

AI Assisted Coding

Assignment 11.3

Name: V Rithik Reddy

Hall ticket no: 2303A51263

Batch no: 19

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

Generate Python code to implement a Contact Manager system using:

- Array (Python list)
- Linked List

The system must support:

- Add contact
- Search contact
- Delete contact

Use meaningful class names, proper methods, and include comments.

Code & Output (Arrays):

```
assignment_11.3.py X  kcp.py
AI_Assistant_coding > assignment_11.3.py > ...
1  #Task 1: Smart Contact Manager (Arrays)
2  # Generate Python code to implement a Contact Manager system using:
3  # * Array (Python list)
4  # * Linked List
5  # The system must support:
6  # * Add contact
7  # * Search contact
8  # * Delete contact
9  # Use meaningful class names, proper methods, and include comments.
10
11 class ContactManagerArray:
12     def __init__(self):
13         self.contacts = [] # Using a list to store contacts
14     def add_contact(self, name, phone):
15         '''Add a new contact to the list.'''
16         self.contacts.append({'name': name, 'phone': phone})
17     def search_contact(self, name):
18         '''Search for a contact by name.'''
19         for contact in self.contacts:
20             if contact['name'] == name:
21                 return contact
22         return None # Contact not found
23     def delete_contact(self, name):
24         '''Delete a contact by name.'''
25         for i, contact in enumerate(self.contacts):
26             if contact['name'] == name:
27                 del self.contacts[i]
28             return True # Contact deleted
29         return False # Contact not found
30 manager = ContactManagerArray()
31 manager.add_contact("Alice", "123-456-7890")
32 manager.add_contact("Bob", "987-654-3210")
33 print(manager.search_contact("Alice")) # Output: {'name': 'Alice', 'phone': '123-456-7890'}
34 print(manager.delete_contact("Bob")) # Output: True
35 print(manager.search_contact("Bob")) # Output: None
```

TERMINAL

```
PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding> & C:/Users/hariv/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/hariv/OneDrive/Documents/SRU/3 year II sem/AI_Assistant_coding/assignment_11.3.py"
● {'name': 'Alice', 'phone': '123-456-7890'}
True
None
○ PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding>
```

Explanation(rrays):

This implementation uses a Python list to store contact dictionaries. Adding contacts is efficient ($O(1)$ average). Searching and deletion require linear traversal ($O(n)$). The array approach is simple and easy to implement but less efficient for frequent deletions in large datasets.

Code & Output (Linked-Lists):

The screenshot shows a code editor with two tabs: `assignment_11.3.py` and `kk.py`. The `assignment_11.3.py` tab contains the following Python code:

```

1 "Task 1: Smart Contact Manager (Linked Lists)"
2 class ContactNode:
3     def __init__(self, name, phone):
4         self.name = name
5         self.phone = phone
6         self.next = None # Pointer to the next contact
7 class ContactManagerLinkedList:
8     def __init__(self):
9         self.head = None # Start of the linked list
10    def add_contact(self, name, phone):
11        #Add a new contact to the linked list.
12        new_node = ContactNode(name, phone)
13        new_node.next = self.head # Point new node to the current head
14        self.head = new_node # Update head to the new node
15    def search_contact(self, name):
16        #Search for a contact by name.
17        current = self.head
18        while current:
19            if current.name == name:
20                return {'name': current.name, 'phone': current.phone}
21            current = current.next
22        return None # Contact not found
23    def delete_contact(self, name):
24        #Delete a contact by name.
25        current = self.head
26        previous = None
27        while current:
28            if current.name == name:
29                if previous: # If it's not the head node
30                    previous.next = current.next
31                else: # If it's the head node
32                    self.head = current.next
33                return True # Contact deleted
34            previous = current
35            current = current.next
36        return False # Contact not found
37 manager_linked_list = ContactManagerLinkedList()
38 manager_linked_list.add_contact("Charlie", "555-555-5555")
39 manager_linked_list.add_contact("Dave", "444-444-4444")
40 print(manager_linked_list.search_contact("Charlie")) # Output: {'name': 'Charlie', 'phone': '555-555-5555'}
41 print(manager_linked_list.delete_contact("Dave")) # Output: True
42 print(manager_linked_list.search_contact("Dave")) # Output: None

```

The terminal window below shows the execution of the script:

```

PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding> & C:/Users/hariv/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/hariv/OneDrive/Documents/SRU/3 year II sem/AI_Assistant_coding/assignment_11.3.py"
● {'name': 'Charlie', 'phone': '555-555-5555'}
True
None
○ PS C:\Users\hariv\OneDrive\Documents\SRU\3 year II sem\AI_Assistant_coding>

```

Explanation (Linked-Lists):

The linked list implementation allows dynamic memory allocation. Insertion at the beginning is O(1). Searching and deletion are O(n). Unlike arrays, linked lists avoid shifting elements during deletion. However, they require extra memory for pointers and are slightly more complex to implement.

Comparision (Arrays VS Linked-Lists):

- **Insertion Efficiency:** Linked List (O(1) at head) is better than array when frequent insertions occur.
- **Deletion Efficiency:** Linked List avoids shifting elements.
- **Search Efficiency:** Both require O(n).
- **Memory Usage:** Array is more memory-efficient.

Task 2: Library Book Search System (Queue & Priority Queue) Prompt:

Generate Python code to implement:

- A Queue (FIFO)
- A Priority Queue prioritizing faculty requests over student requests

Include enqueue and dequeue methods.

Code & Output (Queue):

The screenshot shows a code editor with two panes. The left pane displays the Python code for a Queue class, and the right pane shows the terminal output of running the script.

```
Assignment_113.py > ...
82 "Task 2: Library Book Search System (Queue)"
83 # Generate Python code to implement:
84 # A Queue (FIFO)
85 # A Priority Queue prioritizing faculty requests over students
86 # Include enqueue and dequeue methods.
87 from collections import deque
88
89 class Queue:
90     def __init__(self):
91         self.queue = deque() # Using deque for efficiency
92     def enqueue(self, item):
93         '''Add an item to the end of the queue.'''
94         self.queue.append(item)
95     def dequeue(self):
96         '''Remove and return the item at the front of the queue.'''
97         if not self.is_empty():
98             return self.queue.popleft()
99         return None # Queue is empty
100    def is_empty(self):
101        '''Check if the queue is empty.'''
102        return len(self.queue) == 0
103
104 library_queue = Queue()
105 library_queue.enqueue("Student Request: Book A")
106 library_queue.enqueue("Faculty Request: Book B")
107
108 print(library_queue.dequeue()) # Output: "Student Request: Book A"
109
110
```

The terminal output shows:

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u "E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3\Assignment_11.3.py"
Student Request: Book A
Faculty Request: Book B
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation(Queue):

The queue follows FIFO (First In, First Out). Requests are processed in the order they arrive. This is suitable for standard book request management.

Code & Output (Priority Queue):

The screenshot shows a code editor with two panes. The left pane displays the Python code for a Priority Queue class using a heap, and the right pane shows the terminal output of running the script.

```
Assignment_113.py > ...
112 "Task 2: Library Book Search System (Priority Queue)"
113 import heapq
114
115 class PriorityQueue:
116     def __init__(self):
117         self.queue = [] # Using a List to store the priority queue
118     def enqueue(self, item, priority):
119         '''Add an item with a given priority to the queue.'''
120         heapq.heappush(self.queue, (priority, item))
121     def dequeue(self):
122         '''Remove and return the item with the highest priority (Lowest number).'''
123         if not self.is_empty():
124             return heapq.heappop(self.queue)[1] # Return the item, not the priority
125         return None # Queue is empty
126     def is_empty(self):
127         '''Check if the priority queue is empty.'''
128         return len(self.queue) == 0
129
130 priority_queue = PriorityQueue()
131 priority_queue.enqueue("Student Request: Book A", priority=2) # Lower priority
132 priority_queue.enqueue("Faculty Request: Book B", priority=1) # Higher priority
133
134 print(priority_queue.dequeue()) # Output: "Faculty Request: Book B"
```

The terminal output shows:

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3\Assignment_11.3.py
Student Request: Book A
Faculty Request: Book B
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation (Priority Queue):

The priority queue uses a heap. Faculty requests are assigned higher priority (lower numeric value). This ensures faculty members are served before students.

Task 3: Emergency Help Desk (Stack) Prompt:

Generate Python code to implement a Stack for managing support tickets with push, pop, peek, is_empty methods.

Code & Output:

The screenshot shows a code editor with two tabs: 'Assignment_113.py' and 'Assignment_113.3'. The code in 'Assignment_113.py' is as follows:

```
132 # Task 3: Emergency Help Desk (Stack)
133 # Generate Python code to implement a Stack for managing support
134 # tickets with push, pop, peek, is_empty methods.
135
136 class Stack:
137     def __init__(self):
138         self.stack = [] # Using a List to store stack items
139
140     def push(self, item):
141         '''Add an item to the top of the stack.'''
142         self.stack.append(item)
143
144     def pop(self):
145         '''Remove and return the item at the top of the stack.'''
146         if not self.is_empty():
147             return self.stack.pop()
148         return None # Stack is empty
149
150     def peek(self):
151         '''Return the item at the top of the stack without
152         removing it.'''
153         if not self.is_empty():
154             return self.stack[-1]
155         return None # Stack is empty
156
157     def is_empty(self):
158         '''Check if the stack is empty.'''
159         return len(self.stack) == 0
160
161 # Example Usage
162 help_desk_stack = Stack()
163 help_desk_stack.push("Support Ticket 1")
164 help_desk_stack.push("Support Ticket 2")
165 print(help_desk_stack.peek()) # Output: "Support Ticket 2"
166 print(help_desk_stack.pop()) # Output: "Support Ticket 2"
167 print(help_desk_stack.is_empty()) # Output: False
```

The terminal window shows the execution of the code:

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3\Assignment_11.3.py"
Support Ticket 2
Support Ticket 2
False
```

Explanation:

The stack manages support tickets using LIFO order, where the most recent ticket is resolved first. Push, pop, and peek operations demonstrate escalation handling effectively. This structure is suitable for urgent issue resolution workflows. AI assistance helped design stack methods and improve operational clarity.

Task 4: Hash Table Prompt:

Generate a Python HashTable class with insert, search, and delete methods using collision handling through chaining.

Code & Output:

```

File Edit Selection View Go Run Terminal Help ← →
Assignment_113.py ...
Assignment_113.py > ...
Task 4: Hash Table
# Generate a Python HashTable class with insert, search, and delete methods using collision handling through chaining.
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[None] for _ in range(size)] # Create a list of empty lists for chaining
    def hash(self, key):
        return hash(key) % self.size
    def insert(self, key, value):
        index = self.hash(key)
        # Check if the key already exists and update it
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value) # Update existing key
                return
        # If key does not exist, add new key-value pair
        self.table[index].append((key, value))
    def search(self, key):
        index = self.hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v # Return the value associated with the key
        return None # Key not found
    def delete(self, key):
        index = self.hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i] # Remove the key-value pair
                return True # Deletion successful
        return False # Key not found
# Example usage
hash_table = HashTable()
hash_table.insert("name", "Alice")
hash_table.insert("age", 30)
print(hash_table.search("name")) # Output: "Alice"
print(hash_table.delete("age")) # Output: True
print(hash_table.search("age")) # Output: None

```

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3\Assignment_11.3.py"

Alice
True
None

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>

Explanation:

The hash table stores data using a hashing function to determine storage index. Collision handling is done using chaining, allowing multiple elements per bucket. This ensures efficient average-time operations. AI helped generate structured bucket management logic.

Task 5: Real-Time Application Challenge

Prompt:

Design a Campus Resource Management feature and implement one selected feature using an appropriate data structure.

Code & Output:

The screenshot shows a code editor with two tabs: 'Assignment_11.3.py' and 'Code'. The 'Assignment_11.3.py' tab displays Python code for a 'CampusResourceManager' class. The 'Code' tab shows the terminal output of running the script.

```
Assignment_11.3.py
282
283 "Task 5: Real-Time Application Challenge"
284 # Design a Campus Resource Management feature and implement one selected feature
285 # using an appropriate data structure.
286 class CampusResourceManager:
287     def __init__(self):
288         self.resources = {} # Using a dictionary to manage resources
289     def add_resource(self, resource_name, quantity):
290         '''Add a resource with its quantity.'''
291         if resource_name in self.resources:
292             self.resources[resource_name] += quantity # Update existing resource
293         else:
294             self.resources[resource_name] = quantity # Add new resource
295     def search_resource(self, resource_name):
296         '''Search for a resource by name.'''
297         return self.resources.get(resource_name, None) # Return quantity or None if
298         not found
299     def delete_resource(self, resource_name):
300         '''Delete a resource by name.'''
301         if resource_name in self.resources:
302             del self.resources[resource_name] # Remove the resource
303             return True # Deletion successful
304         return False # Resource not found
305
306 # Example Usage
307 campus_manager = CampusResourceManager()
308 campus_manager.add_resource("Projector", 5)
309 campus_manager.add_resource("Whiteboard", 10)
310 print(campus_manager.search_resource("Projector")) # Output: 5
311 print(campus_manager.delete_resource("Whiteboard")) # Output: True
312 print(campus_manager.search_resource("Whiteboard")) # Output: None
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python Assignment_11.3.py
5
True
None
```

Explanation:

The cafeteria system uses a queue to maintain FIFO order of service. Customers are served in the order they arrive, ensuring fairness. This data structure matches real-world queue behavior. AI assistance helped implement and structure the queue methods efficiently.