brown, Phone: 555-9999
4. Name: Diana Prince, Phone: 555-4444
====================

--- Searching Contacts (Case-Insensitive) ---
Found: Name: Alice Johnson, Phone: 555-1234
Found: Name: Alice Johnson, Phone: 555-1234
Found: Name: Bob Smith, Phone: 555-5678
Not found: John Doe
--- Getting Phone Numbers ---
Charlie Brown's phone: 555-9999
NonExistent's phone: None

--- Deleting Contacts ---

Deleting first contact (Alice Johnson):
Contact deleted successfully: alice johnson
Deletion successful: True

Deleting middle contact (Bob Smith):
Contact deleted successfully: BOB SMITH
Deletion successful: True

Deleting last contact (Charlie Brown):
Contact deleted successfully: charlie brown
Deletion successful: True

Trying to delete non-existent contact:
Contact not found: John Doe
Deletion successful: False

=== All Contacts ===
1. Name: Diana Prince, Phone: 555-4444
====================

Total contacts remaining: 1

--- Additional Tests ---
Contact added successfully: Name: Eve Wilson, Phone: 555-7777
Contact added successfully: Name: Frank Miller, Phone: 555-8888

=== All Contacts ===
1. Name: Diana Prince, Phone: 555-4444
2. Name: Eve Wilson, Phone: 555-7777
3. Name: Frank Miller, Phone: 555-8888
====================

Total contacts: 3

E:\3-2\AI Assisted Coding>
```

Explaination:

In an array, adding at the end is fast, but inserting in the middle is slow because elements must shift.

• In a linked list, insertion is fast because no shifting is needed.

• Searching takes the same time in both (you must check each element).

• Deleting in an array is slower due to shifting elements.

• Linked list is better for frequent insertions and deletions.

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.

Prompt:

build a Library Book Request System where both students and faculty can request books. First, I want to implement a normal Queue (FIFO) to process requests in the order they come. Then, extend it using a Priority Queue where faculty requests are processed before student requests. Generate the full working code

in a single file, including enqueue() and dequeue() methods, and test it with a mix of student and faculty requests.

Code:

```
"""
Library Book Request System
Implements both FIFO Queue and Priority Queue for processing book requests
Faculty requests have priority over student requests
"""

from collections import deque
import heapq


# ============================================================================
# CLASS: BookRequest - Represents a single book request
# ============================================================================
class BookRequest:
    """Stores information about a book request"""

    def __init__(self, requester_name, requester_type, book_title):
        """
        Initialize a book request

        Args:
            requester_name (str): Name of the person requesting the book
            requester_type (str): Type of requester - "Student" or "Faculty"
            book_title (str): Title of the book being requested
        """
        self.requester_name = requester_name
        self.requester_type = requester_type  # "Student" or "Faculty"
        self.book_title = book_title

    def __repr__(self):
        return f"Request({self.requester_name}, {self.requester_type}, '{self.book_title}')"

    def display(self):
        """Pretty print the request details"""
        return f"[{self.requester_type:8}] {self.requester_name:20} -> '{self.book_title}'"


# ============================================================================
# CLASS: NormalQueue - FIFO Queue Implementation
# ============================================================================
class NormalQueue:
    """
    Normal Queue - First In First Out (FIFO)
    Processes requests in the order they arrive
    """

    def __init__(self):
        """Initialize an empty queue using deque for efficient operations"""
        self.queue = deque()
```

```python
class NormalQueue:
    def __init__(self):
        """Initialize an empty queue using deque for efficient operations"""
        self.queue = deque()

    def enqueue(self, request):
        """
        Add a request to the end of the queue

        Args:
            request (BookRequest): The request to add
        """
        self.queue.append(request)

    def dequeue(self):
        """
        Remove and return the request from the front of the queue

        Returns:
            BookRequest: The first request in the queue
            None: If queue is empty
        """
        if not self.is_empty():
            return self.queue.popleft()
        return None

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

    def size(self):
        """Return the number of requests in the queue"""
        return len(self.queue)

    def display_all(self):
        """Display all requests in the queue"""
        if self.is_empty():
            print("Queue is empty!")
            return

        print("--- Queue Contents (FIFO Order) ---")
        for i, request in enumerate(self.queue, 1):
            print(f"  {i}. {request.display()}")

    def process_all(self):
        """Process and display all requests in order"""
        print("\n--- Processing Queue (FIFO) ---")
        count = 0
```

```
  def process_all(self):
      """Process and display all requests in order"""
      print("\n--- Processing Queue (FIFO) ---")
      count = 0
      while not self.is_empty():
          request = self.dequeue()
          count += 1
          print(f"  {count}. Processing: {request.display()}")
      print(f"All {count} requests processed!\n")


# ============================================================================
# CLASS: PriorityQueue - Priority-based Queue Implementation
# ============================================================================
class BookRequestPriorityQueue:
  """
  Priority Queue - Processes faculty requests before student requests
  Faculty = Priority 0, Student = Priority 1 (lower number = higher priority)
  Uses Python's heapq for efficient priority queue operations
  """

  def __init__(self):
      """Initialize an empty priority queue"""
      self.queue = []
      self.counter = 0  # Tie-breaker for requests with same priority

  def enqueue(self, request):
      """
      Add a request to the priority queue
      Faculty requests (priority=0) are processed before Student requests (priority=1)

      Args:
          request (BookRequest): The request to add
      """
      # Determine priority: Faculty gets 0 (higher priority), Student gets 1
      priority = 0 if request.requester_type == "Faculty" else 1

      # Use counter as tie-breaker to maintain FIFO order for same priority
      heapq.heappush(self.queue, (priority, self.counter, request))
      self.counter += 1

  def dequeue(self):
      """
      Remove and return the highest priority request

      Returns:
          BookRequest: The request with highest priority (faculty first, then by arrival)
```

```
  def dequeue(self):
      Returns:
          BookRequest: The request with highest priority (faculty first, then by arrival)
          None: If queue is empty
      """
      if not self.is_empty():
          priority, order, request = heapq.heappop(self.queue)
          return request
      return None

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

  def size(self):
      """Return the number of requests in the queue"""
      return len(self.queue)

  def display_all(self):
      """Display all requests in priority order"""
      if self.is_empty():
          print("Queue is empty!")
          return

      print("--- Priority Queue Contents (Faculty First) ---")
      # Create a sorted copy to display
      sorted_queue = sorted(self.queue, key=lambda x: (x[0], x[1]))
      for i, (priority, order, request) in enumerate(sorted_queue, 1):
          print(f"  {i}. {request.display()}")

  def process_all(self):
      """Process and display all requests in priority order"""
      print("\n--- Processing Priority Queue (Faculty First) ---")
      count = 0
      while not self.is_empty():
          request = self.dequeue()
          count += 1
          print(f"  {count}. Processing: {request.display()}")
      print(f"All {count} requests processed!\n")


# ============================================================================
# TEST FUNCTION: Demonstrate both queue implementations
# ============================================================================
ef test_library_request_system():
  """Test the Library Book Request System with sample data"""
```

```python
def test_library_request_system():
    print("=" * 70)
    print("LIBRARY BOOK REQUEST SYSTEM - DEMONSTRATION")
    print("=" * 70)

    # Create sample requests with mixed student and faculty types
    requests_data = [
        ("Alice Johnson", "Student", "Python Programming"),
        ("Dr. Smith", "Faculty", "Advanced Algorithms"),
        ("Bob Wilson", "Student", "Data Science Basics"),
        ("Prof. Davis", "Faculty", "Machine Learning Theory"),
        ("Charlie Brown", "Student", "Web Development"),
        ("Dr. Lee", "Faculty", "Artificial Intelligence"),
        ("Emma Davis", "Student", "Database Design"),
    ]

    # ======================================================================
    # TEST 1: Normal Queue (FIFO)
    # ======================================================================
    print("\n" + "=" * 70)
    print("TEST 1: NORMAL QUEUE (FIFO) - Process in Order of Arrival")
    print("=" * 70)

    normal_queue = NormalQueue()

    # Enqueue all requests
    print("\nEnqueueing requests in arrival order...")
    for name, req_type, book in requests_data:
        request = BookRequest(name, req_type, book)
        normal_queue.enqueue(request)
        print(f"  + {request.display()}")

    print(f"\nQueue size: {normal_queue.size()}")
    normal_queue.display_all()

    # Process all requests
    normal_queue.process_all()

    # ======================================================================
    # TEST 2: Priority Queue (Faculty First)
    # ======================================================================
    print("=" * 70)
    print("TEST 2: PRIORITY QUEUE - Faculty Requests Processed First")
    print("=" * 70)

    priority_queue = BookRequestPriorityQueue()

    # Enqueue all requests (same order as before)
```

```python
def test_library_request_system():
    test_requests = [
        BookRequest("Tom Harris", "Student", "Programming Basics"),
        BookRequest("Dr. Anderson", "Faculty", "Research Methods"),
        BookRequest("Lisa King", "Student", "Cloud Computing"),
    ]

    for req in test_requests:
        interactive_queue.enqueue(req)
        print(f"  Added: {req.display()}")

    print(f"\nCurrent queue size: {interactive_queue.size()}")

    # Dequeue requests one by one
    print("\nDequeuing requests individually...")
    while not interactive_queue.is_empty():
        request = interactive_queue.dequeue()
        print(f"  Dequeued: {request.display()}")

    print(f"\nFinal queue size: {interactive_queue.size()}")
    print("Queue is empty!" if interactive_queue.is_empty() else "Queue still has requests")

    # ======================================================================
    # Summary
    # ======================================================================
    print("\n" + "=" * 70)
    print("TEST SUMMARY")
    print("=" * 70)
    print("""
FIFO Queue:
  - Processes requests in the exact order they arrive
  - Ignores requester type
  - Fair, but may not be efficient for urgent needs

Priority Queue:
  - Processes faculty requests (priority 0) before student requests (priority 1)
  - Faculty get faster service
  - Within same priority level, maintains FIFO order (order of arrival)
  - More flexible for different requirement levels
    """)
    print("=" * 70 + "\n")


# ============================================================================
# MAIN EXECUTION
# ============================================================================
if __name__ == "__main__":
    test_library_request_system()
```

Output:

```
======================================================================
LIBRARY BOOK REQUEST SYSTEM - DEMONSTRATION
======================================================================


======================================================================
TEST 1: NORMAL QUEUE (FIFO) - Process in Order of Arrival
======================================================================

Enqueueing requests in arrival order...
  + [Student ] Alice Johnson        -> 'Python Programming'
  + [Faculty ] Dr. Smith            -> 'Advanced Algorithms'
  + [Student ] Bob Wilson           -> 'Data Science Basics'
  + [Faculty ] Prof. Davis          -> 'Machine Learning Theory'
  + [Student ] Charlie Brown        -> 'Web Development'
  + [Faculty ] Dr. Lee              -> 'Artificial Intelligence'
  + [Student ] Emma Davis           -> 'Database Design'

Queue size: 7
--- Queue Contents (FIFO Order) ---
  1. [Student ] Alice Johnson        -> 'Python Programming'
  2. [Faculty ] Dr. Smith            -> 'Advanced Algorithms'
  3. [Student ] Bob Wilson           -> 'Data Science Basics'
  4. [Faculty ] Prof. Davis          -> 'Machine Learning Theory'
  5. [Student ] Charlie Brown        -> 'Web Development'
  6. [Faculty ] Dr. Lee              -> 'Artificial Intelligence'
  7. [Student ] Emma Davis           -> 'Database Design'

--- Processing Queue (FIFO) ---
  1. Processing: [Student ] Alice Johnson        -> 'Python Programming'
  2. Processing: [Faculty ] Dr. Smith            -> 'Advanced Algorithms'
  3. Processing: [Student ] Bob Wilson           -> 'Data Science Basics'
  4. Processing: [Faculty ] Prof. Davis          -> 'Machine Learning Theory'
  5. Processing: [Student ] Charlie Brown        -> 'Web Development'
  6. Processing: [Faculty ] Dr. Lee              -> 'Artificial Intelligence'
  7. Processing: [Student ] Emma Davis           -> 'Database Design'
All 7 requests processed!

======================================================================
TEST 2: PRIORITY QUEUE - Faculty Requests Processed First
======================================================================

Enqueueing requests (same arrival order as Test 1)...
  + [Student ] Alice Johnson        -> 'Python Programming'
  + [Faculty ] Dr. Smith            -> 'Advanced Algorithms'
  + [Student ] Bob Wilson           -> 'Data Science Basics'
```

```
  7. [Student ] Emma Davis           -> 'Database Design'

--- Processing Priority Queue (Faculty First) ---
  1. Processing: [Faculty ] Dr. Smith            -> 'Advanced Algorithms'
  2. Processing: [Faculty ] Prof. Davis          -> 'Machine Learning Theory'
  3. Processing: [Faculty ] Dr. Lee              -> 'Artificial Intelligence'
  4. Processing: [Student ] Alice Johnson        -> 'Python Programming'
  5. Processing: [Student ] Bob Wilson           -> 'Data Science Basics'
  6. Processing: [Student ] Charlie Brown        -> 'Web Development'
  7. Processing: [Student ] Emma Davis           -> 'Database Design'
All 7 requests processed!

======================================================================
TEST 3: INTERACTIVE OPERATIONS - Adding and Removing Requests
======================================================================

Creating a new priority queue and adding requests individually...
  Added: [Student ] Tom Harris           -> 'Programming Basics'
  Added: [Faculty ] Dr. Anderson         -> 'Research Methods'
  Added: [Student ] Lisa King            -> 'Cloud Computing'

Current queue size: 3

Dequeuing requests individually...
  Dequeued: [Faculty ] Dr. Anderson         -> 'Research Methods'
  Dequeued: [Student ] Tom Harris           -> 'Programming Basics'
  Dequeued: [Student ] Lisa King            -> 'Cloud Computing'

Final queue size: 0
Queue is empty!

======================================================================
TEST SUMMARY
======================================================================

FIFO Queue:
  - Processes requests in the exact order they arrive
  - Ignores requester type
  - Fair, but may not be efficient for urgent needs

Priority Queue:
  - Processes faculty requests (priority 0) before student requests (priority 1)
  - Faculty get faster service
  - Within same priority level, maintains FIFO order (order of arrival)
  - More flexible for different requirement levels
```

Explaination:

Queue (FIFO) → First request comes, first served.(If a student requests first, they get the book first.)

• Priority Queue → Faculty requests are served before students, even if they come later.

• enqueue() → Adds a request to the system.

• dequeue() → Removes and processes the next request.

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)

Prompt:

Build an IT Help Desk Ticket System where support tickets are received sequentially, but issue resolution follows a Last-In, First-Out (LIFO) approach. First, implement a Stack to manage the tickets. The system should support push(ticket), pop(), and peek() operations. Then, simulate at least five tickets being raised and resolved. Also include additional stack operations such as checking whether the stack is empty and checking whether the stack is full (if applicable). Generate the full working code in a single file.

Code:

```
"""
IT Help Desk Ticket System
Implements a Stack (LIFO) to manage support tickets
Tickets are received sequentially but resolved in Last-In, First-Out order
"""


# ============================================================================
# CLASS: SupportTicket - Represents a single support ticket
# ============================================================================
class SupportTicket:
    """Stores information about an IT support ticket"""

    # Class variable to auto-generate ticket IDs
    ticket_counter = 1000

    def __init__(self, user_name, issue_type, description):
        """
        Initialize a support ticket

        Args:
            user_name (str): Name of the user reporting the issue
            issue_type (str): Type of issue (e.g., "Hardware", "Software", "Network")
            description (str): Description of the problem
        """
        SupportTicket.ticket_counter += 1
        self.ticket_id = SupportTicket.ticket_counter
        self.user_name = user_name
        self.issue_type = issue_type
        self.description = description
        self.status = "Open"  # Open, In Progress, Resolved

    def __repr__(self):
        return f"Ticket#{self.ticket_id}"

    def display(self):
        """Pretty print the ticket details"""
        return f"[Ticket #{self.ticket_id}] {self.user_name:20} | {self.issue_type:12} | {self.description:30} | Status: {self.sta

    def get_details(self):
```

```python
class SupportTicket:
    def get_details(self):
        """Get full details of the ticket"""
        details = f"""
        Ticket ID: {self.ticket_id}
        User: {self.user_name}
        Issue Type: {self.issue_type}
        Description: {self.description}
        Status: {self.status}
        """
        return details


# ============================================================================
# CLASS: Stack - LIFO Stack Implementation for Support Tickets
# ============================================================================
class TicketStack:
    """
    Stack implementation for managing IT support tickets
    Last-In, First-Out (LIFO) - Most recently added ticket is resolved first
    """

    def __init__(self, max_size=100):
        """
        Initialize an empty stack with optional maximum size

        Args:
            max_size (int): Maximum number of tickets the stack can hold (default: 100)
        """
        self.items = []
        self.max_size = max_size

    def push(self, ticket):
        """
        Add a ticket to the top of the stack

        Args:
            ticket (SupportTicket): The ticket to add

        Returns:
            bool: True if ticket added successfully, False if stack is full
```

```python
class TicketStack:
    def push(self, ticket):
        if self.is_full():
            print(f"  ERROR: Stack is full! Cannot add ticket #{ticket.ticket_id}")
            return False

        self.items.append(ticket)
        return True

    def pop(self):
        """
        Remove and return the ticket from the top of the stack

        Returns:
            SupportTicket: The top ticket from the stack
            None: If stack is empty
        """
        if not self.is_empty():
            return self.items.pop()
        return None

    def peek(self):
        """
        View the top ticket without removing it

        Returns:
            SupportTicket: The top ticket in the stack
            None: If stack is empty
        """
        if not self.is_empty():
            return self.items[-1]
        return None

    def is_empty(self):
        """
        Check if the stack is empty

        Returns:
            bool: True if stack is empty, False otherwise
        """
        return len(self.items) == 0
```

```python
# ============================================================================
# CLASS: HelpDeskSystem - Main IT Help Desk System
# ============================================================================
class ITHelpDeskSystem:
    """Manages the IT Help Desk with stack-based ticket resolution"""

    def __init__(self, max_tickets=100):
        """
        Initialize the Help Desk System

        Args:
            max_tickets (int): Maximum tickets the system can hold
        """
        self.ticket_stack = TicketStack(max_size=max_tickets)
        self.resolved_tickets = []  # Keep track of resolved tickets

    def receive_ticket(self, user_name, issue_type, description):
        """
        Receive a new support ticket

        Args:
            user_name (str): Name of the user reporting the issue
            issue_type (str): Type of issue
            description (str): Description of the problem

        Returns:
            bool: True if ticket received successfully, False if stack is full
        """
        ticket = SupportTicket(user_name, issue_type, description)
        success = self.ticket_stack.push(ticket)

        if success:
            print(f"  ✓ Ticket received: {ticket.display()}")

        return success

    def resolve_next_ticket(self):
        """
        Resolve the most recently received ticket (LIFO order)
```

```
    return ticket

def peek_next_ticket(self):
    """Show what the next ticket to resolve will be without removing it"""
    ticket = self.ticket_stack.peek()

    if ticket is None:
        print("  No tickets in queue")
        return None

    print(f"  Next to resolve (LIFO): {ticket.display()}")
    return ticket

def view_pending_tickets(self):
    """Display all pending tickets in the stack"""
    print(f"\n  === Pending Tickets ({self.ticket_stack.size()}/{self.ticket_stack.get_capacity()}) ===")
    self.ticket_stack.display_all()

def view_resolved_tickets(self):
    """Display all resolved tickets"""
    if not self.resolved_tickets:
        print("\n  No tickets resolved yet!")
        return

    print(f"\n  === Resolved Tickets ({len(self.resolved_tickets)}) ===")
    for i, ticket in enumerate(self.resolved_tickets, 1):
        print(f"  {i}. {ticket.display()}")

def get_system_status(self):
    """Display current system status"""
    print("\n  === Help Desk System Status ===")
    print(f"  Total Pending Tickets: {self.ticket_stack.size()}")
    print(f"  Total Resolved Tickets: {len(self.resolved_tickets)}")
    print(f"  Stack Capacity: {self.ticket_stack.get_capacity()}")
    print(f"  Available Space: {self.ticket_stack.get_available_space()}")
    print(f"  Stack Empty: {self.ticket_stack.is_empty()}")
    print(f"  Stack Full: {self.ticket_stack.is_full()}")
```

```python
# ============================================================================
# TEST FUNCTION: Simulate IT Help Desk Operations
# ============================================================================
def test_it_help_desk_system():
    """Comprehensive test of the IT Help Desk Ticket System"""

    print("=" * 80)
    print("IT HELP DESK TICKET SYSTEM - STACK-BASED TICKET RESOLUTION (LIFO)")
    print("=" * 80)

    # Initialize the Help Desk System
    help_desk = ITHelpDeskSystem(max_tickets=50)

    # ========================================================================
    # TEST 1: Receive Multiple Tickets
    # ========================================================================
    print("\n" + "=" * 80)
    print("TEST 1: RECEIVING SUPPORT TICKETS")
    print("=" * 80)

    tickets_data = [
        ("John Smith", "Network", "Cannot connect to VPN"),
        ("Sarah Johnson", "Hardware", "Printer not responding on network"),
        ("Mike Chen", "Software", "MS Office license activation issue"),
        ("Emily Watson", "Hardware", "Monitor flickering intermittently"),
        ("David Brown", "Software", "Email client keeps crashing"),
    ]

    print("\nReceiving tickets in order...")
    for user, issue_type, description in tickets_data:
        help_desk.receive_ticket(user, issue_type, description)

    # Display current system status
    help_desk.get_system_status()

    # Display all pending tickets
    help_desk.view_pending_tickets()

    # ========================================================================
    # TEST 2: Peek at Next Ticket (Without Removing)
    # ========================================================================
```

```python
def test_it_help_desk_system():
    print("\n" + "=" * 80)
    print("TEST 2: PEEKING AT NEXT TICKET (LIFO - Most Recently Added)")
    print("=" * 80)

    print("\nChecking which ticket will be resolved next (without removing it)...")
    help_desk.peek_next_ticket()

    print(f"\nStack still has {help_desk.ticket_stack.size()} tickets")

    # ========================================================================
    # TEST 3: Resolve Tickets One by One
    # ========================================================================
    print("\n" + "=" * 80)
    print("TEST 3: RESOLVING TICKETS (LIFO ORDER - Last In, First Out)")
    print("=" * 80)

    print("\nResolving tickets in LIFO order...")
    resolution_count = 0

    while not help_desk.ticket_stack.is_empty():
        resolution_count += 1
        ticket = help_desk.resolve_next_ticket()
        print(f"  {resolution_count}. RESOLVED: {ticket.display()}")
        print(f"     Remaining tickets: {help_desk.ticket_stack.size()}")

    # ========================================================================
    # TEST 4: View Resolved and Pending Tickets
    # ========================================================================
    print("\n" + "=" * 80)
    print("TEST 4: VIEWING RESOLVED AND PENDING TICKETS")
    print("=" * 80)

    help_desk.view_resolved_tickets()
    help_desk.view_pending_tickets()
    help_desk.get_system_status()

    # ========================================================================
    # TEST 5: Stack Operations on Empty Stack
    # ========================================================================
    print("\n" + "=" * 80)
```

```python
def test_it_help_desk_system():
    print("\nAttempting to add 4th ticket when stack is full...")
    ticket = SupportTicket("User4", "Hardware", "Issue 4")
    success = small_stack.push(ticket)
    print(f"  Result: {success} (expected False when full)")

    # ======================================================================
    # SUMMARY
    # ======================================================================
    print("\n" + "=" * 80)
    print("TEST SUMMARY AND KEY CONCEPTS")
    print("=" * 80)
    print("""
STACK (LIFO) CHARACTERISTICS:
  • Last-In, First-Out: Most recently added item is processed first
  • push(item): Add item to top of stack
  • pop(): Remove and return item from top of stack
  • peek(): View top item without removing it
  • is_empty(): Check if stack is empty
  • is_full(): Check if stack reaches max capacity
  • size(): Get current number of items

USE CASE FOR HELP DESK:
  ✓ LIFO approach prioritizes recent issues (more likely related/urgent)
  ✓ Easy to handle interruptions (add high-priority ticket, resolve immediately)
  ✓ Natural for undo/redo operations in software
  ✗ May delay older tickets at bottom of stack

COMPARISON WITH QUEUE (FIFO):
  • Queue: Fair, processes in order of arrival (customer service model)
  • Stack: Recent requests first (interrupt-driven model)
    """)
    print("=" * 80 + "\n")


# ============================================================================
# MAIN EXECUTION
# ============================================================================
if __name__ == "__main__":
    test_it_help_desk_system()
```

Output:

```
TEST 1: RECEIVING SUPPORT TICKETS
================================================================================

Receiving tickets in order...
  ✓ Ticket received: [Ticket #1001] John Smith          | Network      | Cannot connect to VPN
| Status: Open
  ✓ Ticket received: [Ticket #1002] Sarah Johnson       | Hardware     | Printer not responding on netwo
rk | Status: Open
  ✓ Ticket received: [Ticket #1003] Mike Chen           | Software     | MS Office license activation is
sue | Status: Open
  ✓ Ticket received: [Ticket #1004] Emily Watson        | Hardware     | Monitor flickering intermittent
ly | Status: Open
  ✓ Ticket received: [Ticket #1005] David Brown         | Software     | Email client keeps crashing
| Status: Open

  === Help Desk System Status ===
  Total Pending Tickets: 5
  Total Resolved Tickets: 0
  Stack Capacity: 50
  Available Space: 45
  Stack Empty: False
  Stack Full: False

  === Pending Tickets (5/50) ===

  --- Stack Contents (Top to Bottom) ---
  1. [Ticket #1005] David Brown         | Software     | Email client keeps crashing    | Status: Open
  2. [Ticket #1004] Emily Watson        | Hardware     | Monitor flickering intermittently | Status: Ope
n
  3. [Ticket #1003] Mike Chen           | Software     | MS Office license activation issue | Status: Op
en
```

```
================================================================================
TEST 2: PEEKING AT NEXT TICKET (LIFO - Most Recently Added)
================================================================================

Checking which ticket will be resolved next (without removing it)...
  Next to resolve (LIFO): [Ticket #1005] David Brown          | Software     | Email client keeps crashin
g    | Status: Open

Stack still has 5 tickets

================================================================================
TEST 3: RESOLVING TICKETS (LIFO ORDER - Last In, First Out)
================================================================================

Resolving tickets in LIFO order...
  1. RESOLVED: [Ticket #1005] David Brown          | Software     | Email client keeps crashing    | Stat
us: Resolved
     Remaining tickets: 4
  2. RESOLVED: [Ticket #1004] Emily Watson         | Hardware     | Monitor flickering intermittently | S
tatus: Resolved
     Remaining tickets: 3
  3. RESOLVED: [Ticket #1003] Mike Chen            | Software     | MS Office license activation issue |
Status: Resolved
     Remaining tickets: 2
  4. RESOLVED: [Ticket #1002] Sarah Johnson        | Hardware     | Printer not responding on network | S
tatus: Resolved
     Remaining tickets: 1
  5. RESOLVED: [Ticket #1001] John Smith           | Network      | Cannot connect to VPN          | Stat
us: Resolved
     Remaining tickets: 0
```

Explaination:

The program uses a stack to manage help desk tickets.

A stack works in last in, first solved order.

When a new ticket is raised, it is added to the top.

When solving a ticket, the most recent one is handled first.

The program can also check if there are no tickets left or if the stack is full.

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
```

Prompt:

Implement a Hash Table to understand collision handling. The system should support insert, search, and delete operations. Use proper collision handling. Generate the full working code in a single file starting from the given starter code:

```
class HashTable:

pass
```

Code:

E:\3-2\AI Assisted Coding\HashTableCollisionHandling.py

```
"""
Hash Table Implementation with Collision Handling
Demonstrates insert, search, and delete operations
Uses chaining (linked list) to handle hash collisions
"""



# ============================================================================
# CLASS: HashTable - Hash Table with Chaining Collision Handling
# ============================================================================
class HashTable:
    """
    Hash Table implementation using chaining for collision handling
    Each bucket stores a list of (key, value) pairs that hash to the same index
    """

    def __init__(self, size=10):
        """
        Initialize a hash table with a given size

        Args:
            size (int): Number of buckets in the hash table (default: 10)
        """
        self.size = size
        # Initialize each bucket as an empty list for chaining
        self.table = [[] for _ in range(self.size)]
        self.num_entries = 0  # Track number of key-value pairs

    def _hash_function(self, key):
        """
        Hash function using modulo operation
        Converts key to hash index

        Args:
            key: The key to hash

        Returns:
            int: Index in the hash table (0 to size-1)
        """
        # Convert key to string and sum ASCII values, then mod by table size
        hash_value = sum(ord(char) for char in str(key)) % self.size
```

```python
def insert(self, key, value):
    """
    Insert a key-value pair into the hash table
    If key exists, update the value
    If collision occurs, add to the chain at that bucket

    Args:
        key: The key to insert
        value: The value associated with the key

    Returns:
        str: Message indicating success/update
    """
    # Get the hash index for this key
    index = self._hash_function(key)
    bucket = self.table[index]

    # Check if key already exists in this bucket (update case)
    for i, (existing_key, existing_value) in enumerate(bucket):
        if existing_key == key:
            bucket[i] = (key, value)  # Update existing value
            return f"Updated: Key '{key}' -> '{value}' at index {index}"

    # Key doesn't exist, add new entry (collision handling)
    bucket.append((key, value))
    self.num_entries += 1
    collision = "COLLISION!" if len(bucket) > 1 else "inserted"
    return f"Inserted: Key '{key}' -> '{value}' at index {index} [{collision}]"

def search(self, key):
    """
    Search for a key in the hash table and return its value

    Args:
        key: The key to search for

    Returns:
        tuple: (found, value) where found is bool and value is the data
    """
    # Get the hash index
```

```python
def display_table(self):
    """Display the entire hash table structure"""
    print("\n" + "=" * 70)
    print(f"Hash Table (Size: {self.size}, Entries: {self.num_entries}, Load Factor: {self.get_load_factor():.2f})")
    print("=" * 70)

    for index, bucket in enumerate(self.table):
        if bucket:  # Only display non-empty buckets
            collision_indicator = f"[CHAIN of {len(bucket)}]" if len(bucket) > 1 else ""
            print(f"  Index {index:2d}: {bucket} {collision_indicator}")
        else:
            print(f"  Index {index:2d}: (empty)")

def display_statistics(self):
    """Display hash table statistics"""
    total_buckets = self.size
    non_empty_buckets = sum(1 for bucket in self.table if bucket)
    empty_buckets = total_buckets - non_empty_buckets

    # Find longest chain (collision chain length)
    max_chain_length = max(len(bucket) for bucket in self.table) if self.table else 0

    print("\n" + "-" * 70)
    print("Hash Table Statistics:")
    print(f"  Total Buckets: {total_buckets}")
    print(f"  Total Entries: {self.num_entries}")
    print(f"  Non-Empty Buckets: {non_empty_buckets}")
    print(f"  Empty Buckets: {empty_buckets}")
    print(f"  Load Factor: {self.get_load_factor():.2f} ({self.num_entries}/{total_buckets})")
    print(f"  Max Chain Length: {max_chain_length}")
    print(f"  Average Chain Length: {self.num_entries / non_empty_buckets if non_empty_buckets > 0 else 0:.2f}")
    print("-" * 70)

def clear(self):
    """Clear all entries from the hash table"""
    self.table = [[] for _ in range(self.size)]
    self.num_entries = 0
```

```python
# ============================================================================
# CLASS: HashTableWithLinearProbing - Alternative implementation
# ============================================================================
class HashTableWithLinearProbing:
    """
    Hash Table using Linear Probing for collision handling
    When collision occurs, find next available empty slot
    """

    def __init__(self, size=10):
        """
        Initialize hash table with linear probing

        Args:
            size (int): Number of slots in the hash table
        """
        self.size = size
        self.keys = [None] * self.size
        self.values = [None] * self.size
        self.num_entries = 0

    def _hash_function(self, key):
        """Hash function for linear probing"""
        return sum(ord(char) for char in str(key)) % self.size

    def insert(self, key, value):
        """
        Insert with linear probing collision handling
        If slot is occupied, try next slot, next, etc.
        """
        index = self._hash_function(key)
        original_index = index
        probe_count = 0

        # Linear probing: if occupied, try next slot
        while self.keys[index] is not None and self.keys[index] != key:
            index = (index + 1) % self.size
            probe_count += 1

            # Prevent infinite loop (table is full)
            if probe_count >= self.size:
```

```python
# ============================================================================
# TEST FUNCTION: Comprehensive Hash Table Demonstration
# ============================================================================
def test_hash_table():
    """Comprehensive test of hash table operations"""

    print("=" * 70)
    print("HASH TABLE IMPLEMENTATION - COLLISION HANDLING DEMONSTRATION")
    print("=" * 70)

    # ========================================================================
    # TEST 1: Basic Insert and Display
    # ========================================================================
    print("\n" + "=" * 70)
    print("TEST 1: BASIC INSERT OPERATIONS")
    print("=" * 70)

    ht = HashTable(size=7)

    test_data = [
        ("Alice", "alice@email.com"),
        ("Bob", "bob@email.com"),
        ("Charlie", "charlie@email.com"),
        ("David", "david@email.com"),
        ("Eve", "eve@email.com"),
    ]

    print("\nInserting test data...")
    for key, value in test_data:
        result = ht.insert(key, value)
        print(f"  {result}")

    ht.display_table()
    ht.display_statistics()

    # ========================================================================
    # TEST 2: Collision Handling (Chaining)
    # ========================================================================
    print("\n" + "=" * 70)
    print("TEST 2: COLLISION HANDLING - Creating Intentional Collisions")
    print("=" * 70)
```

```python
# ======================================================================
# TEST 3: Search Operations
# ======================================================================
print("\n" + "=" * 70)
print("TEST 3: SEARCH OPERATIONS")
print("=" * 70)

print("\nSearching for keys...")
search_keys = ["Alice", "Frank", "Nonexistent", "Bob"]

for key in search_keys:
    found, value, index = ht2.search(key)
    if found:
        print(f"  ✓ Found: Key '{key}' -> '{value}' (at index {index})")
    else:
        print(f"  ✗ Not Found: Key '{key}'")

# ======================================================================
# TEST 4: Update Operations
# ======================================================================
print("\n" + "=" * 70)
print("TEST 4: UPDATE OPERATIONS")
print("=" * 70)

print("\nUpdating existing keys...")
update_data = [
    ("Alice", "newalice@email.com"),
    ("Bob", "newbob@email.com"),
]

for key, new_value in update_data:
    result = ht2.insert(key, new_value)
    print(f"  {result}")

print("\nVerifying updates...")
for key, _ in update_data:
    found, value, index = ht2.search(key)
    print(f"  Key '{key}': '{value}'")
```

```python
# ======================================================================
# TEST 5: Delete Operations
# ======================================================================
print("\n" + "=" * 70)
print("TEST 5: DELETE OPERATIONS")
print("=" * 70)

print("\nDeleting keys from hash table...")
delete_keys = ["Alice", "Emma", "Nonexistent"]

for key in delete_keys:
    found, index = ht2.delete(key)
    if found:
        print(f"  ✓ Deleted: Key '{key}' from index {index}")
    else:
        print(f"  ✗ Not Found: Could not delete '{key}'")

print("\nHash table after deletions:")
ht2.display_table()
ht2.display_statistics()

# ======================================================================
# TEST 6: Load Factor and Performance
# ======================================================================
print("\n" + "=" * 70)
print("TEST 6: LOAD FACTOR AND COLLISION ANALYSIS")
print("=" * 70)

print("\nLoad Factor indicates how full the hash table is:")
print("  • Low load factor (< 0.5): Few collisions, more memory")
print("  • High load factor (> 0.75): Many collisions, may need resizing")

ht3 = HashTable(size=10)

print("\nInserting items one by one and tracking load factor...")
items = [
    ("User1", "data1"),
    ("User2", "data2"),
    ("User3", "data3"),
    ("User4", "data4"),
```

```python
# TEST 7: Linear Probing (Alternative Collision Handling)
# ============================================================
print("\n" + "=" * 70)
print("TEST 7: LINEAR PROBING - ALTERNATIVE COLLISION HANDLING")
print("=" * 70)

print("\nLinear Probing: When collision occurs, find next empty slot")

lp_table = HashTableWithLinearProbing(size=10)

probe_data = [
    ("John", "john@email.com"),
    ("Jane", "jane@email.com"),
    ("Jack", "jack@email.com"),
    ("Jill", "jill@email.com"),
    ("James", "james@email.com"),
    ("Jessica", "jessica@email.com"),
]

print("\nInserting with linear probing...")
for key, value in probe_data:
    result = lp_table.insert(key, value)
    print(f"  {result}")

lp_table.display_table()

# Search in linear probing table
print("\nSearching in linear probing table...")
for key in ["John", "Jessica", "NotFound"]:
    found, value, index = lp_table.search(key)
    if found:
        print(f"  ✓ Found: '{key}' -> '{value}'")
    else:
        print(f"  X Not Found: '{key}'")

# ============================================================
# SUMMARY
# ============================================================
print("\n" + "=" * 70)
print("SUMMARY - COLLISION HANDLING TECHNIQUES")
```

OUTPUT TERMINAL PORTS POSTMAN CONSOLE DEBUG CONSOLE

```
    print("=" * 70)
    print("""
1. CHAINING (Used in Test 1-6):
   • Each bucket stores a list of key-value pairs
   • When collision occurs, append to the chain
   • Advantages: Simple, can handle high load factors
   • Disadvantages: Extra memory for pointers, slower search

2. LINEAR PROBING (Demonstrated in Test 7):
   • When collision occurs, find next empty slot
   • Wrap around table if necessary
   • Advantages: Better memory usage, cache friendly
   • Disadvantages: Clustering problem, slower deletion

3. HASH FUNCTION:
   • Converts keys into array indices
   • Quality of hash function crucial for performance
   • Good hash function minimizes collisions

4. LOAD FACTOR:
   • Ratio of entries to bucket count
   • Indicates table fullness and collision likelihood
   • Typical threshold: resize when load factor > 0.75

5. KEY OPERATIONS:
   • INSERT: Add or update key-value pair
   • SEARCH: Find value by key
   • DELETE: Remove key-value pair
   • All operations O(1) average case, O(n) worst case with collisions
    """)
    print("=" * 70 + "\n")


# ============================================================================
# MAIN EXECUTION
# ============================================================================
if __name__ == "__main__":
    test_hash_table()
```

Output:

```
E:\3-2\AI Assisted Coding>C:/Users/hp/AppData/Local/Programs/Python/Python311/python.exe "e:/3-2/AI Assisted Co
ling.py"
======================================================================
HASH TABLE IMPLEMENTATION - COLLISION HANDLING DEMONSTRATION
======================================================================

======================================================================
TEST 1: BASIC INSERT OPERATIONS
======================================================================

Inserting test data...
  Inserted: Key 'Alice' -> 'alice@email.com' at index 2 [inserted]
  Inserted: Key 'Bob' -> 'bob@email.com' at index 2 [COLLISION!]
  Inserted: Key 'Charlie' -> 'charlie@email.com' at index 3 [inserted]
  Inserted: Key 'David' -> 'david@email.com' at index 5 [inserted]
  Inserted: Key 'Eve' -> 'eve@email.com' at index 1 [inserted]

======================================================================
Hash Table (Size: 7, Entries: 5, Load Factor: 0.71)
======================================================================
  Index  0: (empty)
  Index  1: [('Eve', 'eve@email.com')]
  Index  2: [('Alice', 'alice@email.com'), ('Bob', 'bob@email.com')] [CHAIN of 2]
  Index  3: [('Charlie', 'charlie@email.com')]
  Index  4: (empty)
  Index  5: [('David', 'david@email.com')]
  Index  6: (empty)

----------------------------------------------------------------------
Hash Table Statistics:
  Total Buckets: 7
  Total Entries: 5
  Non-Empty Buckets: 4
  Empty Buckets: 3
  Load Factor: 0.71 (5/7)
  Max Chain Length: 2
  Average Chain Length: 1.25
----------------------------------------------------------------------

======================================================================
TEST 2: COLLISION HANDLING - Creating Intentional Collisions
======================================================================

Inserting keys that will cause collisions...
(Small hash table increases chance of collisions)
```

```
  Index  2: [('Diana', 'diana@email.com'), ('Grace', 'grace@email.com')] [CHAIN of 2]
  Index  3: [('Frank', 'frank@email.com')]
  Index  4: (empty)

----------------------------------------------------------------------
Hash Table Statistics:
  Total Buckets: 5
  Total Entries: 6
  Non-Empty Buckets: 4
  Empty Buckets: 1
  Load Factor: 1.20 (6/5)
  Max Chain Length: 2
  Average Chain Length: 1.50
----------------------------------------------------------------------

======================================================================
TEST 6: LOAD FACTOR AND COLLISION ANALYSIS
======================================================================

Load Factor indicates how full the hash table is:
  • Low load factor (< 0.5): Few collisions, more memory
  • High load factor (> 0.75): Many collisions, may need resizing

Inserting items one by one and tracking load factor...
  Inserted 'User1': Load Factor = 0.10
  Inserted 'User2': Load Factor = 0.20
  Inserted 'User3': Load Factor = 0.30
  Inserted 'User4': Load Factor = 0.40
  Inserted 'User5': Load Factor = 0.50
  Inserted 'User6': Load Factor = 0.60
  Inserted 'User7': Load Factor = 0.70
  Inserted 'User8': Load Factor = 0.80

----------------------------------------------------------------------
Hash Table Statistics:
  Total Buckets: 10
  Total Entries: 8
  Non-Empty Buckets: 8
  Empty Buckets: 2
  Load Factor: 0.80 (8/10)
  Max Chain Length: 1
  Average Chain Length: 1.00
----------------------------------------------------------------------
```

```
  Index  2: Key='Jane', Value='jane@email.com'
  Index  3: (empty)
  Index  4: (empty)
  Index  5: Key='Jill', Value='jill@email.com'
  Index  6: Key='James', Value='james@email.com'
  Index  7: Key='Jack', Value='jack@email.com'
  Index  8: Key='Jessica', Value='jessica@email.com'
  Index  9: Key='John', Value='john@email.com'

Searching in linear probing table...
  ✓ Found: 'John' -> 'john@email.com'
  ✓ Found: 'Jessica' -> 'jessica@email.com'
  ✗ Not Found: 'NotFound'

======================================================================
SUMMARY - COLLISION HANDLING TECHNIQUES
======================================================================

1. CHAINING (Used in Test 1-6):
   • Each bucket stores a list of key-value pairs
   • When collision occurs, append to the chain
   • Advantages: Simple, can handle high load factors
   • Disadvantages: Extra memory for pointers, slower search

2. LINEAR PROBING (Demonstrated in Test 7):
   • When collision occurs, find next empty slot
   • Wrap around table if necessary
   • Advantages: Better memory usage, cache friendly
   • Disadvantages: Clustering problem, slower deletion

3. HASH FUNCTION:
   • Converts keys into array indices
   • Quality of hash function crucial for performance
   • Good hash function minimizes collisions

4. LOAD FACTOR:
   • Ratio of entries to bucket count
   • Indicates table fullness and collision likelihood
   • Typical threshold: resize when load factor > 0.75

5. KEY OPERATIONS:
   • INSERT: Add or update key-value pair
   • SEARCH: Find value by key
   • DELETE: Remove key-value pair
   • All operations O(1) average case, O(n) worst case with collisions
```

Explaination:

A Hash Table stores data using a key and value.

• A hash function decides where to store the data.

• Sometimes two keys go to the same place. This is called a collision.

• To solve collisions, we use chaining, meaning we store multiple items in a list at the same index.

• The program should allow adding, finding, and removing data correctly.

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.

Prompt:

Design a Campus Resource Management System that includes Student Attendance Tracking as a core feature. The system should allow adding students, marking attendance (present/absent), viewing attendance records, and generating a simple attendance report for each student. Generate the full working code in a single file with clear structure.

Code:

```
"""
Campus Resource Management System with Student Attendance Tracking
Comprehensive system for managing student attendance records and generating reports
"""

from datetime import datetime, timedelta
from collections import defaultdict


# ============================================================================
# CLASS: Student - Represents a single student
# ============================================================================
class Student:
    """Stores information about a student and their attendance records"""

    # Class variable to auto-generate student IDs
    student_id_counter = 1000

    def __init__(self, name, major, email):
        """
        Initialize a student profile

        Args:
            name (str): Full name of the student
            major (str): Academic major/program
            email (str): Student email address
        """
        Student.student_id_counter += 1
        self.student_id = Student.student_id_counter
        self.name = name
        self.major = major
        self.email = email
        self.registration_date = datetime.now()
        self.attendance_records = []  # List of attendance records

    def __repr__(self):
        return f"Student({self.student_id}, {self.name})"

    def display(self):
```

```python
class Student:
        return f"Student({self.student_id}, {self.name})"

    def display(self):
        """Display student details"""
        return f"ID: {self.student_id} | Name: {self.name:25} | Major: {self.major:20} | Email: {self.email}"

    def get_attendance_stats(self):
        """Calculate attendance statistics for this student"""
        if not self.attendance_records:
            return {
                "total_classes": 0,
                "present_count": 0,
                "absent_count": 0,
                "attendance_rate": 0.0
            }

        total = len(self.attendance_records)
        present = sum(1 for record in self.attendance_records if record["status"] == "Present")
        absent = total - present
        rate = (present / total * 100) if total > 0 else 0

        return {
            "total_classes": total,
            "present_count": present,
            "absent_count": absent,
            "attendance_rate": rate
        }


# ============================================================================
# CLASS: AttendanceRecord - Represents a single attendance entry
# ============================================================================
class AttendanceRecord:
    """Stores a single attendance record for a student"""

    def __init__(self, course_name, date, status, notes=""):
        """
        Initialize an attendance record
```

```python
class AttendanceRecord:
    def __init__(self, course_name, date, status, notes=""):
        self.course_name = course_name
        self.date = date if isinstance(date, datetime) else datetime.fromisoformat(date)
        self.status = status.capitalize()  # "Present" or "Absent"
        self.notes = notes
        self.recorded_time = datetime.now()

    def __repr__(self):
        return f"AttendanceRecord({self.course_name}, {self.date.date()}, {self.status})"

    def display(self):
        """Display attendance record details"""
        return f"[{self.date.strftime('%Y-%m-%d')}] {self.course_name:25} | {self.status:8} | {self.notes}"


# ============================================================================
# CLASS: Course - Represents a course
# ============================================================================
class Course:
    """Stores information about a course"""

    course_id_counter = 2000

    def __init__(self, name, instructor, semester):
        """
        Initialize a course

        Args:
            name (str): Course name
            instructor (str): Instructor name
            semester (str): Semester (e.g., "Spring 2024")
        """
        Course.course_id_counter += 1
        self.course_id = Course.course_id_counter
        self.name = name
        self.instructor = instructor
        self.semester = semester
        self.enrolled_students = []
```

```
Campus: {self.campus_name}
Report Generated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}

Overall Statistics:
  Total Students:              {total_students}
  Total Attendance Records:    {total_records}
  Campus Average Attendance:   {avg_rate:.1f}%

Students with Low Attendance (< 75%):
"""

        if at_risk:
            report += "  " + "-" * 65 + "\n"
            for i, student in enumerate(at_risk, 1):
                stats = student.get_attendance_stats()
                report += f"  {i:2d}. {student.name:25} | Attendance: {stats['attendance_rate']:.1f}%\n"
        else:
            report += "  (No at-risk students)\n"

        report += "\n" + "=" * 70 + "\n"

        return report

    def display_all_students(self):
        """Display all students in the system"""
        print("\n" + "=" * 80)
        print(f"All Students ({len(self.students)} total)")
        print("=" * 80)

        if not self.students:
            print("  No students in the system.")
            return

        for student in sorted(self.students.values(), key=lambda x: x.student_id):
            print(f"  {student.display()}")
```

```python
# ============================================================================
# TEST FUNCTION: Comprehensive demonstration
# ============================================================================
def test_campus_management_system():
    """Comprehensive test of the Campus Resource Management System"""

    print("=" * 80)
    print("CAMPUS RESOURCE MANAGEMENT SYSTEM - STUDENT ATTENDANCE TRACKING")
    print("=" * 80)

    # Initialize the system
    campus = CampusResourceManagementSystem("Tech University")

    # ========================================================================
    # TEST 1: Add Students
    # ========================================================================
    print("\n" + "=" * 80)
    print("TEST 1: ADDING STUDENTS TO THE SYSTEM")
    print("=" * 80)

    print("\nAdding students...")
    students_data = [
        ("Alice Johnson", "Computer Science", "alice@techuni.edu"),
        ("Bob Smith", "Information Technology", "bob@techuni.edu"),
        ("Charlie Davis", "Computer Science", "charlie@techuni.edu"),
        ("Diana Martinez", "Software Engineering", "diana@techuni.edu"),
        ("Emma Wilson", "Information Technology", "emma@techuni.edu"),
    ]

    added_students = []
    for name, major, email in students_data:
        student = campus.add_student(name, major, email)
        added_students.append(student)
        print(f"  ✓ Added: {student.display()}")

    campus.display_all_students()
```

```
]

added_courses = []
for name, instructor, semester in courses_data:
    course = campus.create_course(name, instructor, semester)
    added_courses.append(course)
    print(f"  ✓ Created: {course.display()}")

campus.display_all_courses()

# ============================================================
# TEST 3: Enroll Students in Courses
# ============================================================
print("\n" + "=" * 80)
print("TEST 3: ENROLLING STUDENTS IN COURSES")
print("=" * 80)

print("\nEnrolling students...")
enrollment_data = [
    (added_students[0].student_id, added_courses[0].course_id),
    (added_students[0].student_id, added_courses[1].course_id),
    (added_students[1].student_id, added_courses[0].course_id),
    (added_students[1].student_id, added_courses[2].course_id),
    (added_students[2].student_id, added_courses[0].course_id),
    (added_students[3].student_id, added_courses[1].course_id),
    (added_students[4].student_id, added_courses[2].course_id),
]

for student_id, course_id in enrollment_data:
    student = campus.get_student(student_id)
    course = campus.get_course(course_id)
    if campus.enroll_student(student_id, course_id):
        print(f"  ✓ Enrolled {student.name} in {course.name}")

# Display updated courses
campus.display_all_courses()
```

```python
print("\nMarking attendance for students...")
base_date = datetime(2024, 3, 1)

attendance_records = [
    (added_students[0].student_id, "Data Structures", base_date, "Present"),
    (added_students[0].student_id, "Data Structures", base_date + timedelta(days=1), "Present"),
    (added_students[0].student_id, "Data Structures", base_date + timedelta(days=2), "Absent"),
    (added_students[1].student_id, "Data Structures", base_date, "Present"),
    (added_students[1].student_id, "Data Structures", base_date + timedelta(days=1), "Present"),
    (added_students[2].student_id, "Data Structures", base_date, "Present"),
    (added_students[2].student_id, "Data Structures", base_date + timedelta(days=1), "Absent"),
    (added_students[2].student_id, "Data Structures", base_date + timedelta(days=2), "Absent"),
]

for student_id, course, date, status in attendance_records:
    student = campus.get_student(student_id)
    if campus.mark_attendance(student_id, course, date, status):
        print(f"  ✓ {student.name:20} | {course:20} | {date.strftime('%Y-%m-%d')} | {status}")

# ============================================================
# TEST 5: Mark Bulk Attendance
# ============================================================
print("\n" + "=" * 80)
print("TEST 5: MARKING BULK ATTENDANCE FOR A COURSE")
print("=" * 80)

print("\nMarking attendance for Web Development class...")
bulk_attendance = {
    added_students[0].student_id: "Present",
    added_students[3].student_id: "Present",
}

count = campus.mark_bulk_attendance(added_courses[1].course_id, base_date + timedelta(days=3), bulk_attendance)
print(f"  ✓ Successfully marked attendance for {count} students")

# ============================================================
# TEST 6: View Attendance Records
# ============================================================
```

(function) student_id: Any

```
def test_campus_management_system():
        report = campus.generate_student_attendance_report(student.student_id)
        print(report)

    # ======================================================================
    # TEST 8: Generate Course Attendance Report
    # ======================================================================
    print("=" * 80)
    print("TEST 8: GENERATING COURSE ATTENDANCE REPORT")
    print("=" * 80)

    report = campus.generate_course_attendance_report(added_courses[0].course_id)
    print(report)

    # ======================================================================
    # TEST 9: Generate Campus Summary
    # ======================================================================
    print("=" * 80)
    print("TEST 9: GENERATING CAMPUS ATTENDANCE SUMMARY")
    print("=" * 80)

    summary = campus.generate_campus_attendance_summary()
    print(summary)

    # ======================================================================
    # TEST 10: Search Students
    # ======================================================================
    print("=" * 80)
    print("TEST 10: SEARCHING FOR STUDENTS")
    print("=" * 80)

    search_term = "alice"
    results = campus.search_student_by_name(search_term)
    print(f"\nSearch results for '{search_term}':")
    for student in results:
        print(f"  ✓ {student.display()}")

    # ======================================================================
    # SUMMARY
```

```
  ✓ Add/remove students
  ✓ Store student information (name, major, email)
  ✓ Search students by name
  ✓ Auto-generate student IDs

COURSE MANAGEMENT:
  ✓ Create courses
  ✓ Enroll students in courses
  ✓ Track enrolled students

ATTENDANCE TRACKING:
  ✓ Mark individual attendance (Present/Absent)
  ✓ Mark bulk attendance for entire classes
  ✓ Store attendance dates and notes
  ✓ Track attendance history

REPORTING:
  ✓ Student attendance reports with statistics
  ✓ Course attendance reports with date breakdown
  ✓ Campus-wide attendance summary
  ✓ Identify at-risk students (< 75% attendance)
  ✓ Attendance rate calculations

SYSTEM FEATURES:
  ✓ Timestamps for all records
  ✓ Flexible date handling
  ✓ Statistics and analytics
  ✓ Comprehensive error handling
    """)
    print("=" * 80 + "\n")


# ============================================================================
# MAIN EXECUTION
# ============================================================================
if __name__ == "__main__":
    test_campus_management_system()
```

Output:

```
================================================================================
CAMPUS RESOURCE MANAGEMENT SYSTEM - STUDENT ATTENDANCE TRACKING
================================================================================

================================================================================
TEST 1: ADDING STUDENTS TO THE SYSTEM
================================================================================

Adding students...
  ✓ Added: ID: 1001 | Name: Alice Johnson             | Major: Computer Science     | Email: alice@techuni.edu
  ✓ Added: ID: 1002 | Name: Bob Smith                 | Major: Information Technology | Email: bob@techuni.edu
  ✓ Added: ID: 1003 | Name: Charlie Davis             | Major: Computer Science     | Email: charlie@techuni.edu
  ✓ Added: ID: 1004 | Name: Diana Martinez            | Major: Software Engineering | Email: diana@techuni.edu
  ✓ Added: ID: 1005 | Name: Emma Wilson               | Major: Information Technology | Email: emma@techuni.edu

================================================================================
All Students (5 total)
================================================================================
  ID: 1001 | Name: Alice Johnson             | Major: Computer Science     | Email: alice@techuni.edu
  ID: 1002 | Name: Bob Smith                 | Major: Information Technology | Email: bob@techuni.edu
  ID: 1003 | Name: Charlie Davis             | Major: Computer Science     | Email: charlie@techuni.edu
  ID: 1004 | Name: Diana Martinez            | Major: Software Engineering | Email: diana@techuni.edu
  ID: 1005 | Name: Emma Wilson               | Major: Information Technology | Email: emma@techuni.edu

================================================================================
TEST 2: CREATING COURSES
================================================================================

Creating courses...
  ✓ Created: ID: 2001 | Data Structures           | Instructor: Dr. Johnson          | Semester: Spring 2024 | Enrolled: 0
  ✓ Created: ID: 2002 | Web Development           | Instructor: Prof. Smith          | Semester: Spring 2024 | Enrolled: 0
  ✓ Created: ID: 2003 | Database Systems          | Instructor: Dr. Lee              | Semester: Spring 2024 | Enrolled: 0

================================================================================
All Courses (3 total)
================================================================================
  ID: 2001 | Data Structures           | Instructor: Dr. Johnson          | Semester: Spring 2024 | Enrolled: 0
  ID: 2002 | Web Development           | Instructor: Prof. Smith          | Semester: Spring 2024 | Enrolled: 0
  ID: 2003 | Database Systems          | Instructor: Dr. Lee              | Semester: Spring 2024 | Enrolled: 0

================================================================================
TEST 3: ENROLLING STUDENTS IN COURSES
================================================================================
```

```
  ✓ Enrolled Bob Smith in Database Systems
  ✓ Enrolled Charlie Davis in Data Structures
  ✓ Enrolled Diana Martinez in Web Development
  ✓ Enrolled Emma Wilson in Database Systems

================================================================================
All Courses (3 total)
================================================================================
  ID: 2001 | Data Structures           | Instructor: Dr. Johnson          | Semester: Spring 2024 | Enrolled: 3
  ID: 2002 | Web Development           | Instructor: Prof. Smith          | Semester: Spring 2024 | Enrolled: 2
  ID: 2003 | Database Systems          | Instructor: Dr. Lee              | Semester: Spring 2024 | Enrolled: 2

================================================================================
TEST 4: MARKING INDIVIDUAL ATTENDANCE
================================================================================

Marking attendance for students...
  ✓ Alice Johnson        | Data Structures      | 2024-03-01 | Present
  ✓ Alice Johnson        | Data Structures      | 2024-03-02 | Present
  ✓ Alice Johnson        | Data Structures      | 2024-03-03 | Absent
  ✓ Bob Smith            | Data Structures      | 2024-03-01 | Present
  ✓ Bob Smith            | Data Structures      | 2024-03-02 | Present
  ✓ Charlie Davis        | Data Structures      | 2024-03-01 | Present
  ✓ Charlie Davis        | Data Structures      | 2024-03-02 | Absent
  ✓ Charlie Davis        | Data Structures      | 2024-03-03 | Absent

================================================================================
TEST 5: MARKING BULK ATTENDANCE FOR A COURSE
================================================================================

Marking attendance for Web Development class...
  ✓ Successfully marked attendance for 2 students

================================================================================
TEST 6: VIEWING ATTENDANCE RECORDS
================================================================================

Attendance records for students:

  Alice Johnson - 4 records:
    [2024-03-01] Data Structures          | Present  |
    [2024-03-02] Data Structures          | Present  |
    [2024-03-03] Data Structures          | Absent   |
    [2024-03-04] Web Development          | Present  |
```

Explaination:

Campus Resource Management System – Student Attendance Tracking:

The system maintains a record of students and their attendance status.

- When a new student is added, their details are stored in the system.
- When attendance is marked, the student's status is updated as present or absent for that session.
- The system allows viewing individual attendance records to track performance.
- A report can be generated to show total classes attended and overall attendance summary.