SCHOOLOFCOMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENTOFCOMPUTER SCIENCE ENGINEERING	
ProgramName:B. Tech		**AssignmentType: Lab**	**AcademicYear:**2025-2026
CourseCoordinatorName		Venkataramana Veeramsetty	
Instructor(s)Name		Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)	
CourseCode	24CS002PC215	**CourseTitle**	AI Assisted Coding
Year/Sem	II/I	**Regulation**	R24
DateandDay of Assignment	Week6 - Monday	**Time(s)**	
Duration	2 Hours	**Applicableto Batches**	
AssignmentNumber:11.1(Presentassignmentnumber)/**24**(Totalnumberofassignments)			

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

- 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.

CODE:

```
class Stack:
    """
    A simple implementation of a Stack data structure using a Python list.
    
    The stack follows LIFO (Last In, First Out) principle where the last 
element
    added to the stack is the first one to be removed.
    
    Attributes:
        items (list): Internal list to store stack elements
    """
    
    def __init__(self):
        """
        Initialize an empty stack.
        
        Creates a new stack instance with no elements.
        """
        self.items = []
    
    def push(self, item):
        """
        Add an element to the top of the stack.
        
        Args:
            item: The element to be added to the stack
            
        Returns:
            None
        """
        self.items.append(item)
    
    def pop(self):
        """
        Remove and return the top element from the stack.
        
        Returns:
            The top element from the stack
            
        Raises:
```

```
            IndexError: If the stack is empty
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.items.pop()

    def peek(self):
        """
        Return the top element from the stack without removing it.

        Returns:
            The top element from the stack

        Raises:
            IndexError: If the stack is empty
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]

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

        Returns:
            bool: True if the stack is empty, False otherwise
        """
        return len(self.items) == 0

    def size(self):
        """
        Return the number of elements in the stack.

        Returns:
            int: The number of elements in the stack
        """
        return len(self.items)

    def __str__(self):
        """
        Return a string representation of the stack.

        Returns:
            str: String representation of the stack
        """
        return f"Stack({self.items})"

    def __repr__(self):
        """
        Return a detailed string representation of the stack.

        Returns:
            str: Detailed string representation of the stack
        """
        return f"Stack(items={self.items})"
```

```python
# Example usage
if __name__ == "__main__":
    # Create a new stack
    stack = Stack()

    # Test is_empty
    print(f"Is stack empty? {stack.is_empty()}")  # Should print: True

    # Test push
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(f"Stack after pushing 1, 2, 3: {stack}")

    # Test peek
    print(f"Top element (peek): {stack.peek()}")  # Should print: 3

    # Test size
    print(f"Stack size: {stack.size()}")  # Should print: 3

    # Test pop
    popped_item = stack.pop()
    print(f"Popped item: {popped_item}")  # Should print: 3
    print(f"Stack after pop: {stack}")

    # Test is_empty again
    print(f"Is stack empty? {stack.is_empty()}")  # Should print: False
```

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    P

l/Programs/Python/Python312/python.exe "c:/User
MENTS/stack implementation.py"
Is stack empty? True
Stack after pushing 1, 2, 3: Stack([1, 2, 3])
Top element (peek): 3
Stack size: 3
Popped item: 3
Stack after pop: Stack([1, 2])
Is stack empty? False
```

Task Description #2 – Queue Implementation

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

Sample Input Code:

class Queue:

pass

Expected Output:

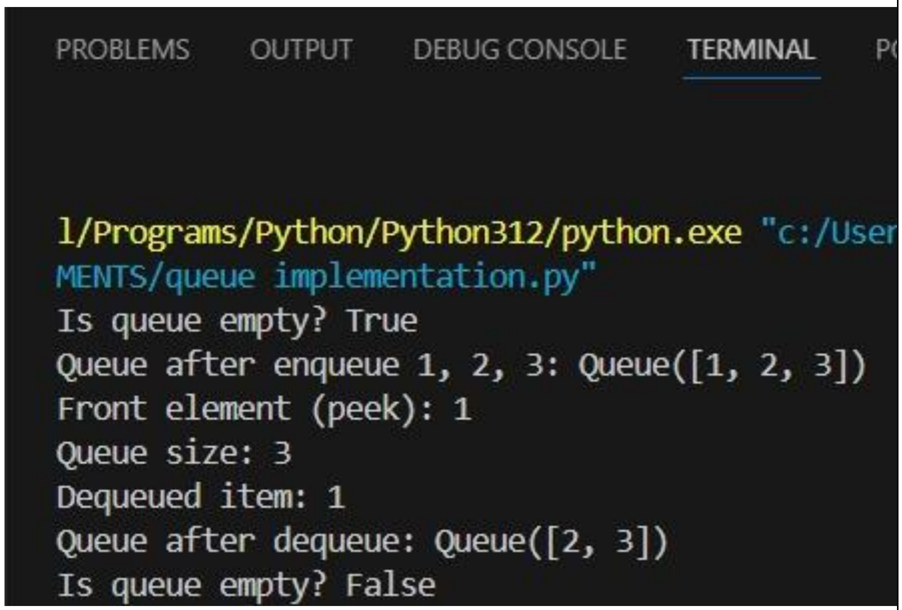
- FIFO-based queue class with enqueue, dequeue, peek, and size methods.

CODE:

```
class Queue:
    """
    A simple implementation of a Queue data structure using a Python list.

    The queue follows FIFO (First In, First Out) principle where the first
element
    added to the queue is the first one to be removed.

    Attributes:
        items (list): Internal list to store queue elements
    """

    def __init__(self):
        """
        Initialize an empty queue.
        """
        self.items = []

    def enqueue(self, item):
        """
        Add an element to the end of the queue.

        Args:
            item: The element to be added to the queue
        """
        self.items.append(item)

    def dequeue(self):
        """
        Remove and return the front element from the queue.

        Returns:
            The front element from the queue

        Raises:
            IndexError: If the queue is empty
        """
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.items.pop(0)

    def peek(self):
        """
        Return the front element from the queue without removing it.

        Returns:
            The front element from the queue

        Raises:
            IndexError: If the queue is empty
```

```python
        """
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.items[0]

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

        Returns:
            bool: True if the queue is empty, False otherwise
        """
        return len(self.items) == 0

    def size(self):
        """
        Return the number of elements in the queue.

        Returns:
            int: The number of elements in the queue
        """
        return len(self.items)

    def __str__(self):
        """
        Return a string representation of the queue.

        Returns:
            str: String representation of the queue
        """
        return f"Queue({self.items})"

    def __repr__(self):
        """
        Return a detailed string representation of the queue.

        Returns:
            str: Detailed string representation of the queue
        """
        return f"Queue(items={self.items})"

# Example usage
if __name__ == "__main__":
    queue = Queue()
    print(f"Is queue empty? {queue.is_empty()}")  # Should print: True
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print(f"Queue after enqueue 1, 2, 3: {queue}")
    print(f"Front element (peek): {queue.peek()}")  # Should print: 1
    print(f"Queue size: {queue.size()}")  # Should print: 3
    dequeued_item = queue.dequeue()
    print(f"Dequeued item: {dequeued_item}")  # Should print: 1
    print(f"Queue after dequeue: {queue}")
    print(f"Is queue empty? {queue.is_empty()}")  # Should print: False
```

OUTPUT:



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:

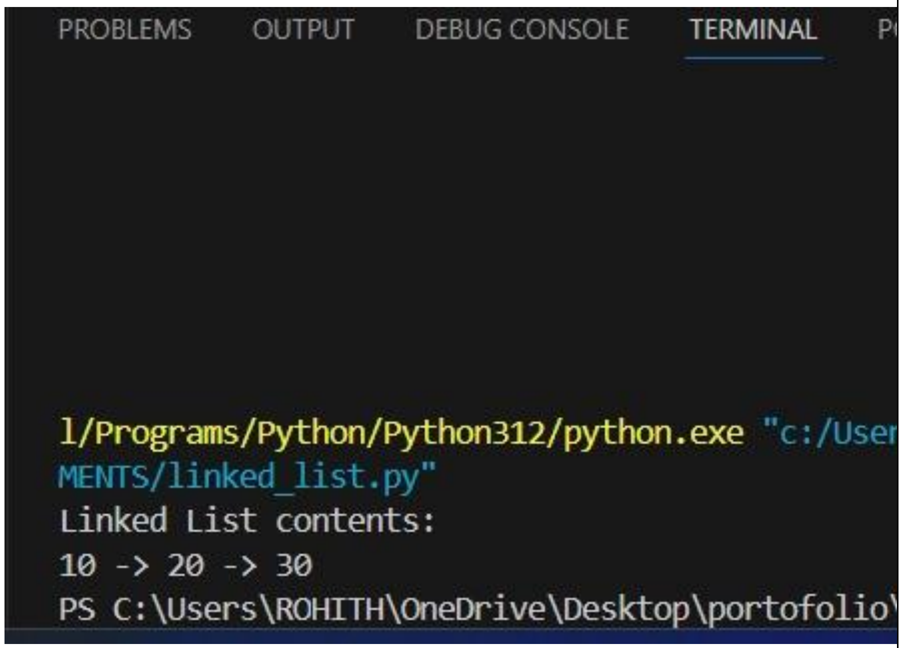
- A working linked list implementation with clear method documentation.

CODE:

```python
class Node:
    """
    Node for singly linked list.

    Attributes:
        data: The value stored in the node.
        next: Reference to the next node in the list.
    """
    def __init__(self, data):
        """
        Initialize a node with data and next pointer.
        Args:
            data: Value to store in the node.
        """
        self.data = data
        self.next = None
```

```python
class LinkedList:
    """
    Singly linked list implementation.

    Attributes:
        head: Reference to the first node in the 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 list.
        Args:
            data: Value to insert.
        """
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def display(self):
        """
        Display all elements in the linked list.
        Prints the data of each node in order.
        """
        current = self.head
        elements = []
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" -> ".join(elements))

# Example usage
if __name__ == "__main__":
    ll = LinkedList()
    ll.insert(10)
    ll.insert(20)
    ll.insert(30)
    print("Linked List contents:")
    ll.display()  # Output: 10 -> 20 -> 30
```

OUTPUT:



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:

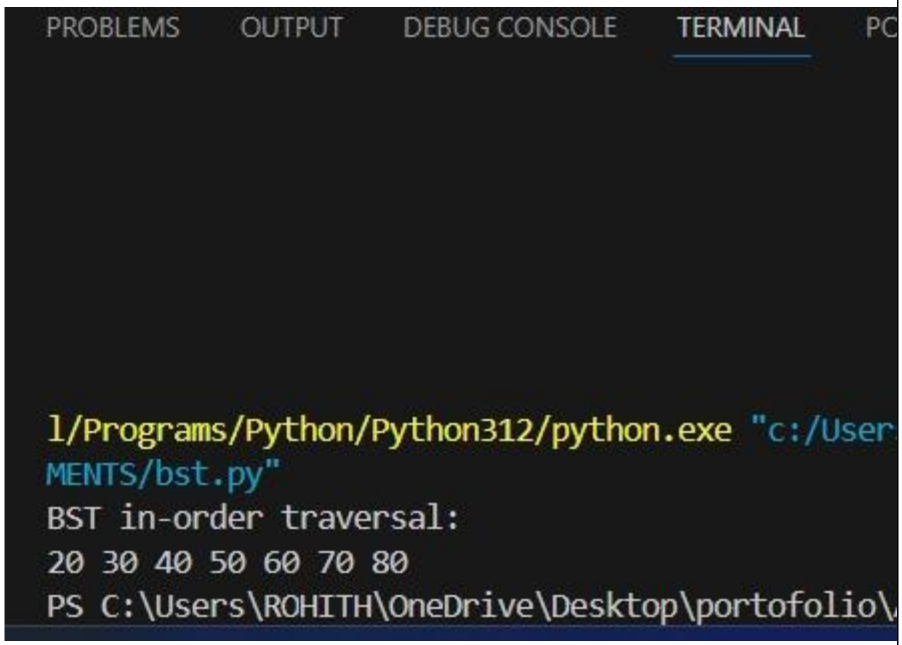
- BST implementation with recursive insert and traversal methods.

CODE:

```
class BSTNode:
    """
    Node for Binary Search Tree.

    Attributes:
        data: The value stored in the node.
        left: Reference to the left child node.
        right: Reference to the right child node.
    """
    def __init__(self, data):
        """
        Initialize a BST node with data and child pointers.
        Args:
            data: Value to store in the node.
        """
        self.data = data
        self.left = None
        self.right = None

class BST:
```

```
    """
    Binary Search Tree implementation.

    Attributes:
        root: Reference to the root node of the BST.
    """
    def __init__(self):
        """
        Initialize an empty BST.
        """
        self.root = None

    def insert(self, data):
        """
        Insert a new value into the BST (recursive).
        Args:
            data: Value to insert.
        """
        self.root = self._insert_recursive(self.root, data)

    def _insert_recursive(self, node, data):
        """
        Helper method to recursively insert a value.
        """
        if node is None:
            return BSTNode(data)
        if data < node.data:
            node.left = self._insert_recursive(node.left, data)
        elif data > node.data:
            node.right = self._insert_recursive(node.right, data)
        # If data == node.data, do not insert duplicates
        return node

    def in_order_traversal(self):
        """
        Perform in-order traversal and print values in sorted order.
        """
        self._in_order_recursive(self.root)
        print()

    def _in_order_recursive(self, node):
        """
        Helper method for in-order traversal.
        """
        if node:
            self._in_order_recursive(node.left)
            print(node.data, end=" ")
            self._in_order_recursive(node.right)

# Example usage
if __name__ == "__main__":
    bst = BST()
    for value in [50, 30, 70, 20, 40, 60, 80]:
        bst.insert(value)
    print("BST in-order traversal:")
    bst.in_order_traversal()  # Output: 20 30 40 50 60 70 80
```

OUTPUT:



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.

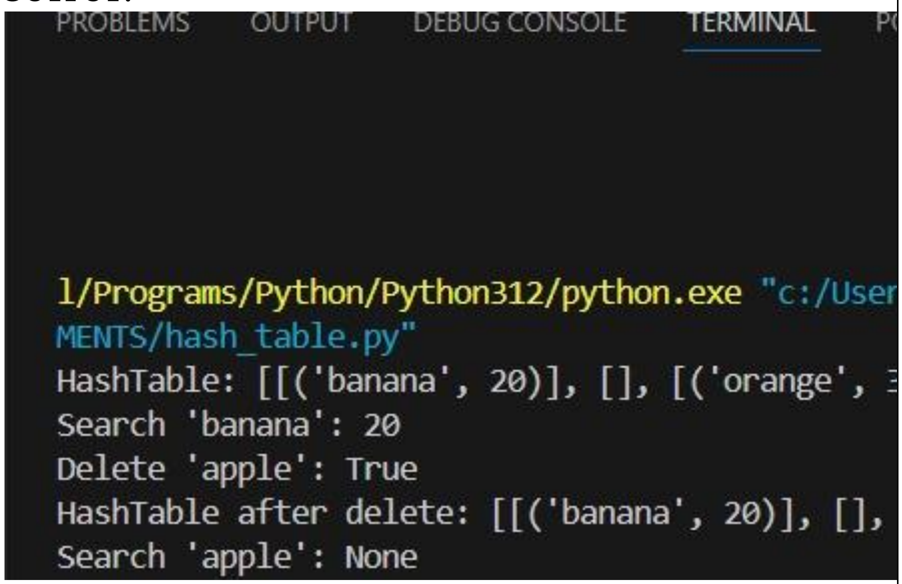
CODE:

```
class HashTable:
    """
    Hash table implementation using chaining for collision handling.

    Attributes:
        size (int): Number of buckets in the hash table.
        table (list): List of buckets, each a list for chaining.
    """
    def __init__(self, size=10):
        """
        Initialize the hash table with a given number of buckets.
        Args:
            size: Number of buckets (default 10).
        """
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        """
```

```python
        Compute the hash value for a given key.
        Args:
            key: The key to hash.
        Returns:
            int: The bucket index for the key.
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """
        Insert a key-value pair into the hash table.
        If the key already exists, update its value.
        Args:
            key: The key to insert.
            value: The value to associate with the key.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return
        bucket.append((key, value))

    def search(self, key):
        """
        Search for a key in the hash table and return its value if found.
        Args:
            key: The key to search for.
        Returns:
            The value associated with the key, or None if not found.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for k, v in bucket:
            if k == key:
                return v
        return None

    def delete(self, key):
        """
        Delete a key-value pair from the hash table if it exists.
        Args:
            key: The key to delete.
        Returns:
            bool: True if the key was found and deleted, False otherwise.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                return True
        return False

    def __str__(self):
```

```
        """
        Return a string representation of the hash table.
        """
        return str(self.table)

# Example usage
if __name__ == "__main__":
    ht = HashTable(size=5)
    ht.insert("apple", 10)
    ht.insert("banana", 20)
    ht.insert("orange", 30)
    print("HashTable:", ht)
    print("Search 'banana':", ht.search("banana"))  # Output: 20
    print("Delete 'apple':", ht.delete("apple"))   # Output: True
    print("HashTable after delete:", ht)
    print("Search 'apple':", ht.search("apple"))   # Output: None
```

OUTPUT:



```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PO

l/Programs/Python/Python312/python.exe "c:/User
MENTS/hash_table.py"
HashTable: [[('banana', 20)], [], [('orange', 3
Search 'banana': 20
Delete 'apple': True
HashTable after delete: [[('banana', 20)], [],
Search 'apple': None
```

Task Description #6 – Graph Representation

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

Sample Input Code:

class Graph:

    pass

Expected Output:

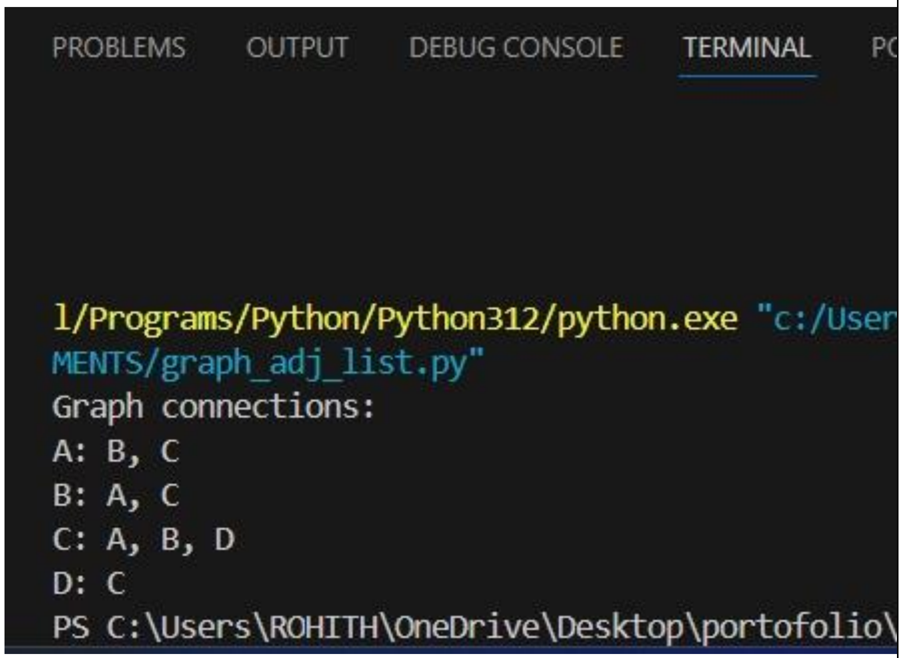
- Graph with methods to add vertices, add edges, and display connections.

CODE:

```
class Graph:
    """
    Graph implementation using an adjacency list.

    Attributes:
        adj_list (dict): Dictionary mapping vertices to lists of adjacent
```

```
vertices.
    """
    def __init__(self):
        """
        Initialize an empty graph.
        """
        self.adj_list = {}

    def add_vertex(self, vertex):
        """
        Add a vertex to the graph.
        Args:
            vertex: The vertex to add.
        """
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []

    def add_edge(self, v1, v2):
        """
        Add an edge between two vertices (undirected).
        Args:
            v1: First vertex.
            v2: Second vertex.
        """
        if v1 not in self.adj_list:
            self.add_vertex(v1)
        if v2 not in self.adj_list:
            self.add_vertex(v2)
        self.adj_list[v1].append(v2)
        self.adj_list[v2].append(v1)

    def display(self):
        """
        Display the graph's adjacency list.
        Prints each vertex and its connections.
        """
        for vertex, neighbors in self.adj_list.items():
            print(f"{vertex}: {', '.join(map(str, neighbors))}")

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_vertex('A')
    g.add_vertex('B')
    g.add_edge('A', 'B')
    g.add_edge('A', 'C')
    g.add_edge('B', 'C')
    g.add_edge('C', 'D')
    print("Graph connections:")
    g.display()
```

OUTPUT:



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:

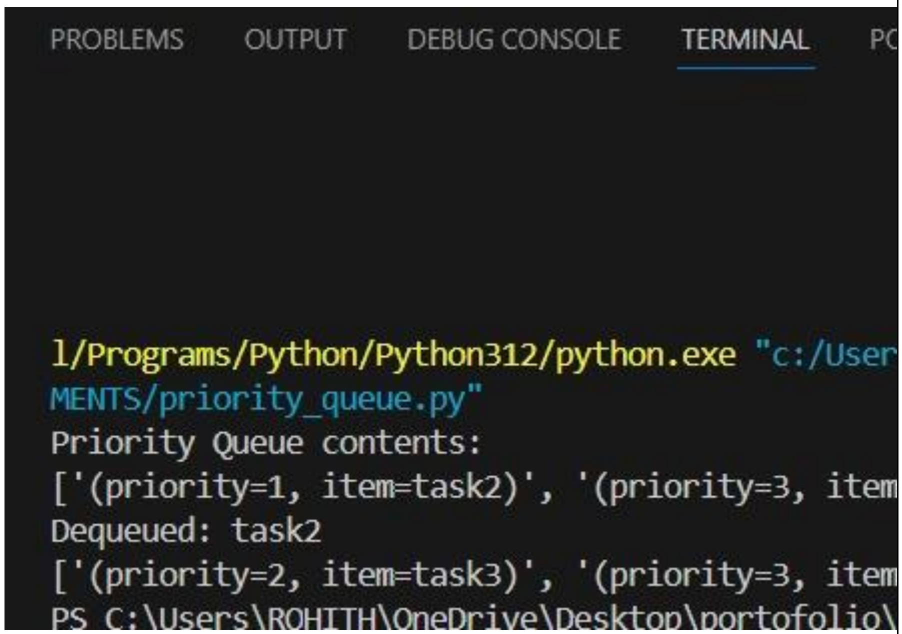
- Implementation with enqueue (priority), dequeue (highest priority), and display methods.

CODE:

```
import heapq

class PriorityQueue:
    """
    Priority Queue implementation using Python's heapq module.

    Attributes:
        heap (list): Internal list to store heap elements as (priority,
item) tuples.
    """
    def __init__(self):
        """
        Initialize an empty priority queue.
        """
        self.heap = []

    def enqueue(self, item, priority):
        """
```

```python
        Add an item to the queue with a given priority.
        Args:
            item: The item to add.
            priority: The priority value (lower means higher priority).
        """
        heapq.heappush(self.heap, (priority, item))

    def dequeue(self):
        """
        Remove and return the item with the highest priority (lowest
priority value).
        Returns:
            The item with the highest priority.
        Raises:
            IndexError: If the queue is empty.
        """
        if not self.heap:
            raise IndexError("dequeue from empty priority queue")
        return heapq.heappop(self.heap)[1]

    def display(self):
        """
        Display all items in the priority queue in heap order.
        Prints each item with its priority.
        """
        print([f"(priority={p}, item={i})" for p, i in self.heap])

    def is_empty(self):
        """
        Check if the priority queue is empty.
        Returns:
            bool: True if empty, False otherwise.
        """
        return len(self.heap) == 0

# Example usage
if __name__ == "__main__":
    pq = PriorityQueue()
    pq.enqueue("task1", 3)
    pq.enqueue("task2", 1)
    pq.enqueue("task3", 2)
    print("Priority Queue contents:")
    pq.display()
    print("Dequeued:", pq.dequeue())  # Should print: task2
    pq.display()
```

OUTPUT:



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.

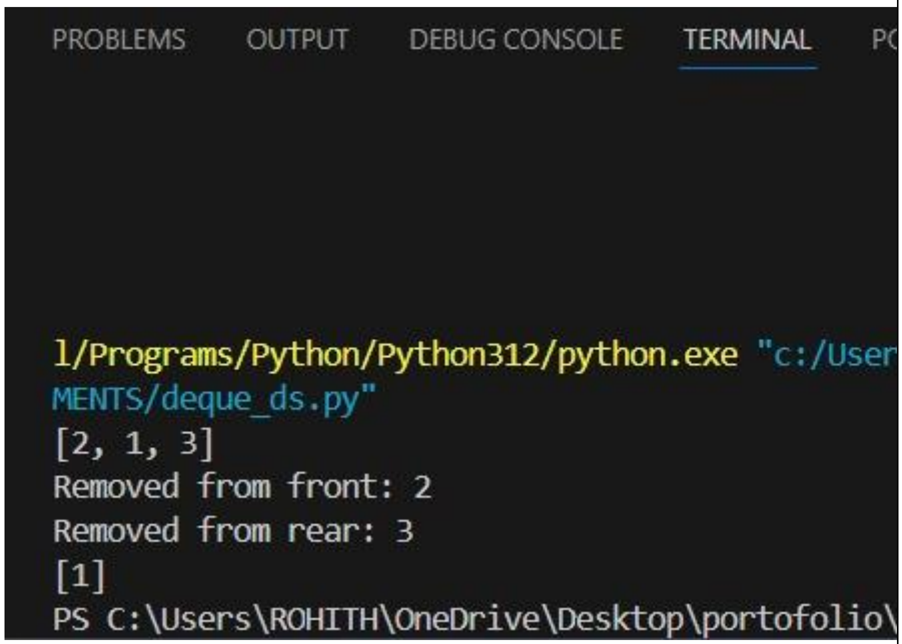
CODE:

```
from collections import deque

class DequeDS:
    """
    Double-ended queue (deque) implementation using collections.deque.

    Supports insertion and removal from both ends.
    """
    def __init__(self):
        """
        Initialize an empty deque.
        """
        self.deque = deque()

    def add_front(self, item):
        """
        Insert an item at the front of the deque.
        Args:
            item: The item to insert.
        """
```

```python
        self.deque.appendleft(item)

    def add_rear(self, item):
        """
        Insert an item at the rear of the deque.
        Args:
            item: The item to insert.
        """
        self.deque.append(item)

    def remove_front(self):
        """
        Remove and return the item from the front of the deque.
        Returns:
            The item removed from the front.
        Raises:
            IndexError: If the deque is empty.
        """
        if not self.deque:
            raise IndexError("remove_front from empty deque")
        return self.deque.popleft()

    def remove_rear(self):
        """
        Remove and return the item from the rear of the deque.
        Returns:
            The item removed from the rear.
        Raises:
            IndexError: If the deque is empty.
        """
        if not self.deque:
            raise IndexError("remove_rear from empty deque")
        return self.deque.pop()

    def display(self):
        """
        Display the contents of the deque.
        """
        print(list(self.deque))

    def is_empty(self):
        """
        Check if the deque is empty.
        Returns:
            bool: True if empty, False otherwise.
        """
        return len(self.deque) == 0

# Example usage
if __name__ == "__main__":
    dq = DequeDS()
    dq.add_rear(1)
    dq.add_front(2)
    dq.add_rear(3)
    dq.display()  # Output: [2, 1, 3]
    print("Removed from front:", dq.remove_front())  # Output: 2
```

```
print("Removed from rear:", dq.remove_rear())    # Output: 3
dq.display()  # Output: [1]
```

OUTPUT:



Task Description #9 – AI-Generated Data Structure Comparisons

Task: Use AI to generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.

Sample Input Code:

No code, prompt AI for a data structure comparison table

Expected Output:

- A markdown table with structure names, operations, and complexities.

COMPARISON TABLE:(OUTPUT)

Here is a markdown table comparing common data structures and their time complexities for basic operations:

Data Structure	Operations	Time Complexity (Average)	Time Complexity (Worst)
Stack	push, pop, peek	O(1)	O(1)
Queue	enqueue, dequeue	O(1)	O(1)
Linked List	insert,	O(1) (at	O(n)

	delete	head)	(search)
	search	O(n)	O(n)
Hash Table	insert, search	O(1)	O(n)
	delete	O(1)	O(n)
BST (Balanced)	insert, search	O(log n)	O(log n)
	delete	O(log n)	O(log n)
BST (Unbalanced)	insert, search	O(log n)	O(n)
	delete	O(log n)	O(n)
Graph (Adj List)	add vertex/edge	O(1)	O(1)
	search neighbor	O(k) (k = degree)	O(k)
Priority Queue	enqueue	O(log n)	O(log n)
	dequeue	O(log n)	O(log n)
Deque	add/remove ends	O(1)	O(1)

Task Description #10 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.

☑ Deliverables (For All Tasks)

1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.

Here is a table mapping each feature to the most appropriate data structure, with justification:(OUTPUT)

Feature	Data Structure	Justification
Student Attendance Tracking	Deque	A deque allows efficient addition/removal from both ends, ideal for logging entry/exit events.
Event Registration System	Hash Table	Hash tables provide fast search, insertion, and removal by participant ID, perfect

		for events.
Library Book Borrowing	Binary Search Tree (BST)	A BST enables quick lookup, insertion, and removal of books by title or due date.
Bus Scheduling System	Graph	A graph models bus routes and stop connections, supporting traversal and connectivity queries.
Cafeteria Order Queue	Queue	A queue serves students in the order they arrive (FIFO), matching the cafeteria's serving logic.

CODE:

```
class EventRegistration:
    """
    Event Registration System using a hash table for fast participant
management.

    Methods:
        register(participant_id, name): Add a participant.
        remove(participant_id): Remove a participant.
        search(participant_id): Find a participant by ID.
        display(): Show all registered participants.
    """
    def __init__(self):
        self.table = {}

    def register(self, participant_id, name):
        """Register a new participant."""
        self.table[participant_id] = name

    def remove(self, participant_id):
        """Remove a participant by ID."""
        if participant_id in self.table:
            del self.table[participant_id]
            return True
        return False

    def search(self, participant_id):
        """Search for a participant by ID."""
        return self.table.get(participant_id, None)

    def display(self):
        """Display all registered participants."""
```

```python
        for pid, name in self.table.items():
            print(f"ID: {pid}, Name: {name}")

# Example usage
if __name__ == "__main__":
    event = EventRegistration()
    event.register(101, "Alice")
    event.register(102, "Bob")
    event.register(103, "Charlie")
    print("All participants:")
    event.display()
    print("Searching for ID 102:", event.search(102))
    print("Removing ID 101:", event.remove(101))
    print("All participants after removal:")
    event.display()
```

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    POR

l/Programs/Python/Python312/python.exe "c:/Users/
MENTS/Event Registration.py"
All participants:
ID: 101, Name: Alice
ID: 102, Name: Bob
ID: 103, Name: Charlie
Searching for ID 102: Bob
Removing ID 101: True
All participants after removal:
ID: 102, Name: Bob
ID: 103, Name: Charlie
PS C:\Users\ROHITH\OneDrive\Desktop\portofolio\AI
```