

AI Assisted Coding

Assignment 11.3

Name: ch. koushik

Hall ticket no: 2303A51938

Batch no: 19

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

Prompt:

Generate Python code to implement a Contact Manager system using:

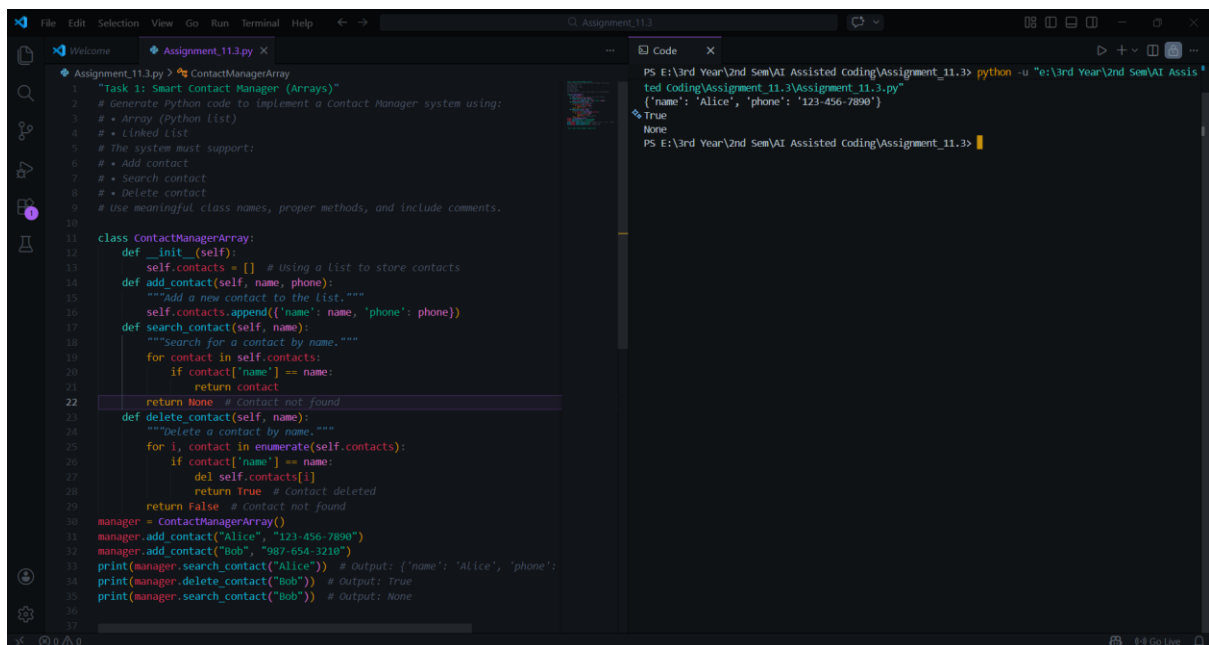
- Array (Python list)
- Linked List

The system must support:

- Add contact
- Search contact
- Delete contact

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

Code & Output (Arrays):



```
File Edit Selection View Go Run Terminal Help
Assignment_11.3
Welcome
Assignment_11.3.py X
Task 1: Smart Contact Manager (Arrays)
# 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.

class ContactManagerArray:
    def __init__(self):
        self.contacts = [] # Using a list to store contacts
    def add_contact(self, name, phone):
        """Add a new contact to the list."""
        self.contacts.append({'name': name, 'phone': phone})
    def search_contact(self, name):
        """Search for a contact by name."""
        for contact in self.contacts:
            if contact['name'] == name:
                return contact
        return None # Contact not found
    def delete_contact(self, name):
        """Delete a contact by name."""
        for i, contact in enumerate(self.contacts):
            if contact['name'] == name:
                del self.contacts[i]
                return True # Contact deleted
        return False # Contact not found

manager = ContactManagerArray()
manager.add_contact("Alice", "123-456-7890")
manager.add_contact("Bob", "987-654-3210")
print(manager.search_contact("Alice")) # Output: {'name': 'Alice', 'phone': '123-456-7890'}
print(manager.delete_contact("Bob")) # Output: True
print(manager.search_contact("Bob")) # Output: None

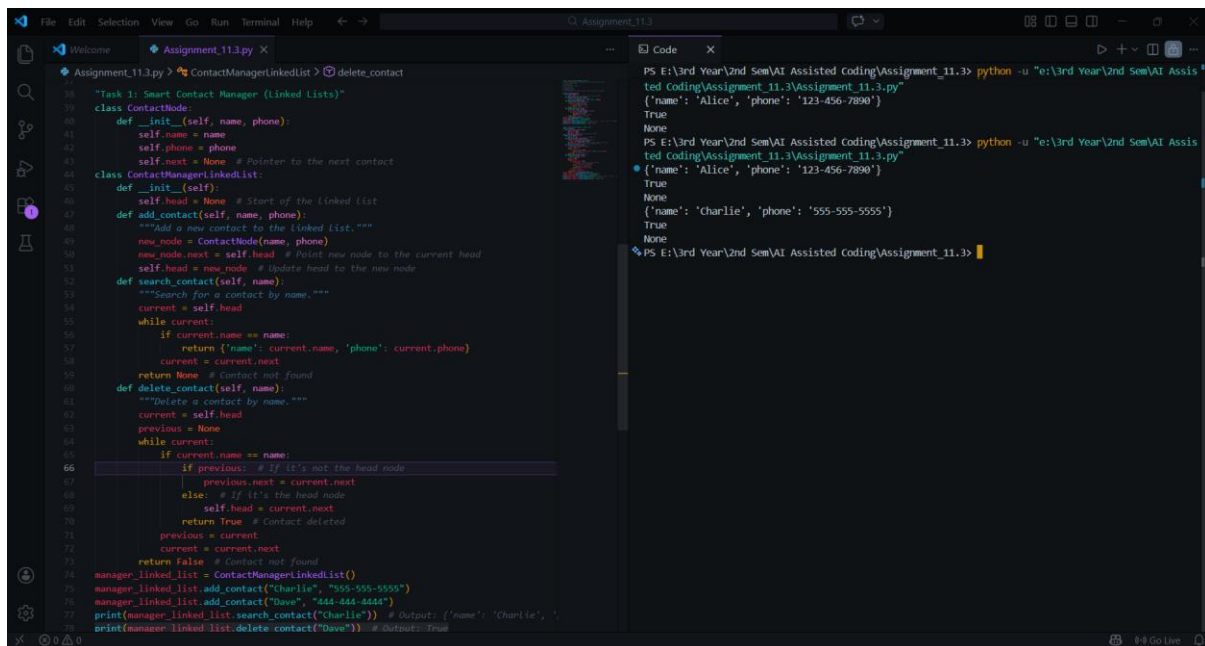
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3\Assignment_11.3.py"
{'name': 'Alice', 'phone': '123-456-7890'}
True
None
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation (Arrays):

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):



```
Assignment_11.3.py X
Task 1: Smart Contact Manager (Linked Lists)
class ContactNode:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
        self.next = None # Pointer to the next contact
class ContactManagerLinkedList:
    def __init__(self):
        self.head = None # Start of the linked list
    def add_contact(self, name, phone):
        """Add a new contact to the linked list."""
        new_node = ContactNode(name, phone)
        new_node.next = self.head # Point new node to the current head
        self.head = new_node # Update head to the new node
    def search_contact(self, name):
        """Search for a contact by name."""
        current = self.head
        while current:
            if current.name == name:
                return {'name': current.name, 'phone': current.phone}
            current = current.next
        return None # Contact not found
    def delete_contact(self, name):
        """Delete a contact by name."""
        current = self.head
        previous = None
        while current:
            if current.name == name:
                if previous: # If it's not the head node
                    previous.next = current.next
                else: # If it's the head node
                    self.head = current.next
                return True # Contact deleted
            previous = current
            current = current.next
        return False # Contact not found
manager_linked_list = ContactManagerLinkedList()
manager_linked_list.add_contact("Charlie", "555-555-5555")
manager_linked_list.add_contact("Dave", "444-444-4444")
print(manager_linked_list.search_contact("Charlie")) # Output: {'name': 'Charlie', 'phone': '555-555-5555'}
print(manager_linked_list.delete_contact("Dave")) # Output: True

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"
{'name': 'Alice', 'phone': '123-456-7890'}
None
{'name': 'Charlie', 'phone': '555-555-5555'}
True
None
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

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.

Comparison (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 image shows a code editor with two panes. The left pane displays a Python script for a library book search system using a queue. The right pane shows the output of the script.

Left Pane (Code):

```
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 st
86 # Include enqueue and dequeue methods.
87 from collections import deque
88 class Queue:
89     def __init__(self):
90         self.queue = deque() # Using deque for efficien
91     def enqueue(self, item):
92         '''Add an item to the end of the queue.'''
93         self.queue.append(item)
94     def dequeue(self):
95         '''Remove and return the item at the front of th
96         if not self.is_empty():
97             return self.queue.popleft()
98         return None # Queue is empty
99     def is_empty(self):
100         '''Check if the queue is empty.'''
101         return len(self.queue) == 0
102 #Example Usage
103 library_queue = Queue()
104 library_queue.enqueue("Student Request: Book A")
105 library_queue.enqueue("Faculty Request: Book B")
106 print(library_queue.dequeue()) # Output: "Student Reque
107
108
```

Right Pane (Output):

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

```

File Edit Selection View Go Run Terminal Help
Assignment_11.3
Welcome Assignment_11.3.py x
Assignment_11.3.py > ...
112 "Task 2: Library Book Search System (Priority Queue)"
113 import heapq
114 class PriorityQueue:
115     def __init__(self):
116         self.queue = [] # Using a List to store the priority
                           queue
117     def enqueue(self, item, priority):
118         '''Add an item with a given priority to the queue.'''
119         heapq.heappush(self.queue, (priority, item))
120     def dequeue(self):
121         '''Remove and return the item with the highest priority
          (lowest number).'''
122         if not self.is_empty():
123             return heapq.heappop(self.queue)[1] # Return the
          item, not the priority
124         return None # Queue is empty
125     def is_empty(self):
126         '''Check if the priority queue is empty.'''
127         return len(self.queue) == 0
128 # Example Usage
129 priority_queue = PriorityQueue()
130 priority_queue.enqueue("Student Request: Book A", priority=2) #
          Lower priority
131 priority_queue.enqueue("Faculty Request: Book B", priority=1) #
          Higher priority
132 print(priority_queue.dequeue()) # Output: "Faculty Request:
          Book B"
Code x
\\3rd Year\\2nd Sem\\AI Assisted Coding\\Assignment_11.3\\Assignment_11.3.py"
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>
Ln 134, Col 1 Spaces: 4 UTF-8 CRLF Python 3.13.12 (Microsoft Store) F4 Go Live Prettier

```

Explanation (Priority Queue):

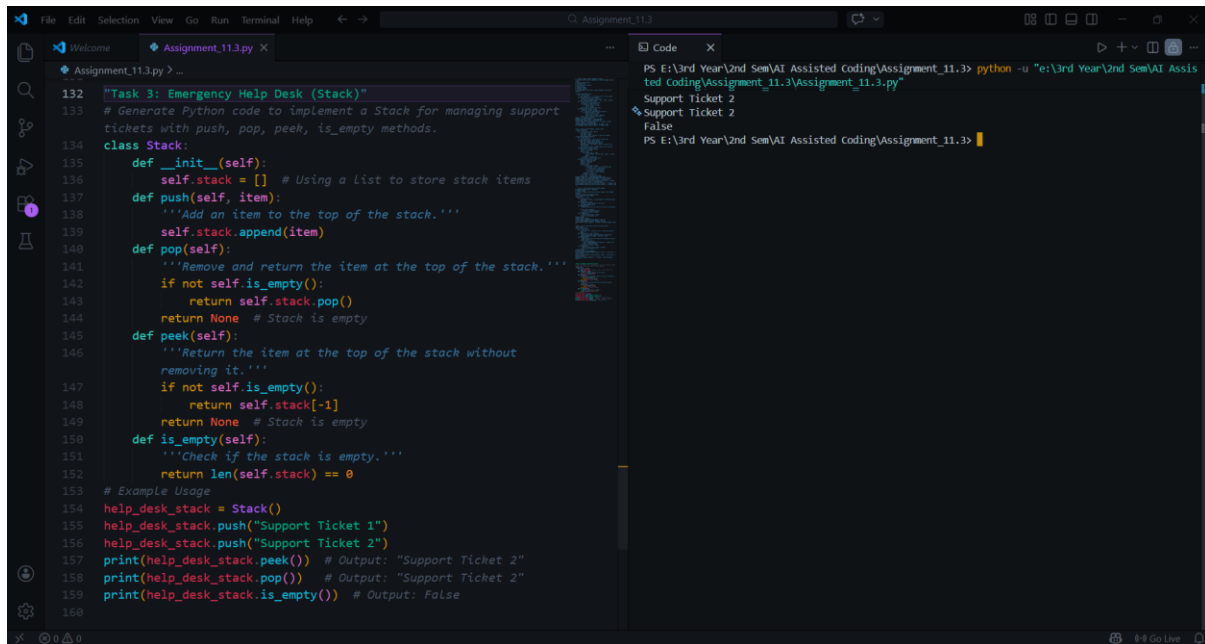
The priority queue uses a heap. Faculty requests are assigned higher priority (lower numeric value). This ensures faculty members are served before students.

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:



```
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 class Stack:
136     def __init__(self):
137         self.stack = [] # Using a list to store stack items
138     def push(self, item):
139         '''Add an item to the top of the stack.'''
140         self.stack.append(item)
141     def pop(self):
142         '''Remove and return the item at the top of the stack.'''
143         if not self.is_empty():
144             return self.stack.pop()
145         return None # Stack is empty
146     def peek(self):
147         '''Return the item at the top of the stack without
148         removing it.'''
149         if not self.is_empty():
150             return self.stack[-1]
151         return None # Stack is empty
152     def is_empty(self):
153         '''Check if the stack is empty.'''
154         return len(self.stack) == 0
155 # Example Usage
156 help_desk_stack = Stack()
157 help_desk_stack.push("Support Ticket 1")
158 help_desk_stack.push("Support Ticket 2")
159 print(help_desk_stack.peek()) # Output: "Support Ticket 2"
160 print(help_desk_stack.pop()) # Output: "Support Ticket 2"
161 print(help_desk_stack.is_empty()) # Output: False
```

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

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:

```
161 "Task 4: Hash Table"
162 # Generate a Python HashTable class with insert, search, and delete methods using collision handling through
    chaining.
163 class HashTable:
164     def __init__(self, size=10):
165         self.size = size
166         self.table = [[] for _ in range(size)] # Create a list of empty lists for chaining
167     def _hash(self, key):
168         """Generate a hash for the given key."""
169         return hash(key) % self.size
170     def insert(self, key, value):
171         """Insert a key-value pair into the hash table."""
172         index = self._hash(key)
173         # Check if the key already exists and update it
174         for i, (k, v) in enumerate(self.table[index]):
175             if k == key:
176                 self.table[index][i] = (key, value) # Update existing key
177                 return
178         # If key does not exist, add new key-value pair
179         self.table[index].append((key, value))
180     def search(self, key):
181         """Search for a value by its key."""
182         index = self._hash(key)
183         for k, v in self.table[index]:
184             if k == key:
185                 return v # Return the value associated with the key
186         return None # Key not found
187     def delete(self, key):
188         """Delete a key-value pair from the hash table."""
189         index = self._hash(key)
190         for i, (k, v) in enumerate(self.table[index]):
191             if k == key:
192                 del self.table[index][i] # Remove the key-value pair
193                 return True # Deletion successful
194         return False # Key not found
195 # Example Usage
196 hash_table = HashTable()
197 hash_table.insert("name", "Alice")
198 hash_table.insert("age", 20)
199 print(hash_table.search("name")) # Output: "Alice"
200 print(hash_table.delete("age")) # Output: True
201 print(hash_table.search("age")) # Output: None
202
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python 11.3.py
Alice
True
None
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation:

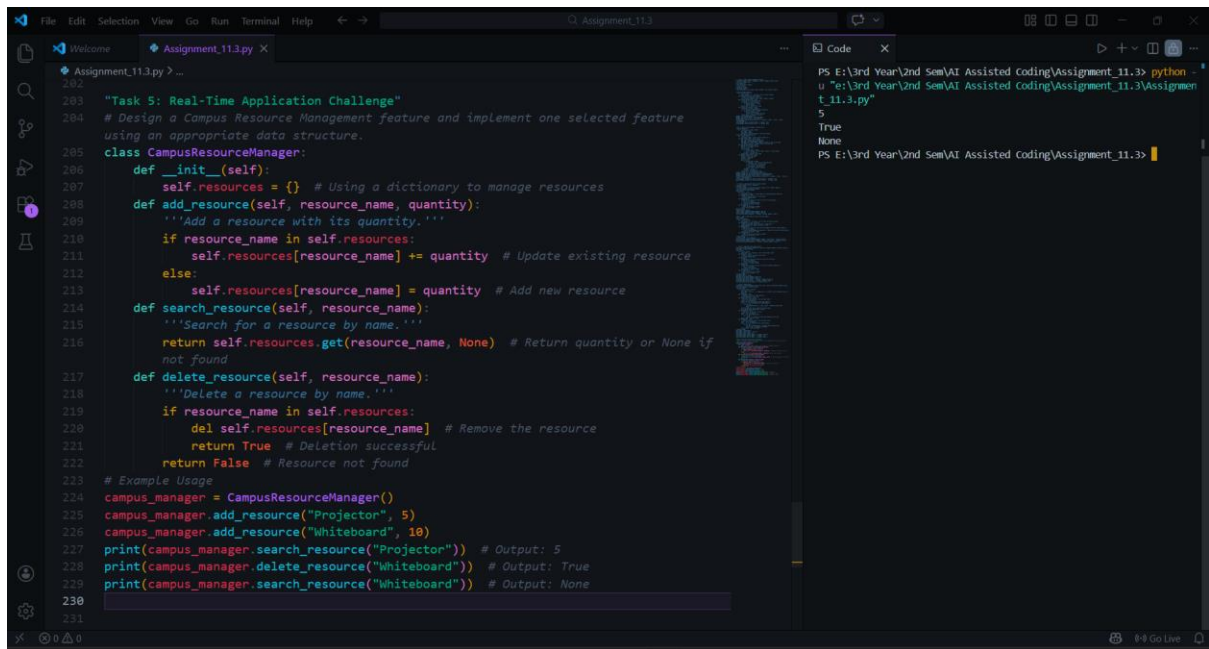
The hash table stores data using a hashing function to determine storage index. Collision handling is done using chaining, allowing multiple elements per bucket. This ensures efficient average-time operations. AI helped generate structured bucket management logic.

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:



```
202
203 "Task 5: Real-Time Application Challenge"
204 # Design a Campus Resource Management feature and implement one selected feature
205 # using an appropriate data structure.
206 class CampusResourceManager:
207     def __init__(self):
208         self.resources = {} # Using a dictionary to manage resources
209     def add_resource(self, resource_name, quantity):
210         '''Add a resource with its quantity.'''
211         if resource_name in self.resources:
212             self.resources[resource_name] += quantity # Update existing resource
213         else:
214             self.resources[resource_name] = quantity # Add new resource
215     def search_resource(self, resource_name):
216         '''Search for a resource by name.'''
217         return self.resources.get(resource_name, None) # Return quantity or None if
218         # not found
219     def delete_resource(self, resource_name):
220         '''Delete a resource by name.'''
221         if resource_name in self.resources:
222             del self.resources[resource_name] # Remove the resource
223             return True # Deletion successful
224         return False # Resource not found
225
226 # Example Usage
227 campus_manager = CampusResourceManager()
228 campus_manager.add_resource("Projector", 5)
229 campus_manager.add_resource("Whiteboard", 10)
230 print(campus_manager.search_resource("Projector")) # Output: 5
231 print(campus_manager.delete_resource("Whiteboard")) # Output: True
232 print(campus_manager.search_resource("Whiteboard")) # Output: None
233
234
```

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.

Final Conclusion:

This lab demonstrated implementation of fundamental data structures using AI assistance. Structures such as arrays, linked lists, stacks, queues, priority queues, and hash tables were explored in practical scenarios. AI tools improved code clarity and development speed. However, logical understanding and correct structure selection remain essential responsibilities of the developer.