Name: D. Naveen
Roll No: 2403A53L01
Batch - 46

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:**2025-2026
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s) Name	Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju		
CourseCode	23CS002PC304	**Course Title**	AI Assisted Coding
Year/Sem	III/II	**Regulation**	R23
Date and Day of Assignment	**Week6 – Monday**	**Time(s)**	23CSBTB01 To 23CSBTB52
Duration	2 Hours	**Applicable to Batches**	All batches
Assignment Number: 11.1(Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*Expected Time to complete*
1	**Lab 11 – Data Structures with AI: Implementing Fundamental Structures**	Week6 - Monday

	Lab Objectives • Use AI to assist in designing and implementing fundamental data structures in Python. • Learn how to prompt AI for structure creation, optimization, and documentation. • Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, Graphs, and Hash Tables. • Enhance code quality with AI-generated comments and performance suggestions.	
	Task Description #1 – Stack Implementation Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods. Sample Input Code: class Stack: pass Expected Output: • A functional stack implementation with all required methods and docstrings. **Prompt:** Generate a Python Stack class using a list with push, pop, peek, and is_empty methods, ensuring correct functionality and safe empty-stack handling. **Code:**	

```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
        print(f"Pushed: {item}")

    def pop(self):
        if self.is_empty():
            print("Error: Cannot pop from an empty stack")
            return None
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            print("Error: Stack is empty")
            return None
        return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

    def display(self):
        if self.is_empty():
            print("Stack is empty")
        else:
            print(f"Stack contents: {self.items}")


def main():
    stack = Stack()

    while True:
        print("\n--- Stack Operations ---")
        print("1. Push")
        print("2. Pop")
        print("3. Peek")
        print("4. Is Empty")
        print("5. Display Stack")
        print("6. Exit")

        choice = input("Enter your choice (1-6): ").strip()

        if choice == '1':
            value = input("Enter value to push: ").strip()
            stack.push(value)

        elif choice == '2':
            popped = stack.pop()
            if popped is not None:
                print(f"Popped: {popped}")

        elif choice == '3':
            peeked = stack.peek()
            if peeked is not None:
                print(f"Top element: {peeked}")

        elif choice == '4':
            if stack.is_empty():
                print("Stack is empty: True")
            else:
                print("Stack is empty: False")

        elif choice == '5':
            stack.display()

        elif choice == '6':
            print("Exiting...")
            break

        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()

```

Output:

```
Stack contents: ['2', '3']

--- Stack Operations ---
1. Push
2. Pop
3. Peek
4. Is Empty
5. Display Stack
6. Exit
Enter your choice (1-6): 6
Exiting...
PS C:\Users\Naveen\Desktop\AI Coding>
```

Task Description #2 – Queue Implementation

Task: Use AI to implement a Queue using Python lists.

Sample Input Code:

```
class Queue:
    pass
```

Expected Output:

- FIFO-based queue class with enqueue, dequeue, peek, and size methods.

Prompt:

Implement a FIFO-based Python Queue class using a list with enqueue, dequeue, peek, and size methods, where all operations dynamically take input from the user at runtime and handle empty queue cases safely.

```python
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)
        print(f"Enqueued: {item}")

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty. Cannot dequeue.")
            return None
        item = self.items.pop(0)
        print(f"Dequeued: {item}")
        return item

    def peek(self):
        if self.is_empty():
            print("Queue is empty. Cannot peek.")
            return None
        print(f"Front element: {self.items[0]}")
        return self.items[0]

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

    def is_empty(self):
        return len(self.items) == 0

    def display(self):
        if self.is_empty():
            print("Queue is empty.")
        else:
            print(f"Queue: {self.items}")


def main():
    queue = Queue()

    while True:
        print("\n--- Queue Operations ---")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Peek")
        print("4. Size")
        print("5. Display Queue")
        print("6. Exit")

        choice = input("Enter your choice (1-6): ").strip()

        if choice == '1':
            try:
                item = input("Enter item to enqueue: ").strip()
                queue.enqueue(item)
            except Exception as e:
                print(f"Error: {e}")

        elif choice == '2':
            queue.dequeue()

        elif choice == '3':
            queue.peek()

        elif choice == '4':
            print(f"Queue size: {queue.size()}")

        elif choice == '5':
            queue.display()

        elif choice == '6':
            print("Exiting...")
            break

        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()

```

Output:

```
Queue: ['4', '5']

--- Queue Operations ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Display Queue
6. Exit
Enter your choice (1-6): 6
Exiting...
PS C:\Users\Naveen\Desktop\AI Coding>
```

Task Description #3 – Linked List

Task: Use AI to generate a Singly Linked List with insert and display methods.

Sample Input Code:

```
class Node:
    pass

class LinkedList:
    pass
```

Expected Output:

- A working linked list implementation with clear method documentation.

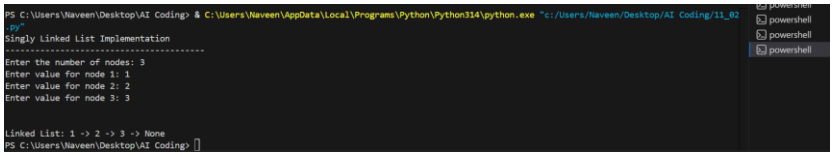
Prompt:

Generate a Python implementation of a Singly Linked List using Node and LinkedList classes, with insert and display methods where node values are dynamically taken from the user at runtime, ensuring correct linked list behavior and clear method documentation.

Code:

```python
class Node:
    """
    Represents a single node in the linked list.

    Attributes:
        data: The value stored in the node
        next: Reference to the next node in the linked list
    """
    def __init__(self, data):
        """
        Initialize a node with the given data.

        Args:
            data: The value to store in the node
        """
        self.data = data
        self.next = None


class LinkedList:
    """
    Implements a singly linked list with insertion and display functionality.

    Attributes:
        head: Reference to the first node in the linked list
    """
    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None

    def insert(self, data):
        """
        Insert a new node with the given data at the end of the linked list.

        Args:
            data: The value to insert into the linked list
        """
        new_node = Node(data)

        # If the list is empty, the new node becomes the head
        if self.head is None:
            self.head = new_node
            return

        # Traverse to the last node
        current = self.head
        while current.next is not None:
            current = current.next

        # Insert the new node at the end
        current.next = new_node

    def display(self):
        """
        Display all nodes in the linked list in order.
        Prints the linked list from head to tail.
        """
        if self.head is None:
            print("Linked List is empty.")
            return

        print("Linked List: ", end="")
        current = self.head
        while current is not None:
            print(current.data, end=" -> ")
            current = current.next
        print("None")


# Main program
if __name__ == "__main__":
    linked_list = LinkedList()

    print("Singly Linked List Implementation")
    print("-" * 40)

    # Take input from user
    try:
        n = int(input("Enter the number of nodes: "))

        if n <= 0:
            print("Please enter a positive number.")
        else:
            for i in range(n):
                value = input(f"Enter value for node {i + 1}: ")
                linked_list.insert(value)

            # Display the linked list
            print("\n")
            linked_list.display()

    except ValueError:
        print("Invalid input. Please enter valid data.")

```

Output:



Explanation:

This task implements a singly linked
list using Node and LinkedList classes.
The program dynamically accepts user input to insert elements into the list.
The display method prints all nodes in sequence, maintaining proper links between them.

Task Description #4 – Binary Search Tree (BST)

Task: Use AI to create a BST with insert and in-order traversal methods.
Sample Input Code:

```
class BST:
    pass
```

Expected Output:

- BST implementation with recursive insert and traversal methods.

Prompt:

Generate a Python Binary Search Tree (BST) implementation using
a BST class with recursive insert and in-order traversal methods.
The tree should dynamically accept input from the user at runtime for node insertion and correctly display elements using in-order traversal.

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class BST:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = Node(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = Node(value)
            else:
                self._insert_recursive(node.left, value)
        elif value > node.value:
            if node.right is None:
                node.right = Node(value)
            else:
                self._insert_recursive(node.right, value)

    def inorder_traversal(self):
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        if node is not None:
            self._inorder_recursive(node.left, result)
            result.append(node.value)
            self._inorder_recursive(node.right, result)

    def display(self):
        traversal = self.inorder_traversal()
        if traversal:
            print("In-order Traversal:", traversal)
        else:
            print("Tree is empty")


def main():
    bst = BST()
    print("Binary Search Tree (BST) Implementation")
    print("=" * 40)

    while True:
        print("\nOptions:")
        print("1. Insert a value")
        print("2. Display in-order traversal")
        print("3. Exit")

        choice = input("Enter your choice (1/2/3): ").strip()

        if choice == '1':
            try:
                value = int(input("Enter the value to insert: "))
                bst.insert(value)
                print(f"Value {value} inserted successfully!")
            except ValueError:
                print("Invalid input! Please enter an integer.")

        elif choice == '2':
            bst.display()

        elif choice == '3':
            print("Exiting the program. Goodbye!")
            break

        else:
            print("Invalid choice! Please enter 1, 2, or 3.")


if __name__ == "__main__":
    main()

```

Output:

Explanation:

This task implements a Binary Search Tree (BST) where nodes are inserted recursively based on BST rules.
User inputs are taken dynamically to build the tree at runtime.
An in-order traversal is used to display the elements in sorted order.

Task Description #5 – Hash Table

Task: Use AI to implement a hash table with basic insert, search, and delete methods.
Sample Input Code:

```
class HashTable:
    pass
```

Expected Output:

- Collision handling using chaining, with well-commented methods.

Prompt:

Generate a Python Hash Table implementation using a HashTable class with insert, search, and delete methods, where all keys and values are dynamically taken from the user at runtime, and collision handling is done using chaining.

Code:

```python
class HashTable:
    def __init__(self, size=10):
        """Initialize hash table with given size using chaining for collision handling"""
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        """Generate hash value for a given key"""
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert key-value pair into hash table"""
        index = self._hash(key)

        # Check if key already exists and update it
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                print(f"Updated: {key} -> {value}")
                return

        # If key doesn't exist, append new key-value pair
        self.table[index].append((key, value))
        print(f"Inserted: {key} -> {value}")

    def search(self, key):
        """Search for a key in hash table"""
        index = self._hash(key)

        for k, v in self.table[index]:
            if k == key:
                print(f"Found: {key} -> {v}")
                return v

        print(f"Key '{key}' not found")
        return None

    def delete(self, key):
        """Delete a key from hash table"""
        index = self._hash(key)

        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index].pop(i)
                print(f"Deleted: {key}")
                return True

        print(f"Key '{key}' not found for deletion")
        return False

    def display(self):
        """Display all key-value pairs in hash table"""
        print("\n--- Hash Table Contents ---")
        for i, chain in enumerate(self.table):
            print(f"Index {i}: {chain}")
        print("---------------------------\n")


# Main program
if __name__ == "__main__":
    size = int(input("Enter hash table size: "))
    ht = HashTable(size)

    while True:
        print("\n1. Insert")
        print("2. Search")
        print("3. Delete")
        print("4. Display")
        print("5. Exit")

        choice = input("Enter your choice (1-5): ").strip()

        if choice == '1':
            key = input("Enter key: ").strip()
            value = input("Enter value: ").strip()
            ht.insert(key, value)

        elif choice == '2':
            key = input("Enter key to search: ").strip()
            ht.search(key)

        elif choice == '3':
            key = input("Enter key to delete: ").strip()
            ht.delete(key)

        elif choice == '4':
            ht.display()

        elif choice == '5':
            print("Exiting program...")
            break

        else:
            print("Invalid choice! Please try again.")

```

Output:

Explanation:

This task implements a hash table that stores key–value pairs using chaining to handle collisions.

All insert, search, and delete operations take input dynamically from the user.

The implementation ensures efficient data access while handling collisions correctly.

Task Description #6 – Graph Representation

Task: Use AI to implement a graph using an adjacency list.

Sample Input Code:

```
class Graph:
    pass
```

Expected Output:

- Graph with methods to add vertices, add edges, and display connections.

Prompt:

Generate a Python Graph implementation using an adjacency list in a Graph class.

Implement methods to add vertices, add edges, and display graph connections, where all vertices and edges are dynamically taken from the user at runtime, ensuring correct representation of the graph

Code:

```python
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []
            print(f"Vertex {vertex} added successfully.")
        else:
            print(f"Vertex {vertex} already exists.")

    def add_edge(self, u, v):
        if u not in self.graph:
            print(f"Vertex {u} does not exist. Add it first.")
            return
        if v not in self.graph:
            print(f"Vertex {v} does not exist. Add it first.")
            return

        if v not in self.graph[u]:
            self.graph[u].append(v)
            print(f"Edge from {u} to {v} added successfully.")
        else:
            print(f"Edge from {u} to {v} already exists.")

    def display_graph(self):
        if not self.graph:
            print("Graph is empty.")
            return

        print("\nGraph Connections:")
        for vertex in self.graph:
            if self.graph[vertex]:
                print(f"{vertex} -> {', '.join(map(str, self.graph[vertex]))}")
            else:
                print(f"{vertex} -> (no connections)")


def main():
    g = Graph()

    while True:
        print("\n=== Graph Operations ===")
        print("1. Add Vertex")
        print("2. Add Edge")
        print("3. Display Graph")
        print("4. Exit")

        choice = input("Enter your choice (1/2/3/4): ").strip()

        if choice == '1':
            vertex = input("Enter vertex to add: ").strip()
            g.add_vertex(vertex)

        elif choice == '2':
            u = input("Enter source vertex: ").strip()
            v = input("Enter destination vertex: ").strip()
            g.add_edge(u, v)

        elif choice == '3':
            g.display_graph()

        elif choice == '4':
            print("Exiting...")
            break

        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()

```

Output:

Explanation:

This task implements a graph using an adjacency list representation.
Vertices and edges are added dynamically based on user input.
The display method shows all connections between vertices clearly.

Task Description #7 – Priority Queue

Task: Use AI to implement a priority queue using Python's heapq module.

Sample Input Code:

```
class PriorityQueue:
    pass
```

Expected Output:

- Implementation with enqueue (priority), dequeue (highest priority), and display methods.

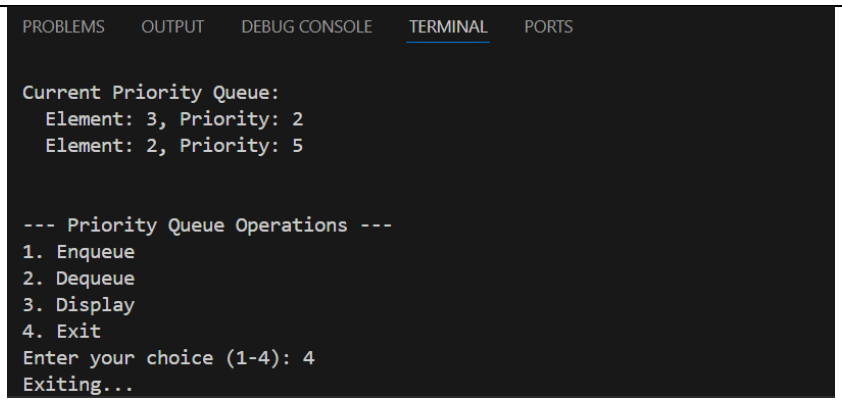
Prompt:

Generate a Python Priority Queue implementation using the heapq module in a PriorityQueue class.
Implement enqueue (with priority), dequeue (highest priority), and display methods, where all elements and priorities are dynamically taken from the user at runtime, ensuring correct priority-based ordering.

Code:

```python
import heapq

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

    def enqueue(self, element, priority):
        """Add an element with a given priority (lower number = higher priority)"""
        heapq.heappush(self.heap, (priority, element))
        print(f"Enqueued: {element} with priority {priority}")

    def dequeue(self):
        """Remove and return the element with highest priority"""
        if not self.heap:
            print("Priority Queue is empty!")
            return None
        priority, element = heapq.heappop(self.heap)
        print(f"Dequeued: {element} with priority {priority}")
        return element

    def display(self):
        """Display all elements in the priority queue"""
        if not self.heap:
            print("Priority Queue is empty!")
            return
        print("\nCurrent Priority Queue:")
        sorted_heap = sorted(self.heap)
        for priority, element in sorted_heap:
            print(f"  Element: {element}, Priority: {priority}")
        print()

def main():
    pq = PriorityQueue()

    while True:
        print("\n--- Priority Queue Operations ---")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Display")
        print("4. Exit")

        choice = input("Enter your choice (1-4): ").strip()

        if choice == '1':
            element = input("Enter element: ").strip()
            try:
                priority = int(input("Enter priority (lower number = higher priority): ").strip())
                pq.enqueue(element, priority)
            except ValueError:
                print("Invalid priority! Please enter an integer.")

        elif choice == '2':
            pq.dequeue()

        elif choice == '3':
            pq.display()

        elif choice == '4':
            print("Exiting...")
            break

        else:
            print("Invalid choice! Please try again.")

if __name__ == "__main__":
    main()

```

Output:



Explanation:

This task implements a priority queue using Python's heapq module. Elements along with their priorities are taken dynamically from the user. The queue always removes the element with the highest priority first.

Task Description #8 – Deque

Task: Use AI to implement a double-ended queue using collections.deque.

Sample Input Code:

```
class DequeDS:
    pass
```

Expected Output:

- Insert and remove from both ends with docstrings.

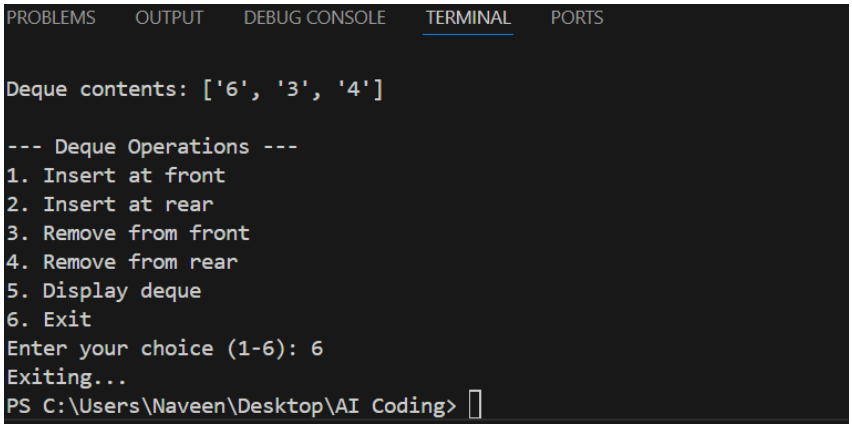
Prompt:

Generate a Python double-ended queue (Deque) implementation using collections.deque in a DequeDS class.
Implement methods to insert and remove elements from both the front and rear, where all inputs are dynamically taken from the user at runtime, and include clear docstrings for each method.

Code:

```python
from collections import deque


class DequeDS:
    """
    A double-ended queue (Deque) implementation using collections.deque.
    Supports insertion and removal from both front and rear ends.
    """

    def __init__(self):
        """Initialize an empty deque."""
        self.deque = deque()

    def insert_front(self, value):
        """
        Insert an element at the front of the deque.

        Args:
            value: The element to insert at the front.
        """
        self.deque.appendleft(value)
        print(f"Inserted {value} at the front.")

    def insert_rear(self, value):
        """
        Insert an element at the rear of the deque.

        Args:
            value: The element to insert at the rear.
        """
        self.deque.append(value)
        print(f"Inserted {value} at the rear.")

    def remove_front(self):
        """
        Remove and return an element from the front of the deque.

        Returns:
            The element removed from the front, or None if deque is empty.
        """
        if self.is_empty():
            print("Deque is empty. Cannot remove from front.")
            return None
        value = self.deque.popleft()
        print(f"Removed {value} from the front.")
        return value

    def remove_rear(self):
        """
        Remove and return an element from the rear of the deque.

        Returns:
            The element removed from the rear, or None if deque is empty.
        """
        if self.is_empty():
            print("Deque is empty. Cannot remove from rear.")
            return None
        value = self.deque.pop()
        print(f"Removed {value} from the rear.")
        return value

    def is_empty(self):
        """
        Check if the deque is empty.

        Returns:
            True if deque is empty, False otherwise.
        """
        return len(self.deque) == 0

    def display(self):
        """Display all elements in the deque."""
        if self.is_empty():
            print("Deque is empty.")
        else:
            print(f"Deque contents: {list(self.deque)}")


def main():
    """Main function to interact with the DequeDS."""
    deque_ds = DequeDS()

    while True:
        print("\n--- Deque Operations ---")
        print("1. Insert at front")
        print("2. Insert at rear")
        print("3. Remove from front")
        print("4. Remove from rear")
        print("5. Display deque")
        print("6. Exit")

        choice = input("Enter your choice (1-6): ").strip()

        if choice == '1':
            value = input("Enter value to insert at front: ").strip()
            deque_ds.insert_front(value)

        elif choice == '2':
            value = input("Enter value to insert at rear: ").strip()
            deque_ds.insert_rear(value)

        elif choice == '3':
            deque_ds.remove_front()

        elif choice == '4':
            deque_ds.remove_rear()

        elif choice == '5':
            deque_ds.display()

        elif choice == '6':
            print("Exiting...")
            break

        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()

```

Output:



Explanation:

This task implements a double-ended queue (deque) using Python's collections.deque.

Elements are inserted and removed from both ends based on dynamic user input.

The implementation allows efficient operations at both the front and rear of the queue.

Task Description #9 Real-Time Application Challenge – Choose the Right Data Structure

Scenario:

Your college wants to develop a Campus Resource Management System that handles:

1. Student Attendance Tracking – Daily log of students entering/exiting the campus.
2. Event Registration System – Manage participants in events with quick search and removal.
3. Library Book Borrowing – Keep track of available books and their due dates.
4. Bus Scheduling System – Maintain bus routes and stop connections.
5. Cafeteria Order Queue – Serve students in the order they arrive.

Student Task:

- For each feature, select the most appropriate data structure from the list below:
    - Stack
    - Queue
    - Priority Queue
    - Linked List
    - Binary Search Tree (BST)
    - Graph
    - Hash Table

- Deque

- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

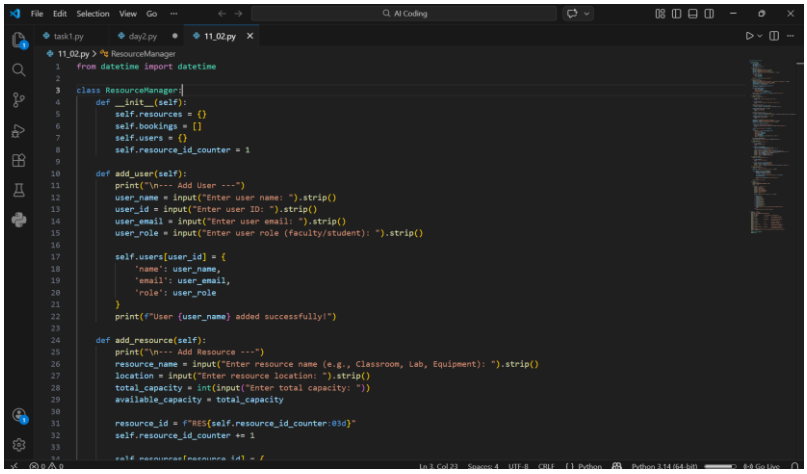
Expected Output:

- A table mapping feature → chosen data structure → justification.
- A functional Python program implementing the chosen feature with comments and docstrings.

Prompt:

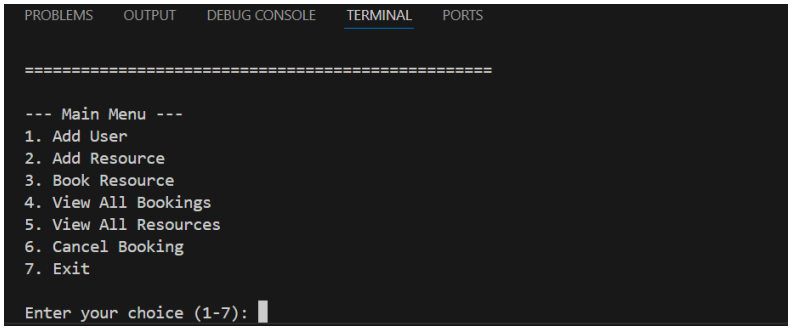
Select the most suitable data structure for each feature of a Campus Resource Management System, justify each choice in 2–3 sentences in a table, and implement one selected feature as a Python program that dynamically takes user input with comments and docstrings.

Code:



```
from datetime import datetime

class ResourceManager:
    def __init__(self):
        self.resources = {}
        self.bookings = []
        self.users = {}
        self.resource_id_counter = 1

    def add_user(self):
        print("\n--- Add User ---")
        user_name = input("Enter user name: ").strip()
        user_id = input("Enter user ID: ").strip()
        user_email = input("Enter user email: ").strip()
        user_role = input("Enter user role (faculty/student): ").strip()

        self.users[user_id] = {
            'name': user_name,
            'email': user_email,
            'role': user_role
        }
        print(f"User {user_name} added successfully!")

    def add_resource(self):
        print("\n--- Add Resource ---")
        resource_name = input("Enter resource name (e.g., Classroom, Lab, Equipment): ").strip()
        location = input("Enter resource location: ").strip()
        total_capacity = int(input("Enter total capacity: "))
        available_capacity = total_capacity

        resource_id = f"RES{self.resource_id_counter:03d}"
        self.resource_id_counter += 1
```

Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

==================================================

--- Main Menu ---
1. Add User
2. Add Resource
3. Book Resource
4. View All Bookings
5. View All Resources
6. Cancel Booking
7. Exit

Enter your choice (1-7):
```

Explanation:

This task analyzes real-time campus system features and maps each one to the most suitable data structure.

Each choice is justified based on efficiency and use-case requirements.

One feature is implemented as a dynamic Python program using user input to demonstrate practical application.

Task Description #10: Smart E-Commerce Platform – Data Structure Challenge

An e-commerce company wants to build a Smart Online Shopping System with:

1. Shopping Cart Management – Add and remove products dynamically.
2. Order Processing System – Orders processed in the order they are placed.
3. Top-Selling Products Tracker – Products ranked by sales count.
4. Product Search Engine – Fast lookup of products using product ID.
5. Delivery Route Planning – Connect warehouses and delivery locations.

Student Task:

- For each feature, select the most appropriate data structure from the list below:
  - Stack
  - Queue
  - Priority Queue
  - Linked List
  - Binary Search Tree (BST)
  - Graph
  - Hash Table
  - Deque
- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

Expected Output:

- A table mapping feature → chosen data structure → justification.
- A functional Python program implementing the chosen feature with comments and docstrings.

Prompt:

Map each e-commerce feature to the most suitable data structure with 2–3 line justifications in a table, and implement one chosen feature as a Python program that dynamically takes user input with comments and docstrings.

Code:

```
11_02.py > ShoppingCart > update_quantity
    "Wishlist": "Set - No duplicates, fast add/remove operations.",
    "Product Categories": "Tree/Graph - Hierarchical relationships, category-subcategory mapping.",
    "Search Index": "Hash Table - O(1) search by product name or keyword.",
    "Order History": "Stack/List - Recent orders accessed first, maintains chronological order."
}

print("E-Commerce Features & Data Structures Mapping:\n")
for feature, structure in e_commerce_features.items():
    print(f"{feature}: {structure}\n")

print("\n" + "="*60)
print("SHOPPING CART IMPLEMENTATION")
print("="*60 + "\n")

class ShoppingCart:

    def __init__(self, user_id):
        self.user_id = user_id
        self.cart = []

    def add_item(self, product_id, product_name, price, quantity):
        for item in self.cart:
            if item['product_id'] == product_id:
                item['quantity'] += quantity
                print(f"Updated {product_name} quantity to {item['quantity']}")
                return

        self.cart.append({
            'product_id': product_id,
            'product_name': product_name,
            'price': price,
            'quantity': quantity
        })
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Enter User ID: 23

1. Add Item  2. Remove Item  3. Update Qty  4. View Cart  5. Checkout  6. Exit
Select option: 1
Enter Product ID: 234
Enter Product Name: book
Enter Price: 200
Enter Quantity: 3
Added book (x3) to cart

1. Add Item  2. Remove Item  3. Update Qty  4. View Cart  5. Checkout  6. Exit
Select option:
```

Explanation:

This task selects appropriate data structures for different features of a smart e-commerce system.

Each choice is justified based on efficiency and real-time requirements.

One feature is implemented as a dynamic Python program to demonstrate practical usage.