

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 > ...
1  contacts_array = []
2
3  def add_contact_array(name, phone, email):
4      """Adds a new contact to the contacts_array."""
5      contact = {'name': name, 'phone': phone, 'email': email}
6      contacts_array.append(contact)
7      print(f"Contact '{name}' added.")
8
9  def search_contact_array(name):
10     """Searches for a contact by name in the contacts_array."""
11     for contact in contacts_array:
12         if contact['name'].lower() == name.lower():
13             return contact
14     return None
15
16 def delete_contact_array(name):
17     """Deletes a contact by name from the contacts_array."""
18     global contacts_array # Moved to the top of the function
19     initial_len = len(contacts_array)
20     contacts_array = [contact for contact in contacts_array if contact['name'].lower() != name.lower()]
21     if len(contacts_array) < initial_len:
22         print(f"Contact '{name}' deleted.")
23         return True
24     else:
25         print(f"Contact '{name}' not found.")
26         return False
27
28
29 print("\n--- Demonstrating Array-based Contact Manager ---")
30
31 # 1. Add contacts
32 add_contact_array("Alice", "111-222-3333", "alice@example.com")
33 add_contact_array("Bob", "444-555-6666", "bob@example.com")
34 add_contact_array("Charlie", "777-888-9999", "charlie@example.com")
35
36 print("\nContacts after adding:")
37 print(contacts_array)
38
39 # 2. Search for a contact
40 print("\nSearching for 'Bob':")
41 found_contact = search_contact_array("Bob")
42 if found_contact:
43     print(f"Found: {found_contact}")
44 else:
45     print("Contact not found.")
46
47 print("\nSearching for 'David':")
48 found_contact = search_contact_array("David")
49 if found_contact:
50     print(f"Found: {found_contact}")
51 else:
52     print("Contact not found.")
53
54 # 3. Delete a contact
55 print("\ndeleting 'Alice':")
56 delete_contact_array("Alice")
57
58 print("\nContacts after deleting Alice:")
59 print(contacts_array)
60
61 print("\nAttempting to delete 'David' (non-existent):")
62 delete_contact_array("David")
63
64 print("\nFinal contacts array:")
65 print(contacts_array)
66
67 print("--- Array-based Contact Manager Demonstration Complete ---")
68
69
70 class Node:
71     def __init__(self, contact_data):
72         self.contact_data = contact_data
73         self.next = None
```

```

Assignment 11.3.py > ...
75 class LinkedList:
76
77     def add_contact_ll(self, name, phone, email):
78         contact = {'name': name, 'phone': phone, 'email': email}
79         new_node = Node(contact)
80         if self.head is None:
81             self.head = new_node
82         else:
83             current = self.head
84             while current.next:
85                 current = current.next
86             current.next = new_node
87         print(f"Contact '{name}' added to linked list.")
88
89     def search_contact_ll(self, name):
90         current = self.head
91         while current:
92             if current.contact_data['name'].lower() == name.lower():
93                 return current.contact_data
94             current = current.next
95         return None
96
97     def delete_contact_ll(self, name):
98         current = self.head
99         prev = None
100
101         # If head node itself holds the key to be deleted
102         if current and current.contact_data['name'].lower() == name.lower():
103             self.head = current.next
104             print(f"Contact '{name}' deleted from linked list.")
105             return True
106
107         # Search for the key to be deleted, keep track of the previous node
108         # as we need to change prev.next
109         while current and current.contact_data['name'].lower() != name.lower():
110             prev = current
111             current = current.next
112
113

```

Spaces: 4 UTF-8

```

Settings | Assignment 11.3.py > ...
75 class LinkedList:
99     def delete_contact_ll(self, name):
100
101         # If key was not present in linked list
102         if current is None:
103             print(f"Contact '{name}' not found in linked list.")
104             return False
105
106         # Unlink the node from linked list
107         prev.next = current.next
108         print(f"Contact '{name}' deleted from linked list.")
109         return True
110
111     def display_contacts_ll(self):
112         contacts = []
113         current = self.head
114         if not current:
115             print("Linked list is empty.")
116             return
117         while current:
118             contacts.append(current.contact_data)
119             current = current.next
120         for contact in contacts:
121             print(contact)
122
123     print("\n--- Demonstrating Linked List-based Contact Manager ---")
124
125     # Create an instance of the LinkedList
126     ll_contacts = LinkedList()
127
128     # 1. Add contacts
129     ll_contacts.add_contact_ll("David", "101-202-3030", "david@example.com")
130     ll_contacts.add_contact_ll("Eve", "202-303-4040", "eve@example.com")
131     ll_contacts.add_contact_ll("Frank", "303-404-5050", "frank@example.com")
132
133     print("\nContacts after adding:")
134

```

Spaces: 4 UTF-8

```

Assignment 11.3.py > ...
143 # 1. Add Contacts
144 ll_contacts.add_contact_ll("David", "101-202-3030", "david@example.com")
145 ll_contacts.add_contact_ll("Eve", "202-303-4040", "eve@example.com")
146 ll_contacts.add_contact_ll("Frank", "303-404-5050", "frank@example.com")
147
148 print("\nContacts after adding:")
149 ll_contacts.display_contacts_ll()
150
151 # 2. Search for a contact
152 print("\nSearching for 'Eve':")
153 found_ll_contact = ll_contacts.search_contact_ll("Eve")
154 if found_ll_contact:
155     print(f"Found: {found_ll_contact}")
156 else:
157     print("Contact not found.")
158
159 print("\nSearching for 'Grace':")
160 found_ll_contact = ll_contacts.search_contact_ll("Grace")
161 if found_ll_contact:
162     print(f"Found: {found_ll_contact}")
163 else:
164     print("Contact not found.")
165
166 # 3. Delete a contact
167 print("\nDeleting 'David':")
168 ll_contacts.delete_contact_ll("David")
169
170 print("\nContacts after deleting David:")
171 ll_contacts.display_contacts_ll()
172
173 print("\nAttempting to delete 'Grace' (non-existent):")
174 ll_contacts.delete_contact_ll("Grace")
175
176 print("\nFinal linked list contacts:")
177 ll_contacts.display_contacts_ll()
178
179 print("\n--- Linked List-based Contact Manager Demonstration Complete ---")

```

OUTPUT:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
problems (Ctrl+Shift+M) Y

--- Demonstrating Array-based Contact Manager ---
Contact 'Alice' added.
Contact 'Bob' added.
Contact 'Charlie' added.

Contacts after adding:
[{'name': 'Alice', 'phone': '111-222-3333', 'email': 'alice@example.com'}, {'name': 'Bob', 'phone': '444-555-6666', 'email': 'bob@example.com'}, {'name': 'Charlie', 'phone': '777-888-9999', 'email': 'charlie@example.com'}]

Searching for 'Bob':
Found: {'name': 'Bob', 'phone': '444-555-6666', 'email': 'bob@example.com'}

Searching for 'David':
Contact not found.

Deleting 'Alice':
Contact 'Alice' deleted.

Contacts after deleting Alice:
[{'name': 'Bob', 'phone': '444-555-6666', 'email': 'bob@example.com'}, {'name': 'Charlie', 'phone': '777-888-9999', 'email': 'charlie@example.com'}]

Attempting to delete 'David' (non-existent):
Contact 'David' not found.

Final contacts array:
[{'name': 'Bob', 'phone': '444-555-6666', 'email': 'bob@example.com'}, {'name': 'Charlie', 'phone': '777-888-9999', 'email': 'charlie@example.com'}]
--- Array-based Contact Manager Demonstration Complete ---

--- Demonstrating Linked List-based Contact Manager ---
Contact 'David' added to linked list.
Contact 'Eve' added to linked list.
Contact 'Frank' added to linked list.

Contacts after adding:
{'name': 'David', 'phone': '101-202-3030', 'email': 'david@example.com'}
{'name': 'Eve', 'phone': '202-303-4040', 'email': 'eve@example.com'}
{'name': 'Frank', 'phone': '303-404-5050', 'email': 'frank@example.com'}

Searching for 'Eve':

```

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 > ...
183 import collections
184 import heapq
185 # --- 1. Implement a Queue (FIFO) to manage book requests ---
186 class Queue:
187     def __init__(self):
188         self.items = collections.deque()
189     def enqueue(self, request):
190         """Adds a request to the end of the queue."""
191         self.items.append(request)
192         print(f"Enqueued: {request}")
193     def dequeue(self):
194         """Removes and returns the request from the front of the queue."""
195         if not self.is_empty():
196             request = self.items.popleft()
197             print(f"Dequeued: {request}")
198             return request
199         print("Queue is empty. Cannot dequeue.")
200         return None
201     def is_empty(self):
202         """Checks if the queue is empty."""
203         return len(self.items) == 0
204     def size(self):
205         """Returns the number of requests in the queue."""
206         return len(self.items)
207     def display(self):
208         """Displays all items in the queue."""
209         print("Current Queue: [" + ", ".join(str(item) for item in self.items) + "]")
210 print("--- Demonstrating Standard Queue (FIFO) ---")
211 fifo_queue = Queue()
212 fifo_queue.enqueue({"user": "Student A", "book": "Math Textbook", "type": "student"})
213 fifo_queue.enqueue({"user": "Student B", "book": "History Novel", "type": "student"})
214 fifo_queue.enqueue({"user": "Faculty X", "book": "Research Paper", "type": "faculty"})
215 fifo_queue.display()
216 fifo_queue.dequeue()
217 fifo_queue.display()
218 fifo_queue.enqueue({"user": "Student C", "book": "Science Journal", "type": "student"})
219 fifo_queue.display()

Assignment 11.3.py > ...
221 fifo_queue.dequeue()
222 fifo_queue.dequeue()
223 fifo_queue.dequeue()
224 print("-----")
225 # --- 2. Extend the system to a Priority Queue, prioritizing faculty requests ---
226 class PriorityQueue:
227     def __init__(self):
228         self.heap = [] # Stores (priority, index, request) tuples
229         self.counter = 0 # Unique entry ID for stable ordering
230     def enqueue(self, request):
231         """Adds a request to the priority queue with appropriate priority."""
232         # Faculty requests have higher priority (e.g., 0 for faculty, 1 for student)
233         priority = 0 if request["type"] == "faculty" else 1
234         heapq.heappush(self.heap, (priority, self.counter, request))
235         self.counter += 1
236         print(f"Enqueued (P{priority}): {request}")
237     def dequeue(self):
238         """Removes and returns the highest priority request."""
239         if not self.is_empty():
240             priority, _, request = heapq.heappop(self.heap)
241             print(f"Dequeued (P{priority}): {request}")
242             return request
243         print("Priority Queue is empty. Cannot dequeue.")
244         return None
245     def is_empty(self):
246         """Checks if the priority queue is empty."""
247         return len(self.heap) == 0
248     def size(self):
249         """Returns the number of requests in the priority queue."""
250         return len(self.heap)
251     def display(self):
252         """Displays all items in the priority queue (order not guaranteed due to heap structure)."""
253         # For display, we can sort a copy to show current logical order if needed
254         sorted_requests = sorted(self.heap, key=lambda x: x[0])
255         print("Current Priority Queue (logical order): [" + ", ".join(f"P{p} {req}" for p, _, req in sorted_requests) + "]")
256 print("\n--- Demonstrating Priority Queue ---")
257 pq = PriorityQueue()
```

```

Assignment 11.3.py > ...
255     print("Current Priority Queue (logical order): [", ", ".join(f"({p}) {req}" for p, req in sorted_requests), "]")
256     print("\n--- Demonstrating Priority Queue ---")
257     pq = PriorityQueue()
258     pq.enqueue({"user": "Student A", "book": "History Book", "type": "student"})
259     pq.enqueue({"user": "Faculty X", "book": "Advanced Physics", "type": "faculty"})
260     pq.enqueue({"user": "Student B", "book": "Fiction Novel", "type": "student"})
261     pq.enqueue({"user": "Faculty Y", "book": "Quantum Mechanics", "type": "faculty"})
262     pq.enqueue({"user": "Student C", "book": "Biography", "type": "student"})
263     pq.display()
264     print("\nDequeuing requests:")
265     pq.dequeue()
266     pq.display()
267     pq.dequeue()
268     pq.display()
269     pq.dequeue()
270     pq.display()
271     pq.dequeue()
272     pq.display()
273     pq.dequeue()
274     pq.display()
275     pq.dequeue()
276     print("-----")

```

OUTPUT:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
assignment 11.3.py
--- Demonstrating Standard Queue (FIFO) ---
Enqueued: {'user': 'Student A', 'book': 'Math Textbook', 'type': 'student'}
Enqueued: {'user': 'Student B', 'book': 'History Novel', 'type': 'student'}
Enqueued: {'user': 'Faculty X', 'book': 'Research Paper', 'type': 'faculty'}
Current Queue: [ {'user': 'Student A', 'book': 'Math Textbook', 'type': 'student'}, {'user': 'Student B', 'book': 'History Novel', 'type': 'student'}, {'user': 'Faculty X', 'book': 'Research Paper', 'type': 'faculty'} ]
Dequeued: {'user': 'Student A', 'book': 'Math Textbook', 'type': 'student'}
Current Queue: [ {'user': 'Student B', 'book': 'History Novel', 'type': 'student'}, {'user': 'Faculty X', 'book': 'Research Paper', 'type': 'faculty'} ]
Enqueued: {'user': 'Student C', 'book': 'Science Journal', 'type': 'student'}
Current Queue: [ {'user': 'Student B', 'book': 'History Novel', 'type': 'student'}, {'user': 'Faculty X', 'book': 'Research Paper', 'type': 'faculty'}, {'user': 'Student C', 'book': 'Science Journal', 'type': 'student'} ]
Dequeued: {'user': 'Student B', 'book': 'History Novel', 'type': 'student'}
Dequeued: {'user': 'Faculty X', 'book': 'Research Paper', 'type': 'faculty'}
Dequeued: {'user': 'Student C', 'book': 'Science Journal', 'type': 'student'}
Queue is empty. Cannot dequeue.
-----

--- Demonstrating Priority Queue ---
Enqueued (P1): {'user': 'Student A', 'book': 'History Book', 'type': 'student'}
Enqueued (P0): {'user': 'Faculty X', 'book': 'Advanced Physics', 'type': 'faculty'}
Enqueued (P1): {'user': 'Student B', 'book': 'Fiction Novel', 'type': 'student'}
Enqueued (P0): {'user': 'Faculty Y', 'book': 'Quantum Mechanics', 'type': 'faculty'}
Enqueued (P1): {'user': 'Student C', 'book': 'Biography', 'type': 'student'}
Current Priority Queue (logical order): [ (P0) {'user': 'Faculty X', 'book': 'Advanced Physics', 'type': 'faculty'}, (P0) {'user': 'Faculty Y', 'book': 'Quantum Mechanics', 'type': 'faculty'}, (P1) {'user': 'Student B', 'book': 'Fiction Novel', 'type': 'student'}, (P1) {'user': 'Student A', 'book': 'History Book', 'type': 'student'}, (P1) {'user': 'Student C', 'book': 'Biography', 'type': 'student'} ]

Dequeuing requests:
Dequeued (P0): {'user': 'Faculty X', 'book': 'Advanced Physics', 'type': 'faculty'}
Current Priority Queue (logical order): [ (P0) {'user': 'Faculty Y', 'book': 'Quantum Mechanics', 'type': 'faculty'}, (P1) {'user': 'Student A', 'book': 'History Book', 'type': 'student'}, (P1) {'user': 'Student B', 'book': 'Fiction Novel', 'type': 'student'}, (P1) {'user': 'Student C', 'book': 'Biography', 'type': 'student'} ]
Dequeued (P0): {'user': 'Faculty Y', 'book': 'Quantum Mechanics', 'type': 'faculty'}
Current Priority Queue (logical order): [ (P1) {'user': 'Student A', 'book': 'History Book', 'type': 'student'}, (P1) {'user': 'Student B', 'book': 'Fiction Novel', 'type': 'student'}, (P1) {'user': 'Student C', 'book': 'Biography', 'type': 'student'} ]
Dequeued (P1): {'user': 'Student A', 'book': 'History Book', 'type': 'student'}
Current Priority Queue (logical order): [ (P1) {'user': 'Student B', 'book': 'Fiction Novel', 'type': 'student'}, (P1) {'user': 'Student C', 'book': 'Biography', 'type': 'student'} ]
Dequeued (P1): {'user': 'Student B', 'book': 'Fiction Novel', 'type': 'student'}
Current Priority Queue (logical order): [ (P1) {'user': 'Student C', 'book': 'Biography', 'type': 'student'} ]
Dequeued (P1): {'user': 'Student C', 'book': 'Biography', 'type': 'student'}

```

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:

```
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)
```

```
Assignment 11.3.py > ...
280 class Stack:
281     """
282     """
283     def is_full(self):
284         """Checks if the stack is full (if a capacity is set)."""
285         return self.capacity is not None and len(self.items) >= self.capacity
286
287     def display(self):
288         """Displays all items in the stack from top to bottom."""
289         if self.is_empty():
290             print("Current Stack: [Empty]")
291         else:
292             # Display from top to bottom (last item in list is top of stack)
293             print("Current Stack (Top to Bottom): [", ", ".join(str(item) for item in reversed(self.items)), "]")
294
295     # Initialize the Help Desk Stack
296     help_desk_stack = Stack()
297
298     print("--- Raising Tickets ---")
299     help_desk_stack.push({"id": 1, "user": "Alice", "issue": "Login issue"})
300     help_desk_stack.push({"id": 2, "user": "Bob", "issue": "Software installation"})
301     help_desk_stack.push({"id": 3, "user": "Charlie", "issue": "Network connectivity"})
302     help_desk_stack.push({"id": 4, "user": "Diana", "issue": "Printer not working"})
303     help_desk_stack.push({"id": 5, "user": "Eve", "issue": "Email configuration"})
304
305     print(f"Stack size: {help_desk_stack.size()}")
306     help_desk_stack.display()
307
308     print("\n--- Resolving Tickets (LIFO) ---")
309     print(f"Current ticket at the top: {help_desk_stack.peek()}")
310     help_desk_stack.pop()
311     help_desk_stack.display()
312
313     print(f"Current ticket at the top: {help_desk_stack.peek()}")
314     help_desk_stack.pop()
315     help_desk_stack.display()
316
317     help_desk_stack.push({"id": 6, "user": "Frank", "issue": "VPN connection"}) # New ticket comes in
318     help_desk_stack.display()
```

```

Assignment 11.3.py > ...
help_desk_stack.display()
41
42 print("\n--- Resolving Tickets (LIFO) ---")
43 print(f"Current ticket at the top: {help_desk_stack.peek()}")
44 help_desk_stack.pop()
45 help_desk_stack.display()
46
47 print(f"Current ticket at the top: {help_desk_stack.peek()}")
48 help_desk_stack.pop()
49 help_desk_stack.display()
50
51 help_desk_stack.push({"id": 6, "user": "Frank", "issue": "VPN connection"}) # New ticket comes in
52 help_desk_stack.display()
53
54 print(f"Current ticket at the top: {help_desk_stack.peek()}")
55 help_desk_stack.pop()
56 help_desk_stack.display()
57
58 print(f"Current ticket at the top: {help_desk_stack.peek()}")
59 help_desk_stack.pop()
60 help_desk_stack.display()
61
62 print(f"Is the stack empty? {help_desk_stack.is_empty()}")
63 print(f"Current ticket at the top: {help_desk_stack.peek()}")
64 help_desk_stack.pop()
65 help_desk_stack.display()
66
67 print(f"Is the stack empty? {help_desk_stack.is_empty()}")
68 help_desk_stack.pop() # Try to pop from an empty stack
69
70 print("\n--- Demonstrating with a capacity-limited stack ---")
71 limited_stack = Stack(capacity=2)
72 limited_stack.push({"id": 7, "user": "Grace", "issue": "Software bug"})
73 limited_stack.push({"id": 8, "user": "Heidi", "issue": "Hardware fault"})
74 limited_stack.push({"id": 9, "user": "Ivan", "issue": "OS update"}) # This should fail
75 limited_stack.display()
76 print(f"Is the limited stack full? {limited_stack.is_full()}")
77

```

OUTPUT:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Assignment 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': 'Charlie', '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'}

```

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
380 class HashTable:
381     def __init__(self, capacity=10):
382         """
383         Initializes the hash table with a specified capacity.
384         Each bucket in the hash table will be a list to handle collisions via chaining.
385         """
386         self.capacity = capacity
387         self.table = [[] for _ in range(self.capacity)]
388
389     def _hash_function(self, key):
390         """
391         A simple hash function that converts the key into an index within the table's capacity.
392         It uses the sum of ASCII values of characters in the key (if string) or the key itself (if integer).
393         """
394         if isinstance(key, str):
395             return sum(ord(char) for char in key) % self.capacity
396         elif isinstance(key, int):
397             return key % self.capacity
398         else:
399             raise TypeError("Key must be a string or an integer.")
400
401     def insert(self, key, value):
402         """
403         Inserts a key-value pair into the hash table.
404         If the key already exists, its value will be updated.
405         """
406         index = self._hash_function(key)
407         bucket = self.table[index]
408         # Check if key already exists in the bucket (for updating value)
409         for i, (k, v) in enumerate(bucket):
410             if k == key:
411                 bucket[i] = (key, value) # Update existing key's value
412                 print(f"Updated key '{key}' at index {index}")
413                 return
414
415         # If key does not exist, append new key-value pair to the bucket
416         bucket.append((key, value))

```

```
Assignment 11.3.py > HashTable > insert
80 class HashTable:
81     def delete(self, key):
82         """
83         Removes a key-value pair from the hash table.
84         """
85         index = self._hash_function(key)
86         bucket = self.table[index]
87         if key in bucket:
88             del bucket[bucket.index((key, _))] # Remove the key-value pair
89             print(f"Deleted key '{key}' at index {index}")
90             return True
91         print(f"Key '{key}' not found for deletion.")
92         return False
93
94     def display(self):
95         """
96         Prints the current state of the hash table for visualization.
97         """
98         print("\n--- Hash Table Contents ---")
99         for i, bucket in enumerate(self.table):
100             print(f"Bucket {i}: {bucket}")
101         print("-----")
102
103 # Create a hash table with a capacity of 5
104 ht = HashTable(capacity=5)
105 # --- Insert operations ---
106 ht.insert("apple", 10)
107 ht.insert("banana", 20)
108 ht.insert("cherry", 30) # This might collide depending on capacity
109 ht.insert("date", 40)
110 ht.insert("elderberry", 50)
111 ht.insert("fig", 60) # Likely to cause a collision
112 ht.insert("apple", 15) # Update existing key
113 ht.display()
114 # --- Search operations ---
115 print(f"\nSearching for 'banana': {ht.search('banana')}")
116 print(f"Searching for 'fig': {ht.search('fig')}")
117 print(f"Searching for 'grape': {ht.search('grape')}")
118 # --- Delete operations ---
119 ht.delete("cherry")
120 ht.delete("mango") # Try to delete a non-existent key
121 ht.display()
122 # Verify deletion
123 print(f"\nSearching for 'cherry' after deletion: {ht.search('cherry')}")

```


OUTPUT

```
Assignment 11.3.py
Inserted key 'apple' with value '10' at index 0
Inserted key 'banana' with value '20' at index 4
Inserted key 'cherry' with value '30' at index 3
Inserted key 'date' with value '40' at index 4
Inserted key 'elderberry' with value '50' at index 2
Inserted key 'fig' with value '60' at index 0
Updated key 'apple' at index 0

--- Hash Table Contents ---
Bucket 0: [('apple', 15), ('fig', 60)]
Bucket 1: []
Bucket 2: [('elderberry', 50)]
Bucket 3: [('cherry', 30)]
Bucket 4: [('banana', 20), ('date', 40)]
-----
Found key 'banana' with value '20' at index 4

Searching for 'banana': 20
Found key 'fig' with value '60' at index 0
Searching for 'fig': 60
Key 'grape' not found.
Searching for 'grape': None
Deleted key 'cherry' at index 3
Key 'mango' not found for deletion.

--- Hash Table Contents ---
Bucket 0: [('apple', 15), ('fig', 60)]
Bucket 1: []
Bucket 2: [('elderberry', 50)]
Bucket 3: []
Bucket 4: [('banana', 20), ('date', 40)]
-----
Key 'cherry' not found.

Searching for 'cherry' after deletion: None
```

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:

```

Assignment 11.3.py > ...
82 contacts_array = []
83 def record_attendance(student_id, date, status):
84     """Records attendance for a student on a specific date with a given status."""
85     attendance_record = {
86         'student_id': student_id,
87         'date': date,
88         'status': status
89     }
90     contacts_array.append(attendance_record)
91     print(f"Recorded: Student {student_id}, Date: {date}, Status: {status}")
92 def get_attendance_by_student(student_id):
93     """Retrieves all attendance records for a specific student."""
94     student_records = [record for record in contacts_array if record['student_id'] == student_id]
95     return student_records
96 def get_attendance_by_date(date):
97     """Retrieves all attendance records for a specific date."""
98     date_records = [record for record in contacts_array if record['date'] == date]
99     return date_records
100 # --- Demonstration of usage ---
101 print("\n--- Recording Attendance ---")
102 record_attendance(101, '2023-10-26', 'Present')
103 record_attendance(102, '2023-10-26', 'Absent')
104 record_attendance(101, '2023-10-27', 'Late')
105 record_attendance(103, '2023-10-26', 'Present')
106 record_attendance(102, '2023-10-27', 'Present')
107 print("\n--- Attendance for Student 101 ---")
108 student_101_attendance = get_attendance_by_student(101)
109 for record in student_101_attendance:
110     print(record)
111 print("\n--- Attendance for Date 2023-10-26 ---")
112 date_20231026_attendance = get_attendance_by_date('2023-10-26')
113 for record in date_20231026_attendance:
114     print(record)
115

```

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.