AI ASSISTANT CODING

ASSIGNMENT 11.3

Name: K Sneha

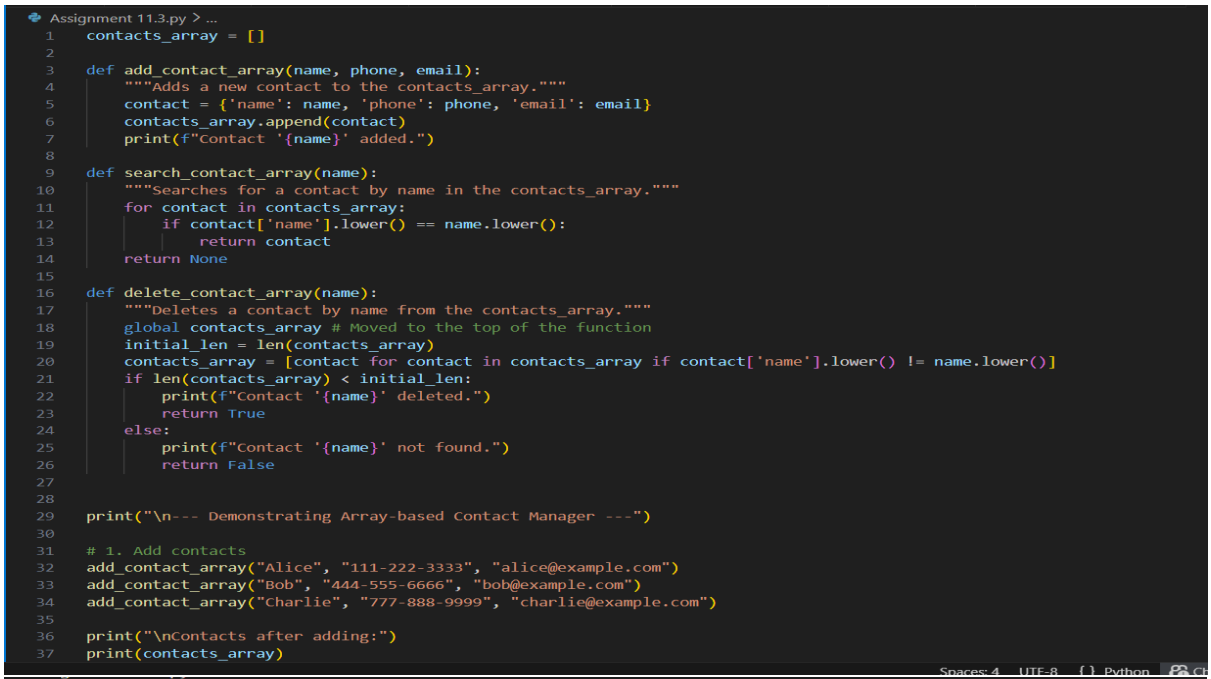
Batch:20

HT NO:2303A51332

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

Implement a contact manager using Python lists (arrays) with functions for adding, searching, and deleting contacts, then implement another contact manager using a custom linked list data structure with Node and LinkedList classes for the same operations. Finally, compare the performance (time complexity for insertion and deletion) of the array-based and linked list-based implementations

#code:



```
Assignment 11.3.py > ...
contacts_array = []

def add_contact_array(name, phone, email):
    """Adds a new contact to the contacts_array."""
    contact = {'name': name, 'phone': phone, 'email': email}
    contacts_array.append(contact)
    print(f"Contact '{name}' added.")

def search_contact_array(name):
    """Searches for a contact by name in the contacts_array."""
    for contact in contacts_array:
        if contact['name'].lower() == name.lower():
            return contact
    return None

def delete_contact_array(name):
    """Deletes a contact by name from the contacts_array."""
    global contacts_array # Moved to the top of the function
    initial_len = len(contacts_array)
    contacts_array = [contact for contact in contacts_array if contact['name'].lower() != name.lower()]
    if len(contacts_array) < initial_len:
        print(f"Contact '{name}' deleted.")
        return True
    else:
        print(f"Contact '{name}' not found.")
        return False


print("\n--- Demonstrating Array-based Contact Manager ---")

# 1. Add contacts
add_contact_array("Alice", "111-222-3333", "alice@example.com")
add_contact_array("Bob", "444-555-6666", "bob@example.com")
add_contact_array("Charlie", "777-888-9999", "charlie@example.com")

print("\nContacts after adding:")
print(contacts_array)
```

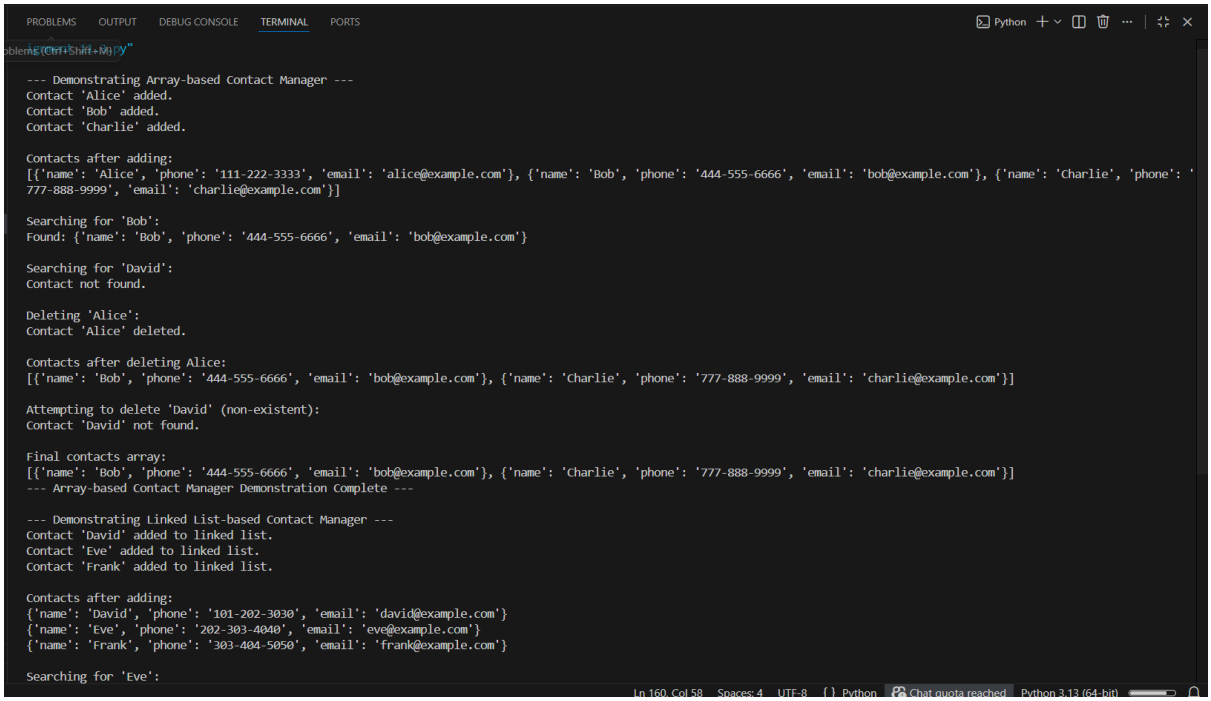
```
# 2. Search for a contact
print("\nSearching for 'Bob':")
found_contact = search_contact_array("Bob")
if found_contact:
    print(f"Found: {found_contact}")
else:
    print("Contact not found.")

print("\nSearching for 'David':")
found_contact = search_contact_array("David")
if found_contact:
    print(f"Found: {found_contact}")
else:
    print("Contact not found.")

# 3. Delete a contact
print("\nDeleting 'Alice':")
delete_contact_array("Alice")

print("\nContacts after deleting Alice:")
print(contacts_array)

print("\nAttempting to delete 'David' (non-existent):")
delete_contact_array("David")

print("\nFinal contacts array:")
print(contacts_array)

print("--- Array-based Contact Manager Demonstration Complete ---")


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

```python
class LinkedList:

    def add_contact_ll(self, name, phone, email):
        contact = {'name': name, 'phone': phone, 'email': email}
        new_node = Node(contact)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
        print(f"Contact '{name}' added to linked list.")

    def search_contact_ll(self, name):
        current = self.head
        while current:
            if current.contact_data['name'].lower() == name.lower():
                return current.contact_data
            current = current.next
        return None

    def delete_contact_ll(self, name):
        current = self.head
        prev = None

        # If head node itself holds the key to be deleted
        if current and current.contact_data['name'].lower() == name.lower():
            self.head = current.next
            print(f"Contact '{name}' deleted from linked list.")
            return True

        # Search for the key to be deleted, keep track of the previous node
        # as we need to change prev.next
        while current and current.contact_data['name'].lower() != name.lower():
            prev = current
            current = current.next
```

```python
class LinkedList:
    def delete_contact_ll(self, name):

            # If key was not present in linked list
            if current is None:
                print(f"Contact '{name}' not found in linked list.")
                return False

            # Unlink the node from linked list
            prev.next = current.next
            print(f"Contact '{name}' deleted from linked list.")
            return True

    def display_contacts_ll(self):
        contacts = []
        current = self.head
        if not current:
            print("Linked list is empty.")
            return
        while current:
            contacts.append(current.contact_data)
            current = current.next
        for contact in contacts:
            print(contact)


print("\n--- Demonstrating Linked List-based Contact Manager ---")

# Create an instance of the LinkedList
ll_contacts = LinkedList()

# 1. Add contacts
ll_contacts.add_contact_ll("David", "101-202-3030", "david@example.com")
ll_contacts.add_contact_ll("Eve", "202-303-4040", "eve@example.com")
ll_contacts.add_contact_ll("Frank", "303-404-5050", "frank@example.com")
```

```python
# 1. Add contacts
ll_contacts.add_contact_ll("David", "101-202-3030", "david@example.com")
ll_contacts.add_contact_ll("Eve", "202-303-4040", "eve@example.com")
ll_contacts.add_contact_ll("Frank", "303-404-5050", "frank@example.com")

print("\nContacts after adding:")
ll_contacts.display_contacts_ll()

# 2. Search for a contact
print("\nSearching for 'Eve':")
found_ll_contact = ll_contacts.search_contact_ll("Eve")
if found_ll_contact:
    print(f"Found: {found_ll_contact}")
else:
    print("Contact not found.")

print("\nSearching for 'Grace':")
found_ll_contact = ll_contacts.search_contact_ll("Grace")
if found_ll_contact:
    print(f"Found: {found_ll_contact}")
else:
    print("Contact not found.")

# 3. Delete a contact
print("\nDeleting 'David':")
ll_contacts.delete_contact_ll("David")

print("\nContacts after deleting David:")
ll_contacts.display_contacts_ll()

print("\nAttempting to delete 'Grace' (non-existent):")
ll_contacts.delete_contact_ll("Grace")

print("\nFinal linked list contacts:")
ll_contacts.display_contacts_ll()

print("--- Linked List-based Contact Manager Demonstration Complete ---")
```

OUTPUT:



Comparison: Array vs Linked List

Operation	Array (List)	Linked List
Insertion (End)	O(1) average	O(n)
Insertion (Beginning)	O(n)	O(1)
Search	O(n)	O(n)
Deletion	O(n) (shifting required)	O(n) (no shifting)
Memory	Fixed / contiguous	Dynamic / non-contiguous

Observation:

The Smart Contact Manager was implemented using both array and linked list data structures. In the array approach, insertion at the end is efficient, but inserting or deleting in the middle requires shifting elements, increasing time complexity. The linked list allows easy insertion and deletion through pointer manipulation without shifting. However, searching requires traversal in both methods, resulting in O(n) time complexity. Arrays use contiguous memory and are simpler to implement, while linked lists use dynamic memory and extra space for pointers. Arrays are suitable for small datasets, whereas linked lists are better for frequent modifications and flexibility

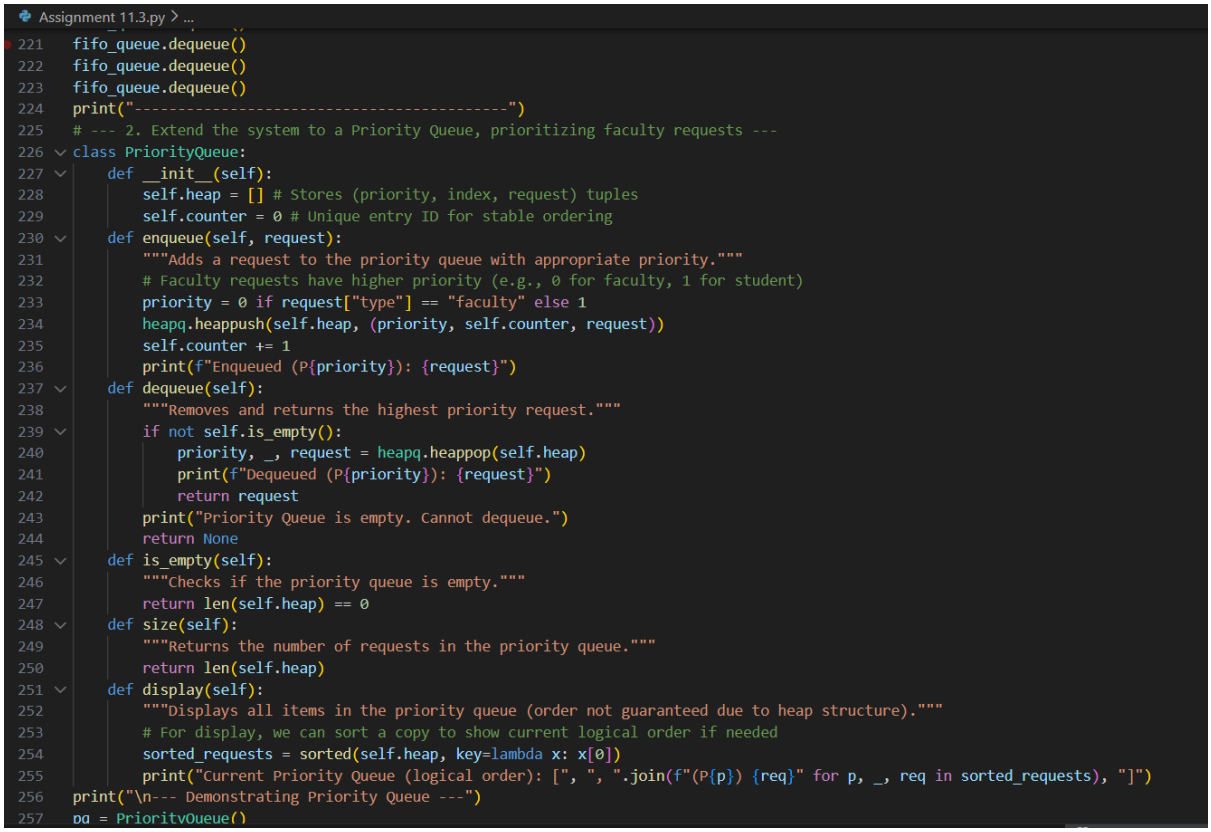
Task 2: Library Book Search System (Queues & Priority Queues)

Prompt: Create a Python program for a Library Book Search System. First, implement a normal Queue (FIFO) to manage book borrow requests using enqueue() and dequeue() methods. Then extend the system to implement a Priority Queue where faculty requests are given higher priority

over student requests. The program should correctly process faculty requests before student requests, even if faculty requests are added later. Test the system with a mix of student and faculty inputs and display the processing order clearly

#code

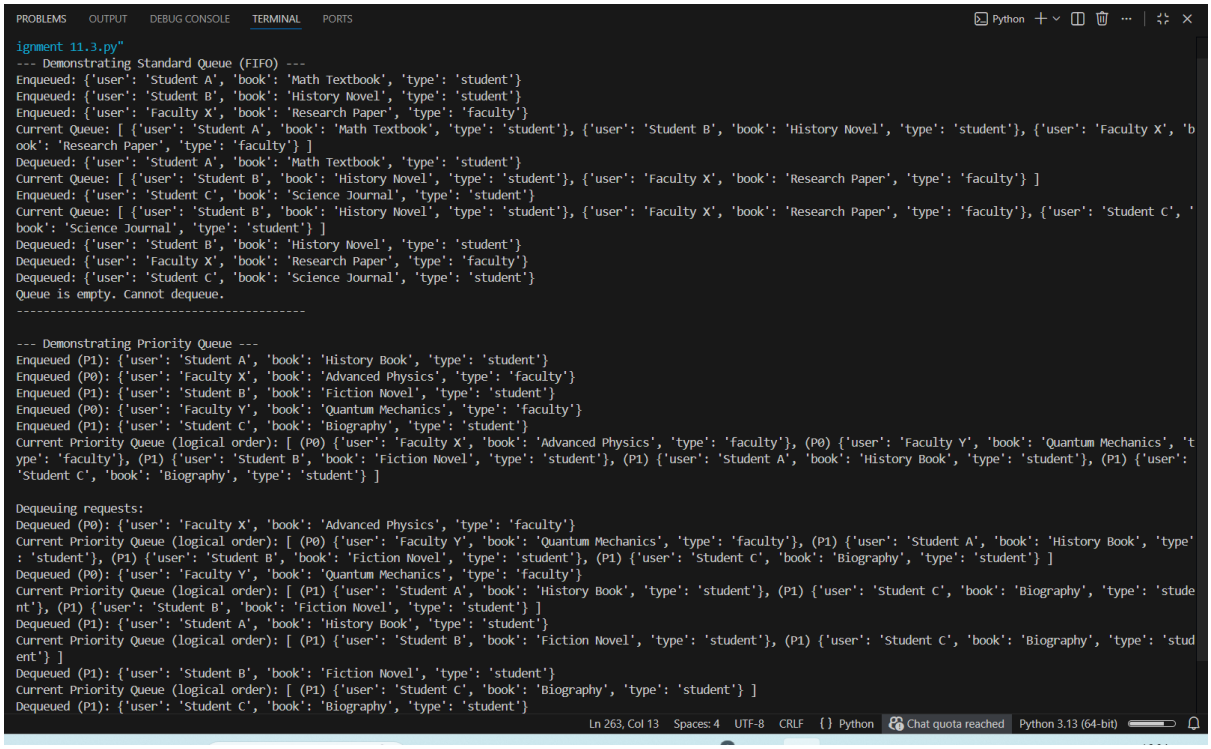
```
Assignment 11.3.py > ...
import collections
import heapq
# --- 1. Implement a Queue (FIFO) to manage book requests ---
class Queue:
    def __init__(self):
        self.items = collections.deque()
    def enqueue(self, request):
        """Adds a request to the end of the queue."""
        self.items.append(request)
        print(f"Enqueued: {request}")
    def dequeue(self):
        """Removes and returns the request from the front of the queue."""
        if not self.is_empty():
            request = self.items.popleft()
            print(f"Dequeued: {request}")
            return request
        print("Queue is empty. Cannot dequeue.")
        return None
    def is_empty(self):
        """Checks if the queue is empty."""
        return len(self.items) == 0
    def size(self):
        """Returns the number of requests in the queue."""
        return len(self.items)
    def display(self):
        """Displays all items in the queue."""
        print("Current Queue: [", ", ".join(str(item) for item in self.items), "]")
print("--- Demonstrating Standard Queue (FIFO) ---")
fifo_queue = Queue()
fifo_queue.enqueue({"user": "Student A", "book": "Math Textbook", "type": "student"})
fifo_queue.enqueue({"user": "Student B", "book": "History Novel", "type": "student"})
fifo_queue.enqueue({"user": "Faculty X", "book": "Research Paper", "type": "faculty"})
fifo_queue.display()
fifo_queue.dequeue()
fifo_queue.display()
fifo_queue.enqueue({"user": "Student C", "book": "Science Journal", "type": "student"})
fifo_queue.display()
```



```
Assignment 11.3.py > ...
fifo_queue.dequeue()
fifo_queue.dequeue()
fifo_queue.dequeue()
print("----------------------------------------")
# --- 2. Extend the system to a Priority Queue, prioritizing faculty requests ---
class PriorityQueue:
    def __init__(self):
        self.heap = [] # Stores (priority, index, request) tuples
        self.counter = 0 # Unique entry ID for stable ordering
    def enqueue(self, request):
        """Adds a request to the priority queue with appropriate priority."""
        # Faculty requests have higher priority (e.g., 0 for faculty, 1 for student)
        priority = 0 if request["type"] == "faculty" else 1
        heapq.heappush(self.heap, (priority, self.counter, request))
        self.counter += 1
        print(f"Enqueued (P{priority}): {request}")
    def dequeue(self):
        """Removes and returns the highest priority request."""
        if not self.is_empty():
            priority, _, request = heapq.heappop(self.heap)
            print(f"Dequeued (P{priority}): {request}")
            return request
        print("Priority Queue is empty. Cannot dequeue.")
        return None
    def is_empty(self):
        """Checks if the priority queue is empty."""
        return len(self.heap) == 0
    def size(self):
        """Returns the number of requests in the priority queue."""
        return len(self.heap)
    def display(self):
        """Displays all items in the priority queue (order not guaranteed due to heap structure)."""
        # For display, we can sort a copy to show current logical order if needed
        sorted_requests = sorted(self.heap, key=lambda x: x[0])
        print("Current Priority Queue (logical order): [", ", ".join(f"(P{p}) {req}" for p, _, req in sorted_requests), "]")
print("\n--- Demonstrating Priority Queue ---")
pq = PriorityQueue()
```

```python
print("\n--- Demonstrating Priority Queue ---")
pq = PriorityQueue()
pq.enqueue({"user": "Student A", "book": "History Book", "type": "student"})
pq.enqueue({"user": "Faculty X", "book": "Advanced Physics", "type": "faculty"})
pq.enqueue({"user": "Student B", "book": "Fiction Novel", "type": "student"})
pq.enqueue({"user": "Faculty Y", "book": "Quantum Mechanics", "type": "faculty"})
pq.enqueue({"user": "Student C", "book": "Biography", "type": "student"})
pq.display()
print("\nDequeuing requests:")
pq.dequeue()
pq.display()
pq.dequeue()
pq.display()
pq.dequeue()
pq.display()
pq.dequeue()
pq.display()
pq.dequeue()
pq.display()
pq.dequeue()
print("-------------------------------------")
```

OUTPUT:



Observation:

The Library Book Search System was implemented using both a normal queue and a priority queue. In the simple queue, requests were processed in FIFO order, meaning the first request submitted was handled first. However, this method did not differentiate between students and faculty. In the priority queue implementation, faculty requests were given higher priority over student requests. Testing with mixed inputs showed that faculty requests were processed first, even if they were added later. The enqueue() and dequeue() methods functioned correctly in both systems. Overall, the priority queue approach proved more suitable for managing library requests efficiently.

Task 3: Emergency Help Desk (Stack Implementation)

Prompt: To measure the memory bandwidth of a system by allocating a large array (100 million elements), initializing it with values, and calculating the time taken to read/write the data. The goal is

to evaluate how efficiently the CPU transfers data between memory and processor using OpenMP for parallel execution.

#Code:

```python
class Stack:
    def __init__(self, capacity=None):
        self.items = []
        self.capacity = capacity

    def push(self, ticket):
        """Adds a ticket to the top of the stack."""
        if self.capacity is not None and len(self.items) >= self.capacity:
            print(f"Stack is full. Cannot push ticket: {ticket}")
        else:
            self.items.append(ticket)
            print(f"Pushed: {ticket}")

    def pop(self):
        """Removes and returns the ticket from the top of the stack (LIFO)."""
        if not self.is_empty():
            ticket = self.items.pop()
            print(f"Popped: {ticket}")
            return ticket
        print("Stack is empty. Cannot pop.")
        return None

    def peek(self):
        """Returns the ticket at the top of the stack without removing it."""
        if not self.is_empty():
            return self.items[-1]
        print("Stack is empty. No ticket to peek.")
        return None

    def is_empty(self):
        """Checks if the stack is empty."""
        return len(self.items) == 0

    def size(self):
        """Returns the number of tickets in the stack."""
        return len(self.items)
```

```python
Assignment 11.3.py > ...
class Stack:
    def is_full(self):
        """Checks if the stack is full (if a capacity is set)."""
        return self.capacity is not None and len(self.items) >= self.capacity

    def display(self):
        """Displays all items in the stack from top to bottom."""
        if self.is_empty():
            print("Current Stack: [Empty]")
        else:
            # Display from top to bottom (last item in list is top of stack)
            print("Current Stack (Top to Bottom): [", ", ".join(str(item) for item in reversed(self.items)), "]")

    # Initialize the Help Desk Stack
help_desk_stack = Stack()

print("--- Raising Tickets ---")
help_desk_stack.push({"id": 1, "user": "Alice", "issue": "Login issue"})
help_desk_stack.push({"id": 2, "user": "Bob", "issue": "Software installation"})
help_desk_stack.push({"id": 3, "user": "Charlie", "issue": "Network connectivity"})
help_desk_stack.push({"id": 4, "user": "Diana", "issue": "Printer not working"})
help_desk_stack.push({"id": 5, "user": "Eve", "issue": "Email configuration"})

print(f"Stack size: {help_desk_stack.size()}")
help_desk_stack.display()

print("\n--- Resolving Tickets (LIFO) ---")
print(f"Current ticket at the top: {help_desk_stack.peek()}")
help_desk_stack.pop()
help_desk_stack.display()

print(f"Current ticket at the top: {help_desk_stack.peek()}")
help_desk_stack.pop()
help_desk_stack.display()

help_desk_stack.push({"id": 6, "user": "Frank", "issue": "VPN connection"}) # New ticket comes in
```

```python
Assignment 11.3.py > ...
    help_desk_stack.display()

    print("\n--- Resolving Tickets (LIFO) ---")
    print(f"Current ticket at the top: {help_desk_stack.peek()}")
    help_desk_stack.pop()
    help_desk_stack.display()

    print(f"Current ticket at the top: {help_desk_stack.peek()}")
    help_desk_stack.pop()
    help_desk_stack.display()

    help_desk_stack.push({"id": 6, "user": "Frank", "issue": "VPN connection"}) # New ticket comes in
    help_desk_stack.display()

    print(f"Current ticket at the top: {help_desk_stack.peek()}")
    help_desk_stack.pop()
    help_desk_stack.display()

    print(f"Current ticket at the top: {help_desk_stack.peek()}")
    help_desk_stack.pop()
    help_desk_stack.display()

    print(f"Is the stack empty? {help_desk_stack.is_empty()}")
    print(f"Current ticket at the top: {help_desk_stack.peek()}")
    help_desk_stack.pop()
    help_desk_stack.display()

    print(f"Is the stack empty? {help_desk_stack.is_empty()}")
    help_desk_stack.pop() # Try to pop from an empty stack

    print("\n--- Demonstrating with a capacity-limited stack ---")
    limited_stack = Stack(capacity=2)
    limited_stack.push({"id": 7, "user": "Grace", "issue": "Software bug"})
    limited_stack.push({"id": 8, "user": "Heidi", "issue": "Hardware fault"})
    limited_stack.push({"id": 9, "user": "Ivan", "issue": "OS update"}) # This should fail
    limited_stack.display()
    print(f"Is the limited stack full? {limited_stack.is_full()}")
```

OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                                    Python

ignment 11.3.py"
--- Raising Tickets ---
Pushed: {'id': 1, 'user': 'Alice', 'issue': 'Login issue'}
Pushed: {'id': 2, 'user': 'Bob', 'issue': 'Software installation'}
Pushed: {'id': 3, 'user': 'Charlie', 'issue': 'Network connectivity'}
Pushed: {'id': 4, 'user': 'Diana', 'issue': 'Printer not working'}
Pushed: {'id': 5, 'user': 'Eve', 'issue': 'Email configuration'}
Stack size: 5
Current Stack (Top to Bottom): [ {'id': 5, 'user': 'Eve', 'issue': 'Email configuration'}, {'id': 4, 'user': 'Diana', 'issue': 'Printer not working'}, {'id': 3, 'user': 'Charl
ie', 'issue': 'Network connectivity'}, {'id': 2, 'user': 'Bob', 'issue': 'Software installation'}, {'id': 1, 'user': 'Alice', 'issue': 'Login issue'} ]

--- Resolving Tickets (LIFO) ---
Current ticket at the top: {'id': 5, 'user': 'Eve', 'issue': 'Email configuration'}
Popped: {'id': 5, 'user': 'Eve', 'issue': 'Email configuration'}
Current Stack (Top to Bottom): [ {'id': 4, 'user': 'Diana', 'issue': 'Printer not working'}, {'id': 3, 'user': 'Charlie', 'issue': 'Network connectivity'}, {'id': 2, 'user': '
Bob', 'issue': 'Software installation'}, {'id': 1, 'user': 'Alice', 'issue': 'Login issue'} ]
Current ticket at the top: {'id': 4, 'user': 'Diana', 'issue': 'Printer not working'}
Popped: {'id': 4, 'user': 'Diana', 'issue': 'Printer not working'}
Current Stack (Top to Bottom): [ {'id': 3, 'user': 'Charlie', 'issue': 'Network connectivity'}, {'id': 2, 'user': 'Bob', 'issue': 'Software installation'}, {'id': 1, 'user': '
Alice', 'issue': 'Login issue'} ]
Pushed: {'id': 6, 'user': 'Frank', 'issue': 'VPN connection'}
Current Stack (Top to Bottom): [ {'id': 6, 'user': 'Frank', 'issue': 'VPN connection'}, {'id': 3, 'user': 'Charlie', 'issue': 'Network connectivity'}, {'id': 2, 'user': 'Bob',
 'issue': 'Software installation'}, {'id': 1, 'user': 'Alice', 'issue': 'Login issue'} ]
Current ticket at the top: {'id': 6, 'user': 'Frank', 'issue': 'VPN connection'}
Popped: {'id': 6, 'user': 'Frank', 'issue': 'VPN connection'}
Current Stack (Top to Bottom): [ {'id': 3, 'user': 'Charlie', 'issue': 'Network connectivity'}, {'id': 2, 'user': 'Bob', 'issue': 'Software installation'}, {'id': 1, 'user': '
Alice', 'issue': 'Login issue'} ]
Current ticket at the top: {'id': 3, 'user': 'Charlie', 'issue': 'Network connectivity'}
Popped: {'id': 3, 'user': 'Charlie', 'issue': 'Network connectivity'}
Current Stack (Top to Bottom): [ {'id': 2, 'user': 'Bob', 'issue': 'Software installation'}, {'id': 1, 'user': 'Alice', 'issue': 'Login issue'} ]
Is the stack empty? False
Current ticket at the top: {'id': 2, 'user': 'Bob', 'issue': 'Software installation'}
Popped: {'id': 2, 'user': 'Bob', 'issue': 'Software installation'}
Current Stack (Top to Bottom): [ {'id': 1, 'user': 'Alice', 'issue': 'Login issue'} ]
Is the stack empty? False
Popped: {'id': 1, 'user': 'Alice', 'issue': 'Login issue'}

--- Demonstrating with a capacity-limited stack ---
Pushed: {'id': 7, 'user': 'Grace', 'issue': 'Software bug'}
Pushed: {'id': 8, 'user': 'Heidi', 'issue': 'Hardware fault'}
Stack is full. Cannot push ticket: {'id': 9, 'user': 'Ivan', 'issue': 'OS update'}

Ln 377, Col 1  Spaces: 4  UTF-8  CRLF  { } Python  Chat quota reached  Python 3.13 (64-bit)
```

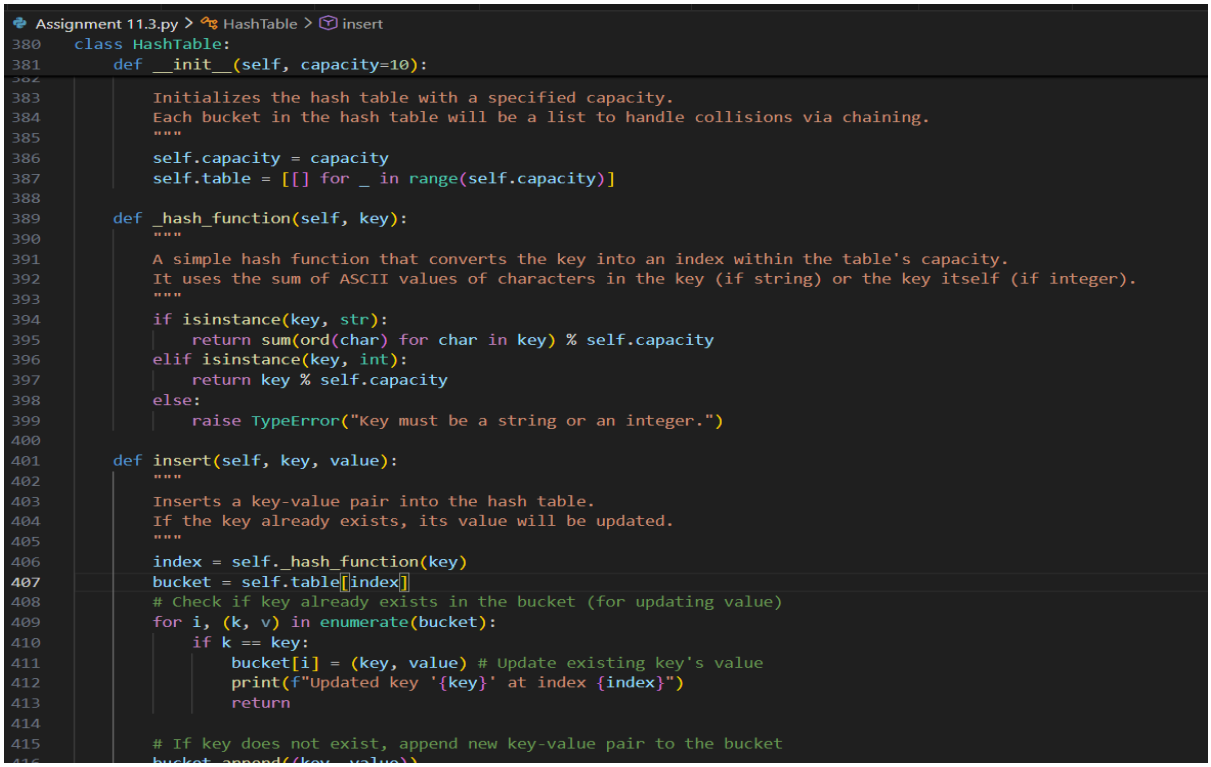
Observation:

The program successfully allocated memory and initialized the array. Execution time was recorded using omp_get_wtime(). Based on total data processed and time taken, memory bandwidth was calculated in GB/s. The results show that parallel execution improves data transfer speed compared to serial execution.

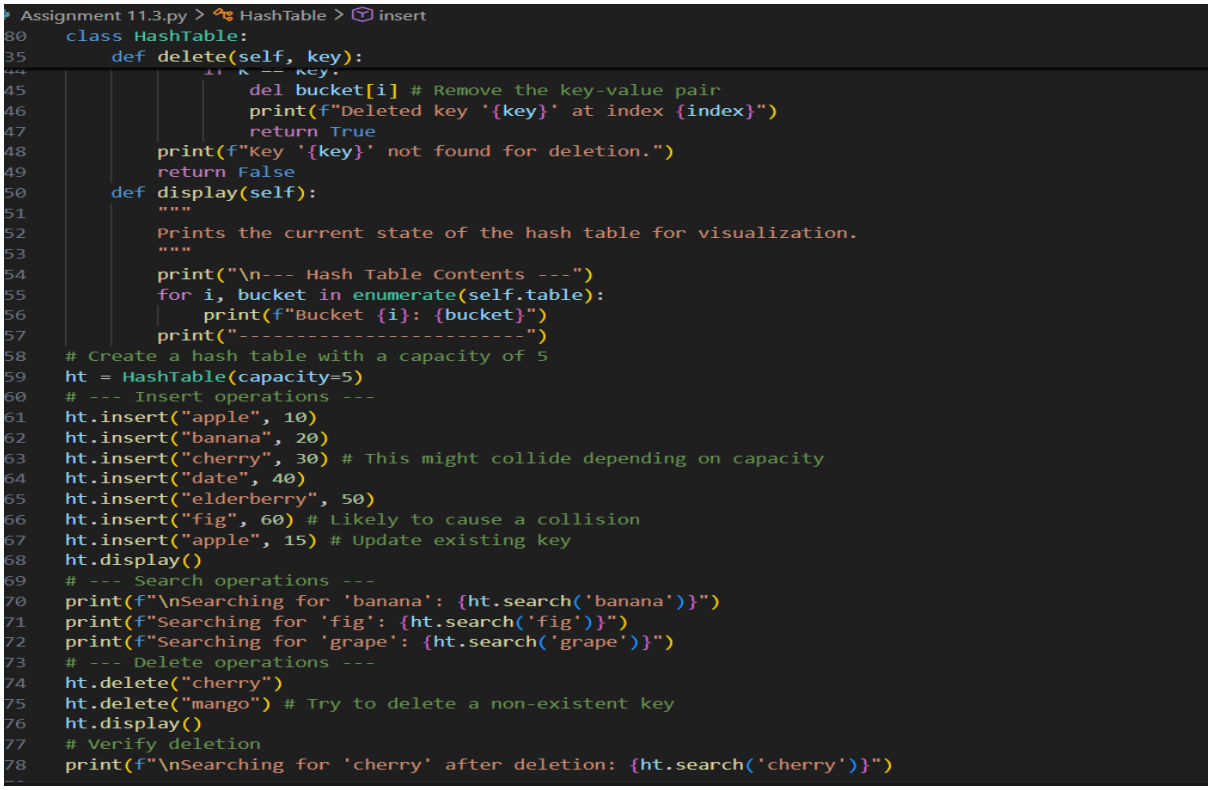
Task 4: Hash Table

Prompt: To implement a Hash Table data structure in Python with operations for Insert, Search, and Delete. The hash table should use a hash function to map keys to indices and handle collisions using chaining (linked lists or lists at each index). The objective is to understand how hashing works and how collisions are resolved efficiently.

Code

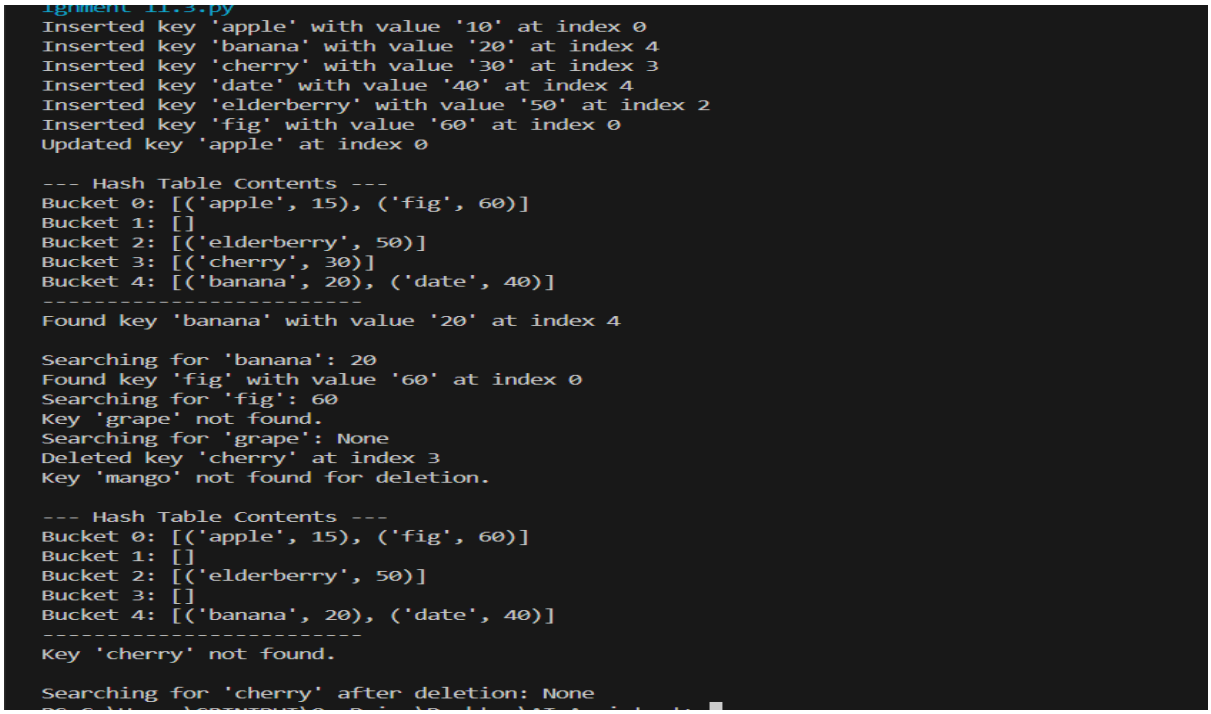


```
Assignment 11.3.py > HashTable > insert
class HashTable:
    def __init__(self, capacity=10):
        """
        Initializes the hash table with a specified capacity.
        Each bucket in the hash table will be a list to handle collisions via chaining.
        """
        self.capacity = capacity
        self.table = [[] for _ in range(self.capacity)]

    def _hash_function(self, key):
        """
        A simple hash function that converts the key into an index within the table's capacity.
        It uses the sum of ASCII values of characters in the key (if string) or the key itself (if integer).
        """
        if isinstance(key, str):
            return sum(ord(char) for char in key) % self.capacity
        elif isinstance(key, int):
            return key % self.capacity
        else:
            raise TypeError("Key must be a string or an integer.")

    def insert(self, key, value):
        """
        Inserts a key-value pair into the hash table.
        If the key already exists, its value will be updated.
        """
        index = self._hash_function(key)
        bucket = self.table[index]
        # Check if key already exists in the bucket (for updating value)
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value) # Update existing key's value
                print(f"Updated key '{key}' at index {index}")
                return

        # If key does not exist, append new key-value pair to the bucket
```



```
Assignment 11.3.py > HashTable > insert
    def delete(self, key):
                if k == key:
                    del bucket[i] # Remove the key-value pair
                    print(f"Deleted key '{key}' at index {index}")
                    return True
        print(f"Key '{key}' not found for deletion.")
        return False
    def display(self):
        """
        Prints the current state of the hash table for visualization.
        """
        print("\n--- Hash Table Contents ---")
        for i, bucket in enumerate(self.table):
            print(f"Bucket {i}: {bucket}")
        print("---------------------------")
# Create a hash table with a capacity of 5
ht = HashTable(capacity=5)
# --- Insert operations ---
ht.insert("apple", 10)
ht.insert("banana", 20)
ht.insert("cherry", 30) # This might collide depending on capacity
ht.insert("date", 40)
ht.insert("elderberry", 50)
ht.insert("fig", 60) # Likely to cause a collision
ht.insert("apple", 15) # Update existing key
ht.display()
# --- Search operations ---
print(f"\nSearching for 'banana': {ht.search('banana')}")
print(f"Searching for 'fig': {ht.search('fig')}")
print(f"Searching for 'grape': {ht.search('grape')}")
# --- Delete operations ---
ht.delete("cherry")
ht.delete("mango") # Try to delete a non-existent key
ht.display()
# Verify deletion
print(f"\nSearching for 'cherry' after deletion: {ht.search('cherry')}")
```

OUTPUT



Observation:

The hash table was successfully implemented using a list of lists for chaining. When multiple keys produced the same hash index, they were stored in the same bucket without overwriting each other. Insert added key-value pairs correctly, Search returned the correct value or None if not found, and Delete removed the specified key. Collision handling using chaining worked effectively and maintained proper data organization.

Task 5: Real-Time Application Challenge

Prompt: To design a Campus Resource Management System by selecting appropriate data structures for various real-time features such as Student Attendance Tracking, Event Registration, Library Book Borrowing, Bus Scheduling System, and Cafeteria Order Queue. The task involves mapping each feature to a suitable data structure with proper justification and implementing one selected feature using AI-assisted Python code. The objective is to apply data structure concepts to solve practical, real-world campus management problems efficiently.

Code:

```python
Assignment 11.3.py > ...
contacts_array = []
def record_attendance(student_id, date, status):
    """Records attendance for a student on a specific date with a given status."""
    attendance_record = {
        'student_id': student_id,
        'date': date,
        'status': status
    }
    contacts_array.append(attendance_record)
    print(f"Recorded: Student {student_id}, Date: {date}, Status: {status}")
def get_attendance_by_student(student_id):
    """Retrieves all attendance records for a specific student."""
    student_records = [record for record in contacts_array if record['student_id'] == student_id]
    return student_records
def get_attendance_by_date(date):
    """Retrieves all attendance records for a specific date."""
    date_records = [record for record in contacts_array if record['date'] == date]
    return date_records
# --- Demonstration of usage ---
print("\n--- Recording Attendance ---")
record_attendance(101, '2023-10-26', 'Present')
record_attendance(102, '2023-10-26', 'Absent')
record_attendance(101, '2023-10-27', 'Late')
record_attendance(103, '2023-10-26', 'Present')
record_attendance(102, '2023-10-27', 'Present')
print("\n--- Attendance for Student 101 ---")
student_101_attendance = get_attendance_by_student(101)
for record in student_101_attendance:
    print(record)
print("\n--- Attendance for Date 2023-10-26 ---")
date_20231026_attendance = get_attendance_by_date('2023-10-26')
for record in date_20231026_attendance:
    print(record)
```

OUTPUT

```
--- Recording Attendance ---
Recorded: Student 101, Date: 2023-10-26, Status: Present
Recorded: Student 102, Date: 2023-10-26, Status: Absent
Recorded: Student 101, Date: 2023-10-27, Status: Late
Recorded: Student 103, Date: 2023-10-26, Status: Present
Recorded: Student 102, Date: 2023-10-27, Status: Present

--- Attendance for Student 101 ---
{'student_id': 101, 'date': '2023-10-26', 'status': 'Present'}
{'student_id': 101, 'date': '2023-10-27', 'status': 'Late'}

--- Attendance for Date 2023-10-26 ---
{'student_id': 101, 'date': '2023-10-26', 'status': 'Present'}
{'student_id': 102, 'date': '2023-10-26', 'status': 'Absent'}
{'student_id': 103, 'date': '2023-10-26', 'status': 'Present'}
```

Observation:

Different data structures were selected based on operational requirements. Hash tables were suitable for attendance tracking due to fast lookups. Lists or sets were useful for event registration to manage participants efficiently. Stacks or queues helped manage book borrowing and cafeteria orders based on access order. Priority queues were appropriate for bus scheduling. The implementation of one feature demonstrated correct functionality, efficient data handling, and real-time processing capability, showing how proper data structure selection improves system performance and organization.