

AI ASSISTANT CODING

ASSIGNMENT - 11.3

Name: Aman Sarkar

Ht.no: 2303A51273

Batch: 05

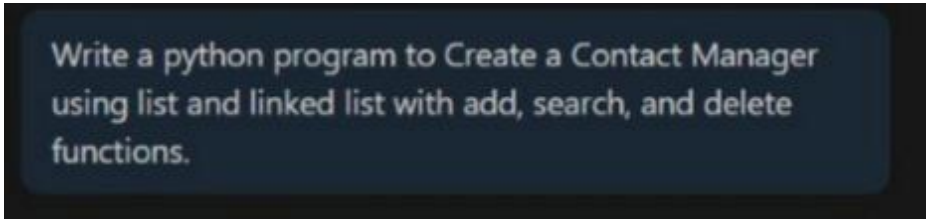
Task 1: Smart Contact Manager (Arrays & Linked Lists)

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.

Prompt:

Write a python program to Create a Contact Manager using list and linked list with add, search, and delete functions.



Write a python program to Create a Contact Manager using list and linked list with add, search, and delete functions.

Code:

```
1 // contact-manager.js
2
3 // Create a single contact
4 function addContact(name, phone, email) {
5   const contact = {
6     name: name,
7     phone: phone,
8     email: email
9   };
10
11   // Add the contact to the array
12   contacts.push(contact);
13
14   // Log the contact to the console
15   console.log('Contact added successfully');
16 }
17
18 // Delete a contact
19 function deleteContact(id) {
20   // Find the contact by ID
21   const contact = contacts.find(c => c.id === id);
22
23   if (!contact) {
24     console.log('Contact not found');
25     return;
26   }
27
28   // Remove the contact from the array
29   contacts.splice(contacts.indexOf(contact), 1);
30
31   // Log the contact to the console
32   console.log('Contact deleted successfully');
33 }
34
35 // Display all contacts
36 function displayContacts() {
37   // Log the contacts to the console
38   console.log(contacts);
39 }
40
41 // Main function
42 function main() {
43   // Add a contact
44   addContact('John Doe', '1234567890', 'john.doe@example.com');
45
46   // Delete a contact
47   deleteContact(1);
48
49   // Display all contacts
50   displayContacts();
51 }
52
53 // Run the main function
54 main();
```

```
1 // contact-manager.js
2
3 // Create a single contact
4 function addContact(name, phone, email) {
5   const contact = {
6     name: name,
7     phone: phone,
8     email: email
9   };
10
11   // Add the contact to the array
12   contacts.push(contact);
13
14   // Log the contact to the console
15   console.log('Contact added successfully');
16 }
17
18 // Delete a contact
19 function deleteContact(id) {
20   // Find the contact by ID
21   const contact = contacts.find(c => c.id === id);
22
23   if (!contact) {
24     console.log('Contact not found');
25     return;
26   }
27
28   // Remove the contact from the array
29   contacts.splice(contacts.indexOf(contact), 1);
30
31   // Log the contact to the console
32   console.log('Contact deleted successfully');
33 }
34
35 // Display all contacts
36 function displayContacts() {
37   // Log the contacts to the console
38   console.log(contacts);
39 }
40
41 // Main function
42 function main() {
43   // Add a contact
44   addContact('John Doe', '1234567890', 'john.doe@example.com');
45
46   // Delete a contact
47   deleteContact(1);
48
49   // Display all contacts
50   displayContacts();
51 }
52
53 // Run the main function
54 main();
```




- 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.

Prompt:

Write a Python program for a library book request system. First, make a normal queue where requests are handled in the order they come. Then, make another version where faculty requests are given first priority over student requests. Include functions to add a request and remove a request.

Write a Python program for a library book request system.
First, make a normal queue where requests are handled in the order they come.
Then, make another version where faculty requests are given first priority over student requests.
Include functions to add a request and remove a request.

Code:

```
File Edit Selection ... Q AI Assistant Coding
buggy.py library_book_request.py contact_manager.py

class NormalQueue:
    def __init__(self, request_id, requester_name, book_title):
        self.request_id = request_id
        self.requester_name = requester_name
        self.book_title = book_title

    def __str__(self):
        return f"({self.request_id}, Requester: {self.requester_name}, Book: {self.book_title})"

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

    def add_request(self, request):
        """Add a request to the queue"""
        self.queue.append(request)
        print("Request added: (request)")

    def remove_request(self):
        """Remove and return the first request"""
        if not self.queue:
            print("Queue is empty")
            return None
        request = self.queue.popleft()
        print("Request processed: (request)")
        return request

    def display_queue(self):
        """Display all requests in queue"""
        if not self.queue:
            print("Queue is empty")
            return
        print("\n--- Normal Queue ---")
        for i, req in enumerate(self.queue, 1):
            print(f"({i}, {req})")

    def size(self):
        return len(self.queue)

# PRIORITY QUEUE IMPLEMENTATION
class PriorityBookRequest:
    def __init__(self, request_id, requester_name, book_title, user_type):
        self.request_id = request_id
        self.requester_name = requester_name
        self.book_title = book_title
        self.user_type = user_type # "Faculty" or "Student"

    def __str__(self):
        return f"({self.request_id}, Requester: {self.requester_name}, ({self.user_type}), Book: {self.book_title})"

    def get_priority(self):
        """Return priority (lower number = higher priority)"""
        if self.user_type.lower() == "Faculty":
            return 1
        else:
            return 2
```

```
File Edit Selection ... Q AI Assistant Coding
buggy.py library_book_request.py contact_manager.py

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

    def add_request(self, request):
        """Add a request with priority sorting"""
        self.queue.append(request)
        self.queue.sort(key=lambda x: x.get_priority())
        print("Request added: (request)")

    def remove_request(self):
        """Remove and return the highest priority request"""
        if not self.queue:
            print("Queue is empty")
            return None
        request = self.queue.pop(0)
        print("Request processed: (request)")
        return request

    def display_queue(self):
        """Display all requests in priority order"""
        if not self.queue:
            print("Queue is empty")
            return
        print("\n--- Priority Queue ---")
        for i, req in enumerate(self.queue, 1):
            print(f"({i}, {req})")

    def size(self):
        return len(self.queue)

# DEMO PROGRAM
if __name__ == "__main__":
    print("\n--- DEMO ---")
    normal_q = NormalQueue()

    # Add requests
    normal_q.add_request(BookRequest(1, "Alice", "Python Programming"))
    normal_q.add_request(BookRequest(2, "Bob", "Data Science"))
    normal_q.add_request(BookRequest(3, "Charlie", "Web Development"))
    normal_q.add_request(BookRequest(4, "Diana", "Machine Learning"))

    normal_q.display_queue()
    print("Queue size: (normal_q.size())")

    # Process requests
    print("\n--- Processing Requests (Normal Queue) ---")
    normal_q.remove_request()
    normal_q.remove_request()
    normal_q.remove_request()
    normal_q.remove_request()

    normal_q.display_queue()
    print("Queue size: (normal_q.size())")

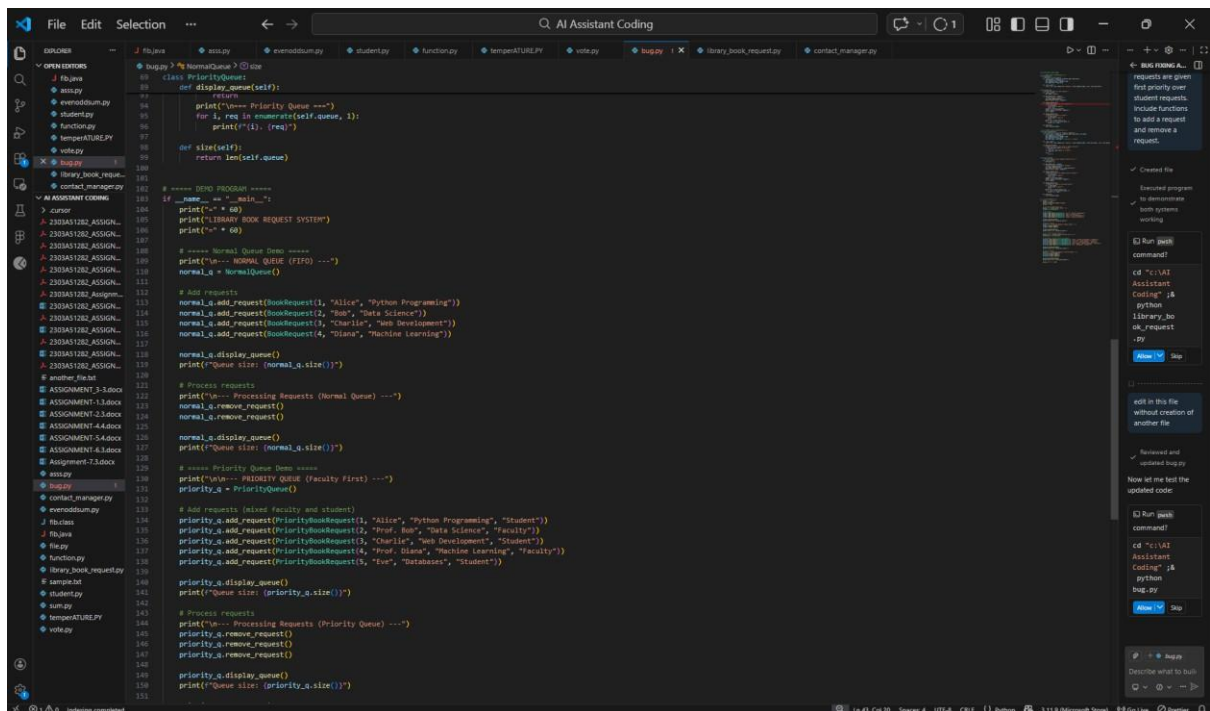
    # Priority Queue Demo
    priority_q = PriorityQueue()

    priority_q.add_request(PriorityBookRequest(1, "Alice", "Python Programming", "Faculty"))
    priority_q.add_request(PriorityBookRequest(2, "Bob", "Data Science", "Student"))
    priority_q.add_request(PriorityBookRequest(3, "Charlie", "Web Development", "Faculty"))
    priority_q.add_request(PriorityBookRequest(4, "Diana", "Machine Learning", "Student"))

    priority_q.display_queue()
    print("Queue size: (priority_q.size())")

    # Process requests
    print("\n--- Processing Requests (Priority Queue) ---")
    priority_q.remove_request()
    priority_q.remove_request()
    priority_q.remove_request()
    priority_q.remove_request()

    priority_q.display_queue()
    print("Queue size: (priority_q.size())")
```

Output:

```

PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bug.py"

=== Priority Queue ===
1. ID: 3, Requester: Charlie (Student), Book: Web Development
2. ID: 5, Requester: Eve (Student), Book: Databases
Queue size: 2

=====
PS C:\AI Assistant coding>

```

Explanation:

- 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.

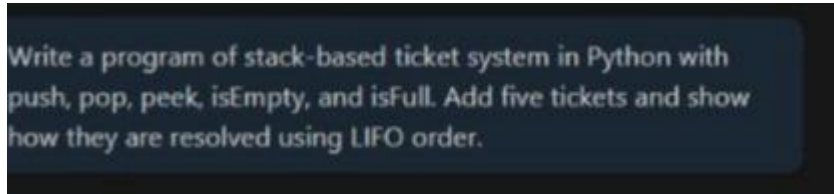
Task 3: Emergency Help Desk (Stack Implementation)

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.

Prompt:

Write a program of stack-based ticket system in Python with `push`, `pop`, `peek`, `isEmpty`, and `isFull`. Add five tickets and show how they are resolved using LIFO order.



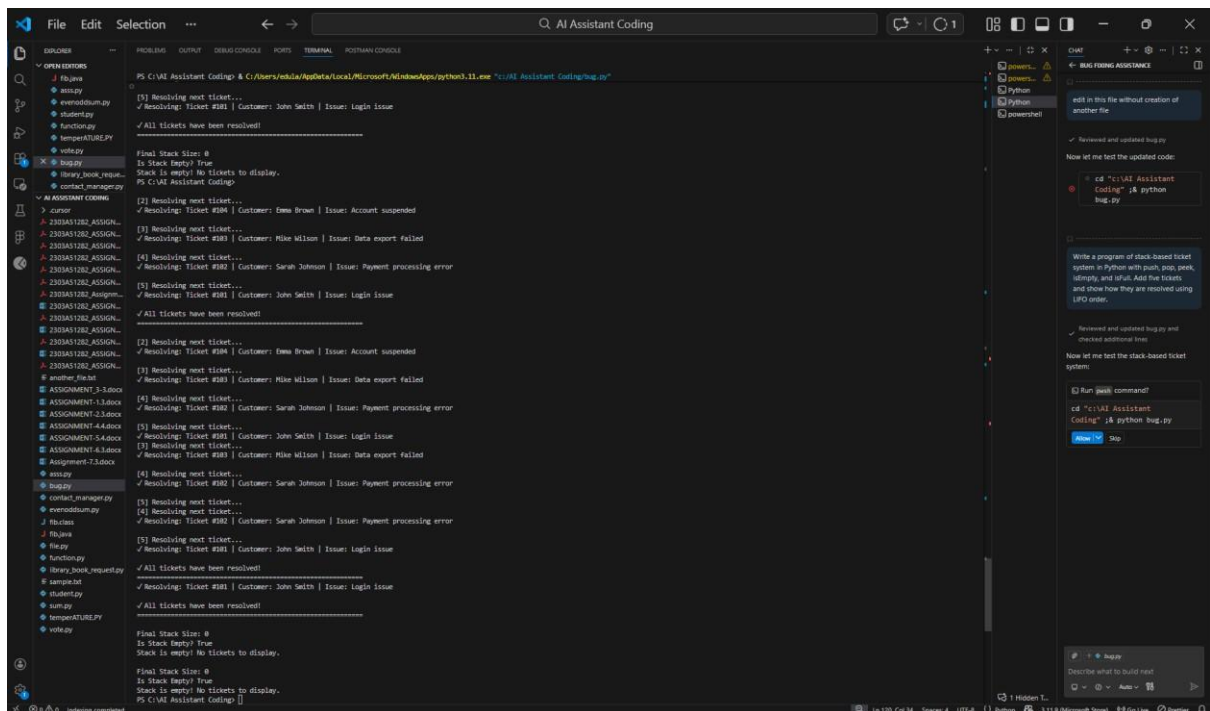
Write a program of stack-based ticket system in Python with `push`, `pop`, `peek`, `isEmpty`, and `isFull`. Add five tickets and show how they are resolved using LIFO order.

Code:


```
1 # --- STACK-BASED TICKET SYSTEM ---
2 class TicketStack:
3     """Represents a support ticket system"""
4     def __init__(self, ticket_id, customer_name, issue):
5         self.ticket_id = ticket_id
6         self.customer_name = customer_name
7         self.issue = issue
8
9     def __str__(self):
10         return f"Ticket #{self.ticket_id} | Customer: {self.customer_name} | Issue: {self.issue}"
11
12 class TicketSystem:
13     """Stack-based ticket management system (LIFO - Last In, First Out)"""
14     def __init__(self, max_size=10):
15         self.stack = []
16         self.max_size = max_size
17
18     def push(self, ticket):
19         """Add a ticket to the stack (top of stack)"""
20         if self.is_full():
21             print(f"Error: Stack is full! Cannot add ticket #{ticket.ticket_id}")
22             return False
23         self.stack.append(ticket)
24         print(f"Ticket added: {ticket}")
25         return True
26
27     def pop(self):
28         """Remove and return the ticket from the top of the stack"""
29         if self.is_empty():
30             print(f"Error: Stack is empty! No tickets to resolve.")
31             return None
32         ticket = self.stack.pop()
33         print(f"Resolving: {ticket}")
34         return ticket
35
36     def peek(self):
37         """View the top ticket without removing it"""
38         if self.is_empty():
39             print(f"Error: Stack is empty!")
40             return None
41         return self.stack[-1]
42
43     def is_empty(self):
44         """Check if the stack is empty"""
45         return len(self.stack) == 0
46
47     def is_full(self):
48         """Check if the stack is full"""
49         return len(self.stack) >= self.max_size
50
51     def size(self):
52         """Return the number of tickets in the stack"""
53         return len(self.stack)
54
55     def display_stack(self):
56         """Display all tickets in the stack (top to bottom)"""
57         for ticket in self.stack:
58             print(ticket)
```

```
1 # --- STACK-BASED TICKET SYSTEM ---
2 class TicketStack:
3     """Represents a support ticket system"""
4     def __init__(self, ticket_id, customer_name, issue):
5         self.ticket_id = ticket_id
6         self.customer_name = customer_name
7         self.issue = issue
8
9     def __str__(self):
10         return f"Ticket #{self.ticket_id} | Customer: {self.customer_name} | Issue: {self.issue}"
11
12 class TicketSystem:
13     """Stack-based ticket management system (LIFO - Last In, First Out)"""
14     def __init__(self, max_size=10):
15         self.stack = []
16         self.max_size = max_size
17
18     def push(self, ticket):
19         """Add a ticket to the stack (top of stack)"""
20         if self.is_full():
21             print(f"Error: Stack is full! Cannot add ticket #{ticket.ticket_id}")
22             return False
23         self.stack.append(ticket)
24         print(f"Ticket added: {ticket}")
25         return True
26
27     def pop(self):
28         """Remove and return the ticket from the top of the stack"""
29         if self.is_empty():
30             print(f"Error: Stack is empty! No tickets to resolve.")
31             return None
32         ticket = self.stack.pop()
33         print(f"Resolving: {ticket}")
34         return ticket
35
36     def peek(self):
37         """View the top ticket without removing it"""
38         if self.is_empty():
39             print(f"Error: Stack is empty!")
40             return None
41         return self.stack[-1]
42
43     def is_empty(self):
44         """Check if the stack is empty"""
45         return len(self.stack) == 0
46
47     def is_full(self):
48         """Check if the stack is full"""
49         return len(self.stack) >= self.max_size
50
51     def size(self):
52         """Return the number of tickets in the stack"""
53         return len(self.stack)
54
55     def display_stack(self):
56         """Display all tickets in the stack (top to bottom)"""
57         for ticket in self.stack:
58             print(ticket)
59
60     def resolve_all(self):
61         """Resolve all tickets in LIFO order"""
62         print(f"Resolving ALL TICKETS (LIFO - Last In, First Out)")
63         count = 1
64         while not self.is_empty():
65             print(f"\n[Count: {count}] Resolving next ticket...")
66             self.pop()
67             count += 1
68         print(f"\nAll tickets have been resolved!")
69         print(f"Count: {count}")
70
71 # --- DEMO PROGRAM ---
72 if __name__ == "__main__":
73     print(f"Stack-based Ticket System")
74     print(f"---")
75     # Create ticket stack with max size of 10
76     ticket_system = TicketSystem(max_size=10)
77
78     # Add five tickets
79     print(f"--- ADDING TICKETS TO THE STACK ---")
80     ticket_system.push(ticket(101, "John Smith", "Login issue"))
81     ticket_system.push(ticket(102, "Sarah Johnson", "Payment processing error"))
82     ticket_system.push(ticket(103, "Mike Wilson", "Data export failed"))
83     ticket_system.push(ticket(104, "Tom Brown", "Account suspension"))
84     ticket_system.push(ticket(105, "David Lee", "Password reset not working"))
85
86     # Display current stack
87     ticket_system.display_stack()
88
89     # Show stack information
90     print(f"Stack Size: {ticket_system.size()}")
91     print(f"Is Stack Empty? {ticket_system.is_empty()}")
92     print(f"Is Stack Full? {ticket_system.is_full()}")
93
94     # Peek at the top ticket
95     print(f"--- PEAK AT TOP TICKET ---")
96     top_ticket = ticket_system.peek()
97     if top_ticket:
98         print(f"Top ticket (without removing): {top_ticket}")
99
100     # Resolve all tickets in LIFO order
101     ticket_system.resolve_all()
102
103     # Display final stack state
104     print(f"Final Stack Size: {ticket_system.size()}")
105     print(f"Is Stack Empty? {ticket_system.is_empty()}")
106     ticket_system.display_stack()
```

Output:



Explanation:

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.

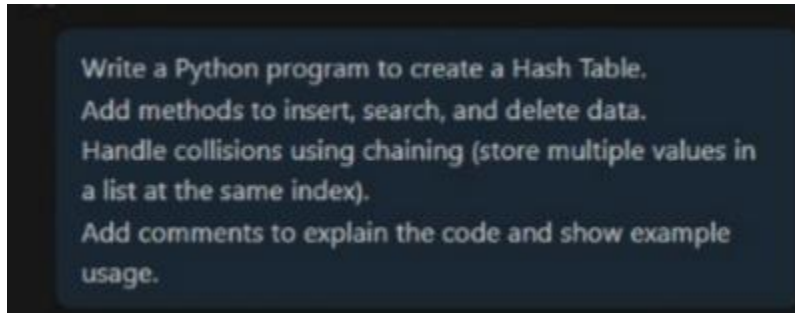
Task 4: Hash Table

Objective

To implement a Hash Table and understand collision handling.

Prompt:

Write a Python program to create a Hash Table.
Add methods to insert, search, and delete data.
Handle collisions using chaining (store multiple values in a list at the same index).
Add comments to explain the code and show example usage.



Code:

```
File Edit Selection ... AI Assistant Coding
buggy.py
# Create Hash Table with CHAINING collision handling.
# Key-Value pair class for storing data.
class Node:
    """Represents a key-value pair in the hash table"""
    def __init__(self, key, value):
        self.key = key
        self.value = value

    def __str__(self):
        return f"({self.key}, {self.value})"

class HashTable:
    """Hash Table implementation using chaining to handle collisions.
    Chaining stores multiple key-value pairs that hash to the same index in a linked list.
    """
    def __init__(self, size=10):
        """Initialize the hash table
        Args:
            size (int): Number of buckets in the hash table
        """
        self.size = size
        # Create an array of empty lists (chains) for collision handling
        self.table = [[] for _ in range(size)]
        self.total_items = 0

    def _hash_function(self, key):
        """Hash function to determine the index for a given key
        Uses simple modulo operation: hash(key) % table_size
        Args:
            key: The key to hash
        Returns:
            int: Index in the hash table
        """
        # Convert key to string and sum ASCII values for different key types
        key_str = str(key)
        hash_value = sum(ord(char) for char in key_str)
        return hash_value % self.size

    def insert(self, key, value):
        """Insert a key-value pair into the hash table
        If key already exists, update its value
        If collision occurs, add to the chain (list) at that index
        Args:
            key: The key to insert
            value: The value associated with the key
        Returns:
            bool: True if insertion successful, False otherwise
        """
        # Find the hash index
        index = self._hash_function(key)
        chain = self.table[index]

        # Check if key already exists in the chain and update if found
        for i, pair in enumerate(chain):
            if pair.key == key:
                chain[i] = (key, value)
                return True

        # Add to the chain if not found
        chain.append((key, value))
        self.total_items += 1
        return True

    def search(self, key):
        """Search for a key in the hash table
        Args:
            key: The key to search for
        Returns:
            list: List of values associated with the key
        """
        index = self._hash_function(key)
        chain = self.table[index]

        # Return all values for the given key
        return [pair.value for pair in chain if pair.key == key]

    def delete(self, key):
        """Delete a key-value pair from the hash table
        Args:
            key: The key to delete
        Returns:
            bool: True if deletion successful, False otherwise
        """
        index = self._hash_function(key)
        chain = self.table[index]

        # Find the index of the key in the chain
        for i, pair in enumerate(chain):
            if pair.key == key:
                # Remove the pair from the chain
                chain.pop(i)
                self.total_items -= 1
                return True

        return False

# Example usage
if __name__ == "__main__":
    # Create a hash table
    ht = HashTable(size=10)

    # Insert key-value pairs
    ht.insert("apple", 1)
    ht.insert("banana", 2)
    ht.insert("apple", 3)  # Collision: 'apple' already exists, update value
    ht.insert("orange", 4)
    ht.insert("banana", 5)  # Collision: 'banana' already exists, update value

    # Search for keys
    print("Search for 'apple':", ht.search("apple"))  # Output: [3]
    print("Search for 'banana':", ht.search("banana"))  # Output: [5]
    print("Search for 'orange':", ht.search("orange"))  # Output: [4]

    # Delete a key
    ht.delete("apple")
    print("Search for 'apple' after deletion:", ht.search("apple"))  # Output: []

    # Print total items
    print("Total items in hash table:", ht.total_items)  # Output: 4
```



```
def insert(self, key, value):
    """Insert a key-value pair into the hash table. If found, update the value.
    If not found, add a new key-value pair to the chain.
    """
    # Check if key already exists in the chain and update if found
    for i, pair in enumerate(self.chain):
        if pair[0] == key:
            self.chain[i][1] = value
            print(f"Updated: {key} = {value} (at index {i})")
            return True

    # If key not found, add new key-value pair to the chain
    self.chain.append((key, value))
    print(f"Inserted: {key} = {value} (at index {len(self.chain)-1})")
    return True

def search(self, key):
    """Search for a key in the hash table.
    Returns: The value associated with the key, or None if not found.
    """
    # Search for the key in the chain
    for i, pair in enumerate(self.chain):
        if pair[0] == key:
            return pair[1]

    # Key not found
    print(f"Key '{key}' not found in the hash table")
    return None

def delete(self, key):
    """Delete a key-value pair from the hash table.
    Returns: True if deletion successful, False if key not found.
    """
    # Search for the key in the chain
    for i, pair in enumerate(self.chain):
        if pair[0] == key:
            del self.chain[i]
            print(f"Deleted: {key} = {pair[1]} (from index {i})")
            return True

    # Key not found
    print(f"Key '{key}' not found in the hash table")
    return False
```

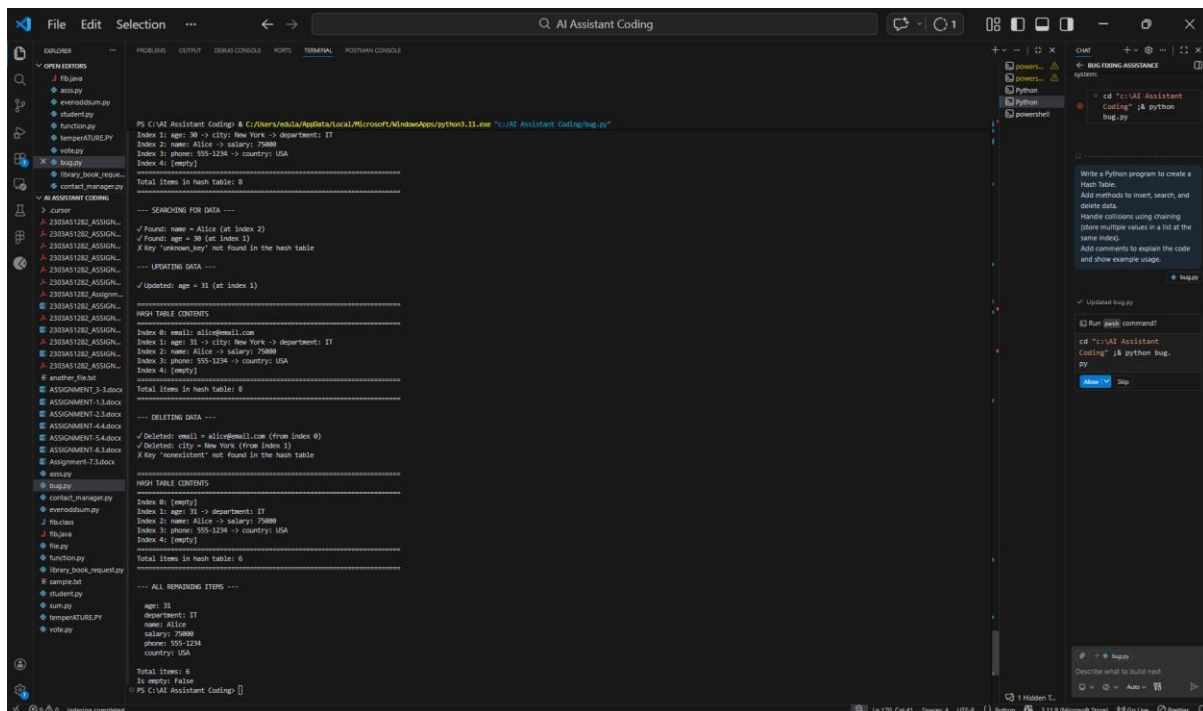
```
def display(self):
    """Display the hash table.
    """
    print("Hash Table Contents:")
    for i, pair in enumerate(self.chain):
        print(f"Index {i}: {pair[0]} = {pair[1]}")

def insert_multiple(self, items):
    """Insert multiple key-value pairs into the hash table.
    """
    for key, value in items:
        self.insert(key, value)

def delete_multiple(self, keys):
    """Delete multiple key-value pairs from the hash table.
    """
    for key in keys:
        self.delete(key)

def display_all_items(self):
    """Display all key-value pairs in the hash table.
    """
    all_items = []
    for pair in self.chain:
        all_items.append(pair)
    return all_items
```

Output:



Explanation:

- 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

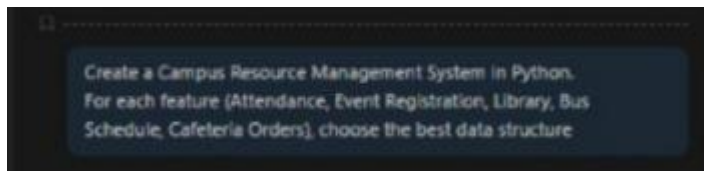
Design a Campus Resource Management System with the following features:

- Student Attendance Tracking

- Event Registration System
- Library Book Borrowing
- Bus Scheduling System
- Cafeteria Order Queue

Prompt:

Create a Campus Resource Management System in Python. For each feature (Attendance, Event Registration, Library, Bus Schedule, Cafeteria Orders), choose the best data structure



Code:

The image shows a code editor with a dark theme. The file explorer on the left shows a project structure with files like `attendance.py`, `event_registration.py`, `library.py`, `bus.py`, `cafeteria.py`, and `campus_resource_management.py`. The main editor shows the code for `campus_resource_management.py`. The code defines a `CampusResourceManagementSystem` class with methods for attendance, event registration, library, bus, and cafeteria. The code uses various data structures like `set`, `deque`, `list`, `dict`, and `heapq` to manage the data. The code is commented with the names of the features and the data structures chosen for each.

```

1  # Campus Resource Management System
2  """
3  Features and chosen data structures:
4  - Attendance: set (O(1) add/check) to track present student IDs
5  - Event Registration: deque (FIFO queue) for fair registration processing
6  - Library: dict (hash table) mapping ISBNs to book records for fast lookup
7  - Bus Schedule: dict of route -> sorted list of departure times (list kept sorted)
8  - Cafeteria Orders: heapq (priority queue) to prioritize faculty over students while preserving arrival order
9  """
10 Run this file to see a small demo of each feature.
11 """
12
13 from collections import deque
14 import heapq
15 import time
16 from bisect import insort
17 from datetime import datetime, timedelta
18
19 # ----- Attendance (set) -----
20 class Attendance:
21     """Track attendance using a set for O(1) add/remove/check."""
22     def __init__(self):
23         self.present = set()
24
25     def mark_present(self, student_id):
26         self.present.add(student_id)
27         print(f"Marked present: {student_id}")
28
29     def mark_absent(self, student_id):
30         self.present.discard(student_id)
31         print(f"Marked absent: {student_id}")
32
33     def is_present(self, student_id):
34         return student_id in self.present
35
36     def present_count(self):
37         return len(self.present)
38
39     def list_present(self):
40         return sorted(self.present)
41
42 # ----- Event Registration (FIFO queue) -----
43 class EventRegistration:
44     """Register attendees in arrival order using deque."""
45     def __init__(self):
46         self.queue = deque()
47
48     def add_registration(self, attendee_id, name):
49         self.queue.append((attendee_id, name))
50         print(f"Registered: {attendee_id} - {name}")
51
52     def process_registration(self):
53         if not self.queue:
54             print("No registrations to process.")
55             return None
56         attendee = self.queue.popleft()
57         print(f"Processing registration: {attendee[0]} - {attendee[1]}")
58         return attendee
59
60     def pending_count(self):
61         return len(self.queue)

```



```
class EventRegistration:
    def __init__(self):
        self.pending_count = 0
        self.event_list = []

    def add_event(self, title, author, copies, borrowers):
        self.event_list.append({
            'title': title,
            'author': author,
            'copies': copies,
            'borrowers': borrowers
        })

    def search(self, isbn):
        return self.catalog.get(isbn)

    def borrow_book(self, isbn, user_id):
        book = self.catalog.get(isbn)
        if not book:
            print("Book not found.")
            return False
        if book['copies'] <= 0:
            print("No copies available.")
            return False
        book['copies'] -= 1
        book['borrowers'].append(user_id)
        print(f"User {user_id} borrowed book: {book['title']}")
        return True

    def return_book(self, isbn, user_id):
        book = self.catalog.get(isbn)
        if not book:
            print("Book not found.")
            return False
        book['borrowers'].remove(user_id)
        book['copies'] += 1
        print(f"User {user_id} returned book: {book['title']}")
        return True

    def list_available(self):
        return [(isbn, info['title'], info['copies']) for isbn, info in self.catalog.items()]

class Library:
    def __init__(self):
        self.catalog = {}

    def add_book(self, isbn, title, author, copies):
        self.catalog[isbn] = {
            'title': title,
            'author': author,
            'copies': copies,
            'borrowers': []
        }

    def search(self, isbn):
        return self.catalog.get(isbn)

    def borrow_book(self, isbn, user_id):
        book = self.catalog.get(isbn)
        if not book:
            print("Book not found.")
            return False
        if book['copies'] <= 0:
            print("No copies available.")
            return False
        book['copies'] -= 1
        book['borrowers'].append(user_id)
        print(f"User {user_id} borrowed book: {book['title']}")
        return True

    def return_book(self, isbn, user_id):
        book = self.catalog.get(isbn)
        if not book:
            print("Book not found.")
            return False
        book['borrowers'].remove(user_id)
        book['copies'] += 1
        print(f"User {user_id} returned book: {book['title']}")
        return True

    def list_available(self):
        return [(isbn, info['title'], info['copies']) for isbn, info in self.catalog.items()]

class BusSchedule:
    def __init__(self):
        self.routes = {}

    def add_route(self, route_id, route_name, stops):
        self.routes[route_id] = {
            'route_name': route_name,
            'stops': stops
        }

    def search(self, route_id):
        return self.routes.get(route_id)

    def list_routes(self):
        return self.routes.values()
```

```
class EventRegistration:
    def __init__(self):
        self.pending_count = 0
        self.event_list = []

    def add_event(self, title, author, copies, borrowers):
        self.event_list.append({
            'title': title,
            'author': author,
            'copies': copies,
            'borrowers': borrowers
        })

    def search(self, isbn):
        return self.catalog.get(isbn)

    def borrow_book(self, isbn, user_id):
        book = self.catalog.get(isbn)
        if not book:
            print("Book not found.")
            return False
        if book['copies'] <= 0:
            print("No copies available.")
            return False
        book['copies'] -= 1
        book['borrowers'].append(user_id)
        print(f"User {user_id} borrowed book: {book['title']}")
        return True

    def return_book(self, isbn, user_id):
        book = self.catalog.get(isbn)
        if not book:
            print("Book not found.")
            return False
        book['borrowers'].remove(user_id)
        book['copies'] += 1
        print(f"User {user_id} returned book: {book['title']}")
        return True

    def list_available(self):
        return [(isbn, info['title'], info['copies']) for isbn, info in self.catalog.items()]

class Library:
    def __init__(self):
        self.catalog = {}

    def add_book(self, isbn, title, author, copies):
        self.catalog[isbn] = {
            'title': title,
            'author': author,
            'copies': copies,
            'borrowers': []
        }

    def search(self, isbn):
        return self.catalog.get(isbn)

    def borrow_book(self, isbn, user_id):
        book = self.catalog.get(isbn)
        if not book:
            print("Book not found.")
            return False
        if book['copies'] <= 0:
            print("No copies available.")
            return False
        book['copies'] -= 1
        book['borrowers'].append(user_id)
        print(f"User {user_id} borrowed book: {book['title']}")
        return True

    def return_book(self, isbn, user_id):
        book = self.catalog.get(isbn)
        if not book:
            print("Book not found.")
            return False
        book['borrowers'].remove(user_id)
        book['copies'] += 1
        print(f"User {user_id} returned book: {book['title']}")
        return True

    def list_available(self):
        return [(isbn, info['title'], info['copies']) for isbn, info in self.catalog.items()]

class BusSchedule:
    def __init__(self):
        self.routes = {}

    def add_route(self, route_id, route_name, stops):
        self.routes[route_id] = {
            'route_name': route_name,
            'stops': stops
        }

    def search(self, route_id):
        return self.routes.get(route_id)

    def list_routes(self):
        return self.routes.values()
```

Output:


```
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL POSTMAN CONSOLE

PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bug.py"
Is empty: False
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bug.py"
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/campus_resource_management.py"

=====
Campus Resource Management Demo
=====

Marked present: 5001
Marked present: 5002
Marked present: 5010
Present list: ['5001', '5002', '5010']
Is 5002 present? True
Marked absent: 5002
Present count: 2
Registered: A001 - Alice
Registered: A002 - Bob
Registered: A003 - Charlie
Pending registrations: [('A001', 'Alice'), ('A002', 'Bob'), ('A003', 'Charlie')]
Processed registration: A001 - Alice
Pending count: 2
Added book: Clean Code (ISBN: 978-0135166307).
Added book: Fluent Python (ISBN: 978-1491958296).
5001 borrowed Clean Code
5003 borrowed Clean Code
No copies available.
Available books: [('978-0135166307', 'Clean Code', 0), ('978-1491958296', 'Fluent Python', 1)]
5001 returned Clean Code
Available books after return: [('978-0135166307', 'Clean Code', 1), ('978-1491958296', 'Fluent Python', 1)]
Added bus time for Route A: 2026-02-18 10:42:24.367227
Added bus time for Route A: 2026-02-18 10:57:24.367227
Added bus time for Route B: 2026-02-18 10:39:24.367227
Next Route A bus: 2026-02-18 10:42:24.367227
Route A schedule: [datetime.datetime(2026, 2, 18, 10, 42, 24, 367227), datetime.datetime(2026, 2, 18, 10, 57, 24, 367227)]
Order added: 0001 (Student)
Order added: 0002 (Faculty)
Order added: 0003 (Student)
Order added: 0004 (Faculty)
Pending cafeteria orders: 4
Serving order: 0002 (Faculty)
Serving order: 0004 (Faculty)
Pending orders after serving: 2

Demo complete.
PS C:\AI Assistant Coding>
```

Explanation:

Library Book Borrowing using a queue:

- The queue stores student names who request a book.
- When a student requests a book, we use `enqueue()` to add them to the queue.
- When a book becomes available, we use `dequeue()` to give it to the first student in line.
- This ensures fairness because the first requester gets the book first.

