AI ASSISTED CODING

Lab Assignment-11.1

Name : T.Shylasri

H.T.NO: 2303A51876

Batch-14(LAB-11)

Date: 23-02-2026

ASSIGNMENT-11.1

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.

• 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.

AI Prompt

#Generate a Python Stack class using a list as the internal storage.

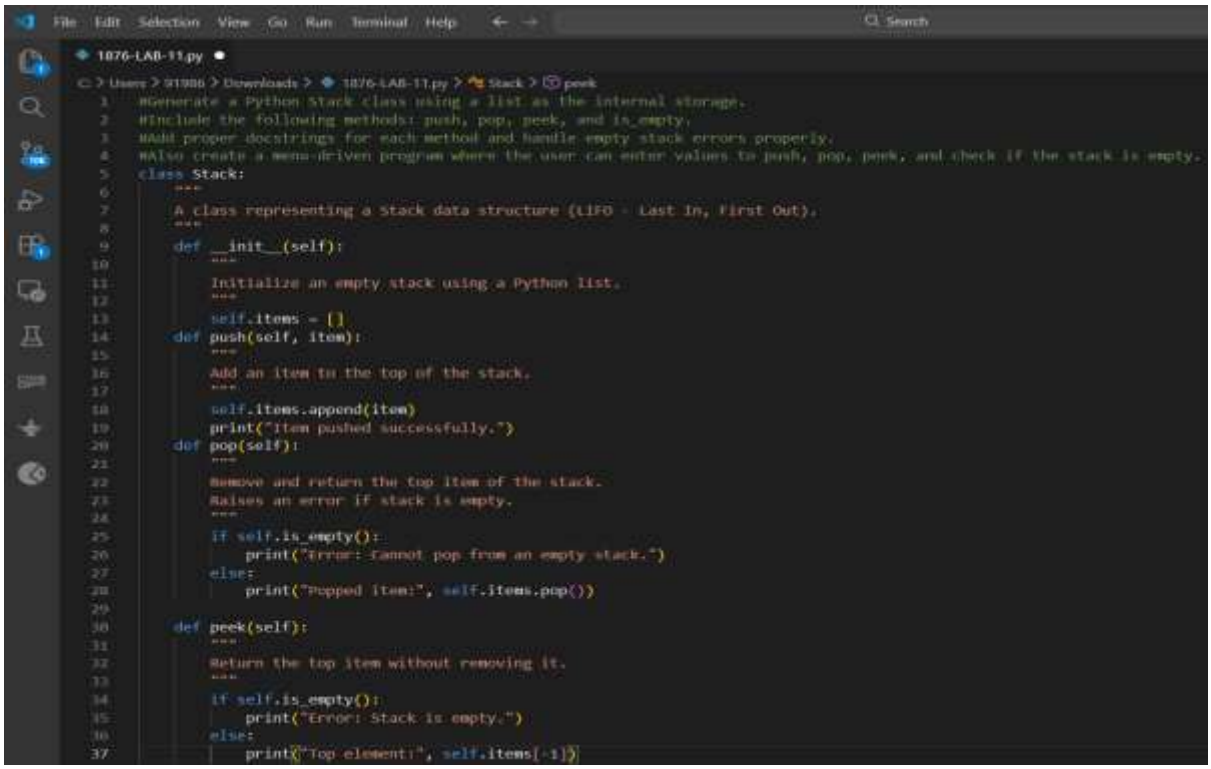
#Include the following methods: push, pop, peek, and is_empty.

#Add proper docstrings for each method and handle empty stack errors properly.

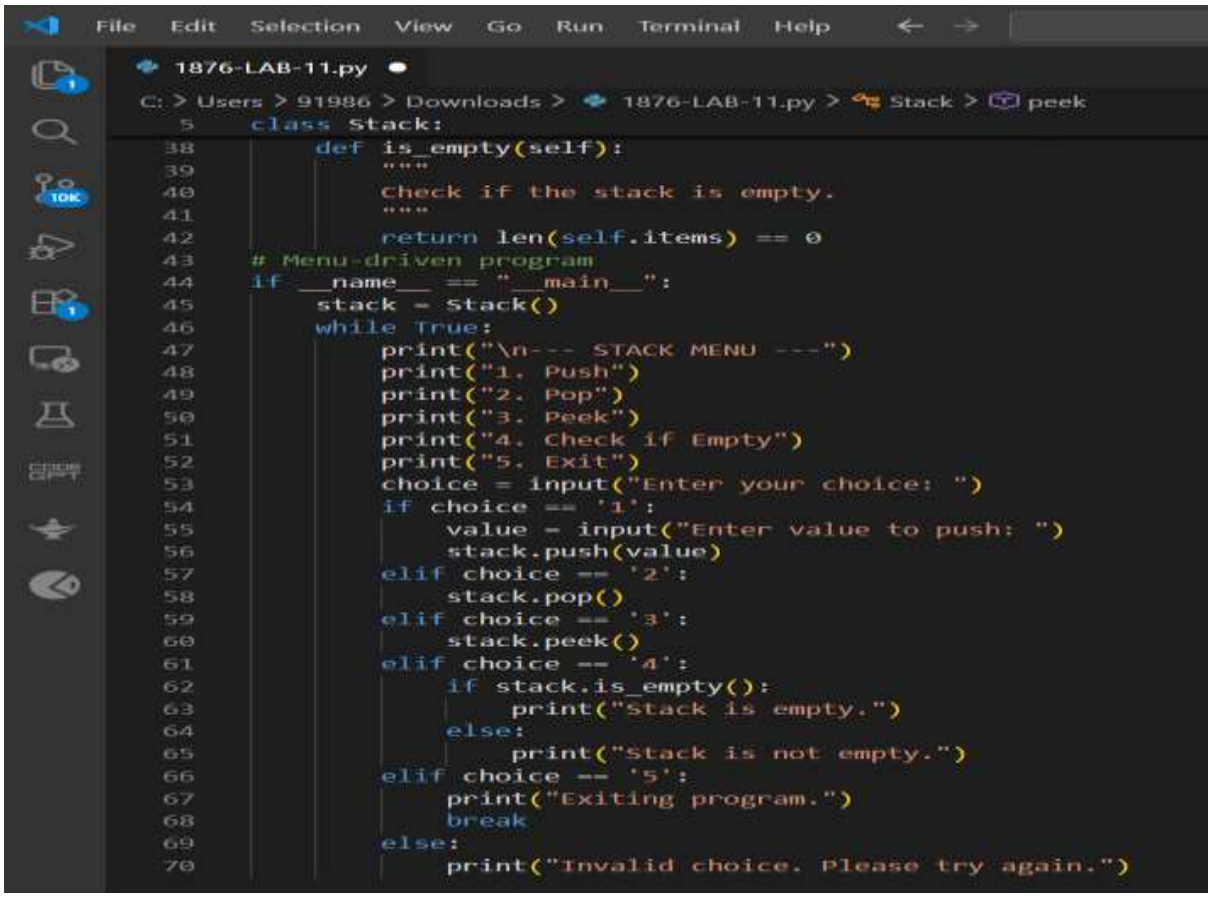
#Also create a menu-driven program where the user can enter values to push, pop, peek, and check if the stack is empty.

Prompt Screenshot

CODE

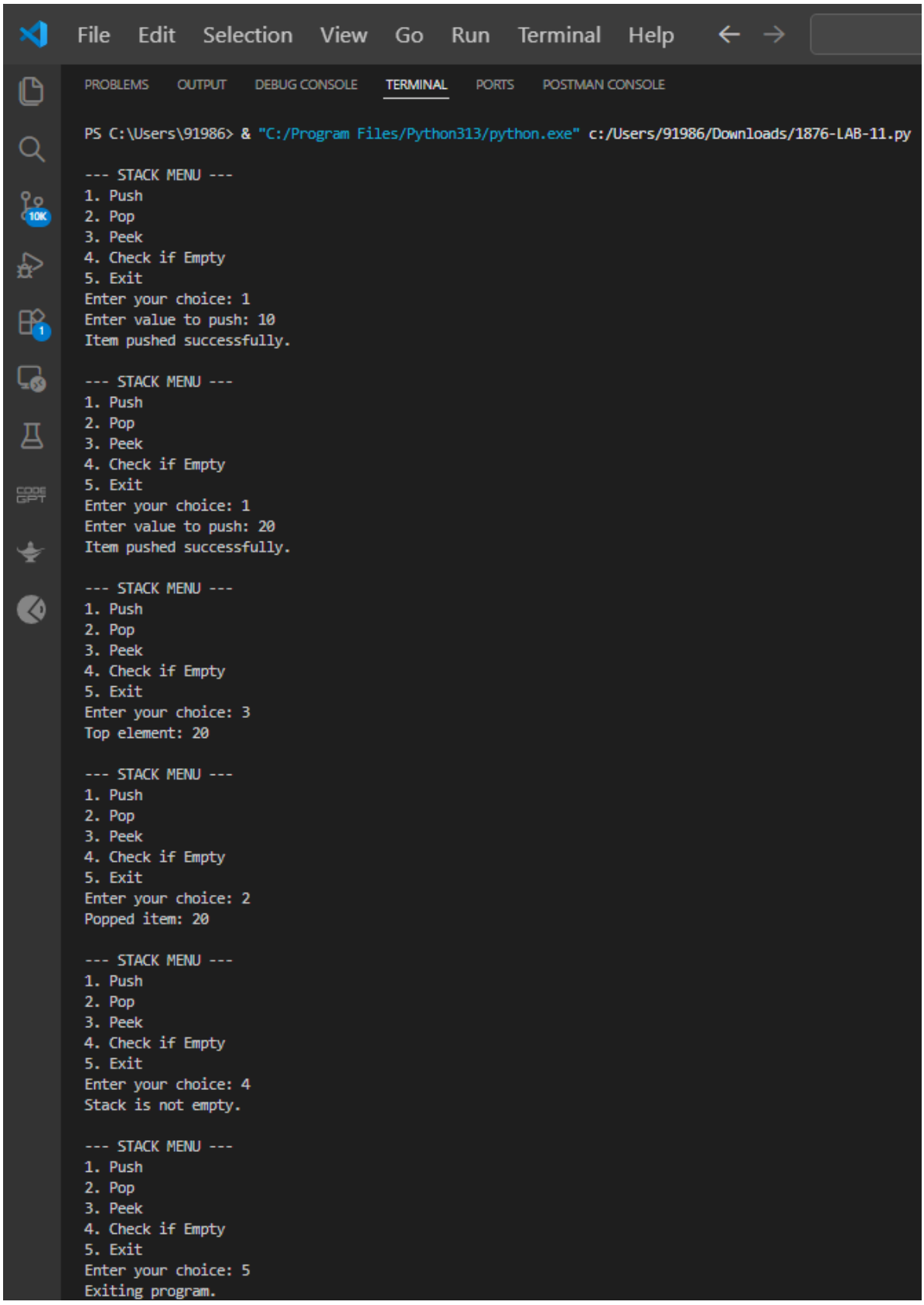


```
#Generate a Python Stack class using a list as the internal storage.
#Include the following methods: push, pop, peek, and is_empty.
#Add proper docstrings for each method and handle empty stack errors properly.
#Also create a menu-driven program where the user can enter values to push, pop, peek, and check if the stack is empty.
class Stack:
    """
    A class representing a Stack data structure (LIFO - Last In, First Out).
    """
    def __init__(self):
        """
        Initialize an empty stack using a Python list.
        """
        self.items = []
    def push(self, item):
        """
        Add an item to the top of the stack.
        """
        self.items.append(item)
        print("Item pushed successfully.")
    def pop(self):
        """
        Remove and return the top item of the stack.
        Raises an error if stack is empty.
        """
        if self.is_empty():
            print("Error: Cannot pop from an empty stack.")
        else:
            print("Popped item:", self.items.pop())

    def peek(self):
        """
        Return the top item without removing it.
        """
        if self.is_empty():
            print("Error: Stack is empty.")
        else:
            print("Top element:", self.items[-1])
```



```
    def is_empty(self):
        """
        Check if the stack is empty.
        """
        return len(self.items) == 0
# Menu-driven program
if __name__ == "__main__":
    stack = Stack()
    while True:
        print("\n--- STACK MENU ---")
        print("1. Push")
        print("2. Pop")
        print("3. Peek")
        print("4. Check if Empty")
        print("5. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            value = input("Enter value to push: ")
            stack.push(value)
        elif choice == '2':
            stack.pop()
        elif choice == '3':
            stack.peek()
        elif choice == '4':
            if stack.is_empty():
                print("Stack is empty.")
            else:
                print("Stack is not empty.")
        elif choice == '5':
            print("Exiting program.")
            break
        else:
            print("Invalid choice. Please try again.")
```

OUTPUT:


File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 1
Enter value to push: 10
Item pushed successfully.
--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 1
Enter value to push: 20
Item pushed successfully.
--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 3
Top element: 20
--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 2
Popped item: 20
--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 4
Stack is not empty.
--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice: 5
Exiting program.

Observation

- The stack follows **LIFO (Last In First Out)** principle.
- The last inserted element is removed first.
- The program correctly handles empty stack conditions.
- User interaction is implemented using a menu-driven system.
- All operations (push, pop, peek, is_empty) work as expected.

Justification

- Python list provides built-in append() and pop() methods which operate in **O(1)** time.
- Using a class improves modularity and reusability.
- Docstrings improve readability and documentation.
- Error handling prevents runtime crashes.
- Menu-driven design makes the program user-friendly.
- AI assistance improved structure, documentation, and code clarity.

Conclusion

The Stack data structure was successfully implemented using a Python list.
All fundamental operations were performed correctly with proper error handling.
The program demonstrates the LIFO behavior effectively through user interaction.

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.

AI Prompt

#Generate a Python Queue class using a list as the internal storage.

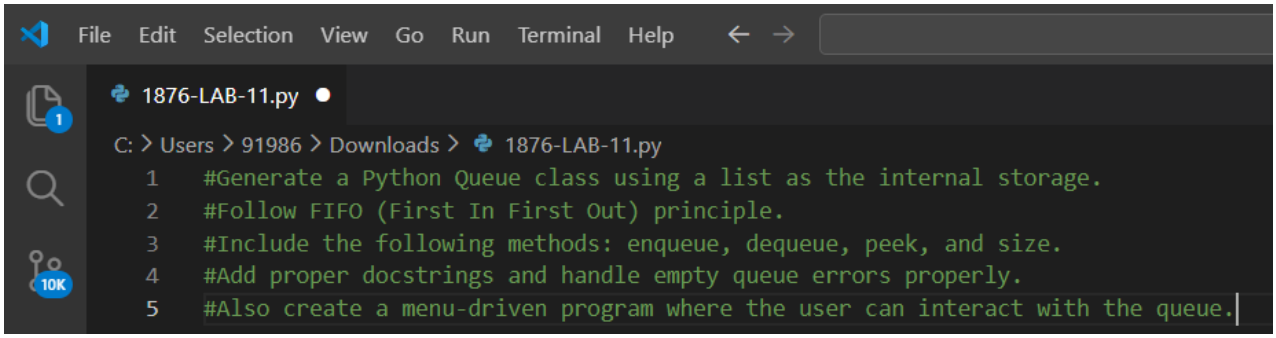
#Follow FIFO (First In First Out) principle.

#Include the following methods: enqueue, dequeue, peek, and size.

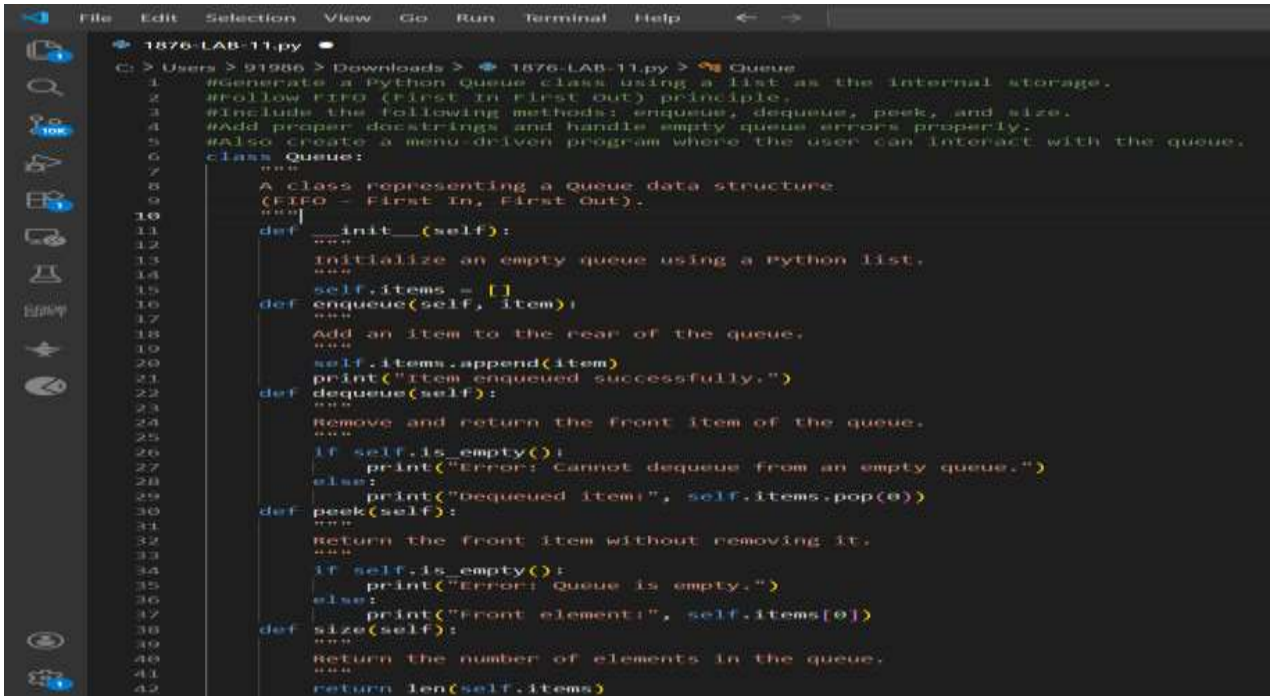
#Add proper docstrings and handle empty queue errors properly.

#Also create a menu-driven program where the user can interact with the queue.

Prompt Screenshot



CODE



```
    def is_empty(self):
        """
        Check if the queue is empty.
        """
        return len(self.items) == 0
# Menu-driven program
if __name__ == "__main__":
    queue = Queue()
    while True:
        print("\n--- QUEUE MENU ---")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Peek")
        print("4. Size")
        print("5. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            value = input("Enter value to enqueue: ")
            queue.enqueue(value)
        elif choice == '2':
            queue.dequeue()
        elif choice == '3':
            queue.peek()
        elif choice == '4':
            print("Queue size:", queue.size())
        elif choice == '5':
            print("Exiting program.")
            break
        else:
            print("Invalid choice. Please try again.")
```

OUTPUT

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 1
Enter value to enqueue: 10
Item enqueued successfully.

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 1
Enter value to enqueue: 20
Item enqueued successfully.

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 3
Front element: 10

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 2
Dequeued item: 10

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 4
Queue size: 1

--- QUEUE MENU ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice: 5
Exiting program.
```

Observation

- The queue follows **FIFO (First In First Out)** principle.
- The first inserted element is removed first.
- The program handles empty queue errors properly.
- User interaction is implemented through a menu-driven system.
- All required methods (enqueue, dequeue, peek, size) work correctly.

Justification

- Python lists allow easy insertion using append().
- The pop(0) method removes the first element, maintaining FIFO order.
- Class-based implementation improves modularity and clarity.
- Docstrings improve documentation and readability.
- AI assistance helped generate structured, optimized, and well-documented code.

Conclusion

The Queue data structure was successfully implemented using Python lists.
All fundamental FIFO operations were performed correctly with proper error handling.
The program demonstrates queue behavior effectively through user interaction.

Time Complexity

Operation	Time Complexity
enqueue()	O(1)
dequeue()	O(n)
peek()	O(1)
size()	O(1)

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

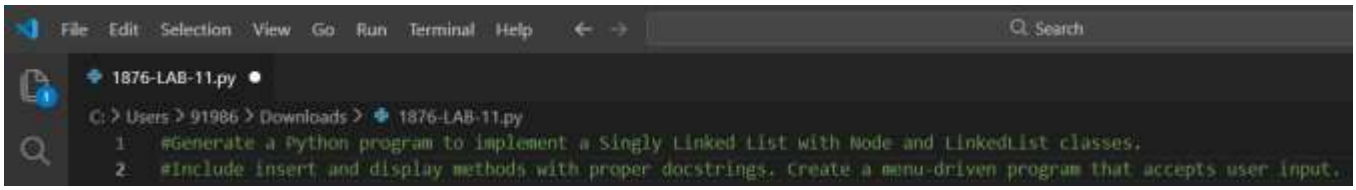
documentation.

AI Prompt

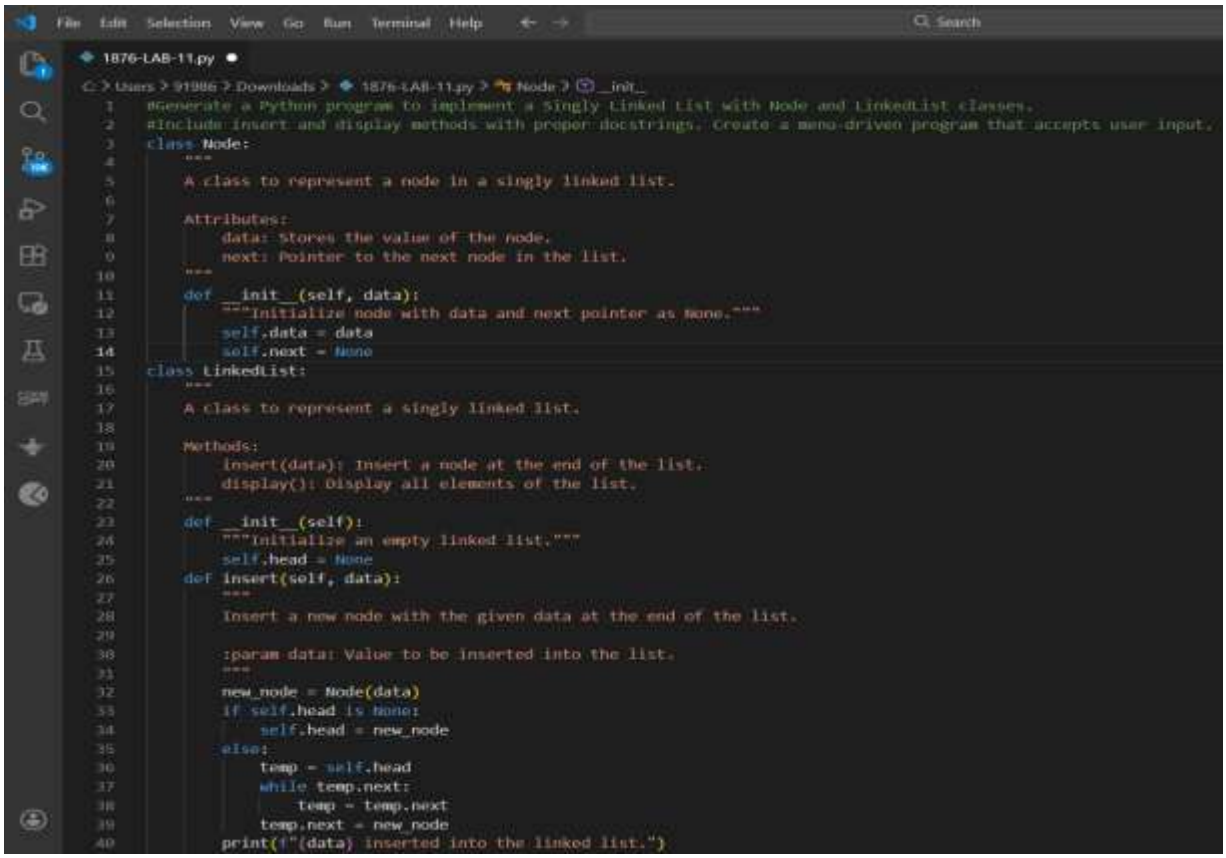
#Generate a Python program to implement a Singly Linked List with Node and LinkedList classes.

#Include insert and display methods with proper docstrings. Create a menu-driven program that accepts user input.

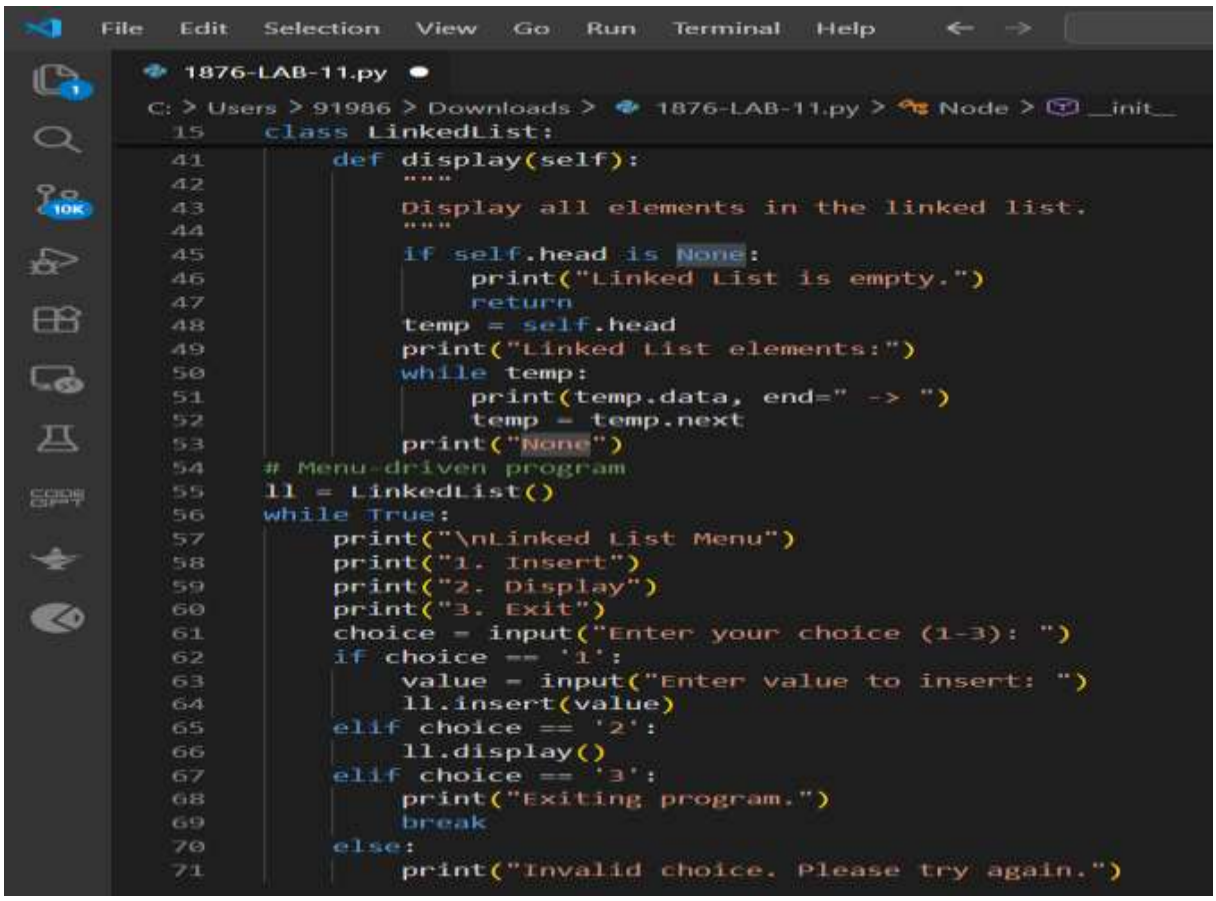
Prompt Screenshot



CODE

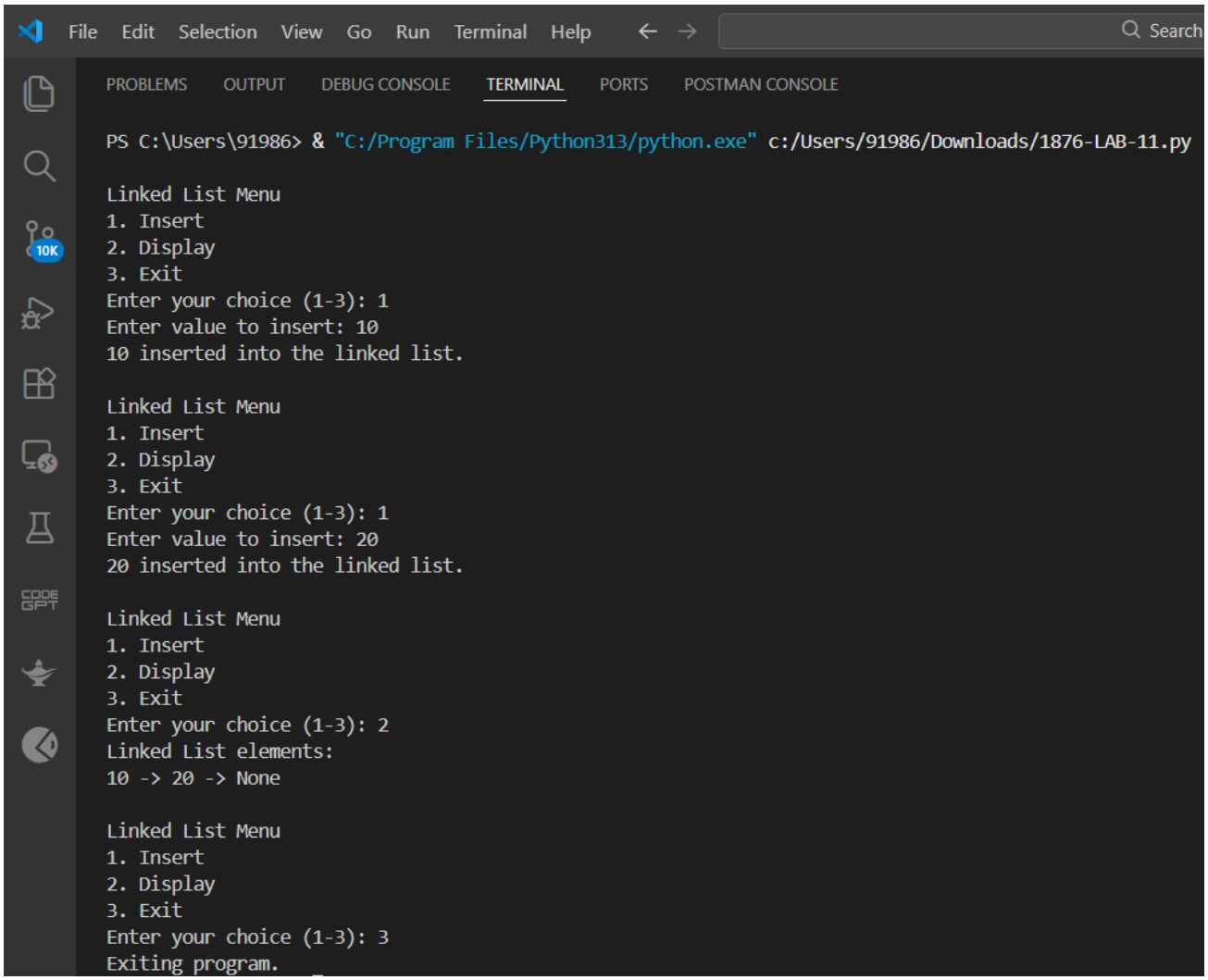


```python
#Generate a Python program to implement a Singly Linked List with Node and LinkedList classes.
#Include insert and display methods with proper docstrings. Create a menu-driven program that accepts user input.
class Node:
    """
    A class to represent a node in a singly linked list.

    Attributes:
        data: Stores the value of the node.
        next: Pointer to the next node in the list.
    """
    def __init__(self, data):
        """Initialize node with data and next pointer as None."""
        self.data = data
        self.next = None
class LinkedList:
    """
    A class to represent a singly linked list.

    Methods:
        insert(data): Insert a node at the end of the list.
        display(): Display all elements of 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.

        :param data: Value to be inserted into the list.
        """
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = new_node
        print(f"{data} inserted into the linked list.")
```



```python
class LinkedList:
    def display(self):
        """
        Display all elements in the linked list.
        """
        if self.head is None:
            print("Linked List is empty.")
            return
        temp = self.head
        print("Linked List elements:")
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")
# Menu-driven program
ll = LinkedList()
while True:
    print("\nLinked List Menu")
    print("1. Insert")
    print("2. Display")
    print("3. Exit")
    choice = input("Enter your choice (1-3): ")
    if choice == '1':
        value = input("Enter value to insert: ")
        ll.insert(value)
    elif choice == '2':
        ll.display()
    elif choice == '3':
        print("Exiting program.")
        break
    else:
        print("Invalid choice. Please try again.")
```

OUTPUT



Observation

- Nodes are created dynamically.
- Each node stores data and a pointer to the next node.
- The list maintains insertion order.
- If the list is empty, appropriate message is displayed.
- The linked list works correctly with user input.

Justification

- Dynamic memory allocation allows flexible size.
- Efficient insertion compared to arrays (no shifting required).
- Clear docstrings improve readability and documentation.

- The structure meets assignment requirements.
- Menu-driven interface ensures easy testing.

Conclusion

The **Singly Linked List** was successfully implemented using:

- Node class
- LinkedList class
- insert() and display() methods

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:

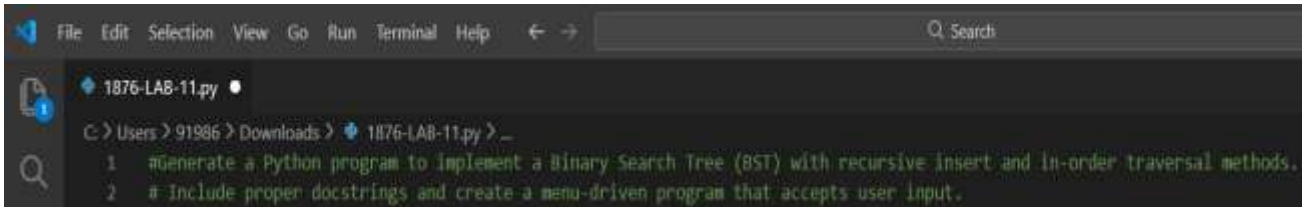
• BST implementation with recursive insert and traversal methods.

AI Prompt

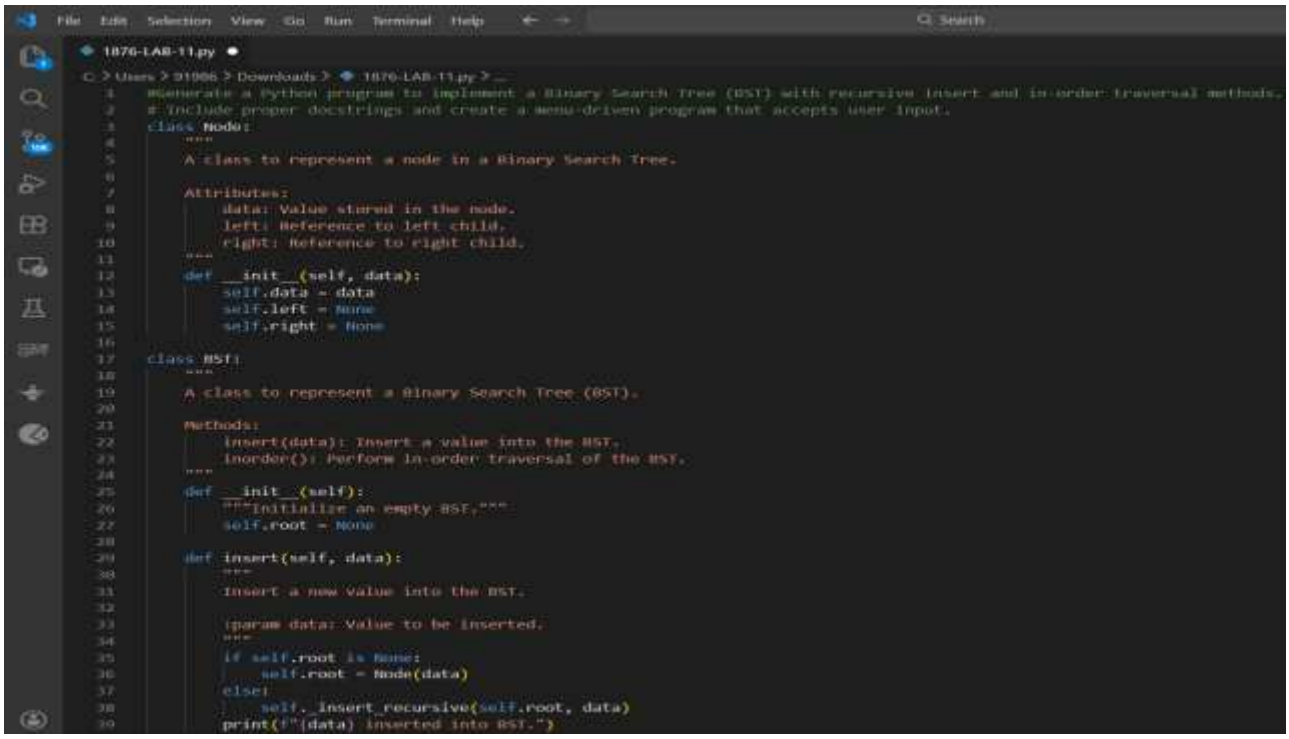
#Generate a Python program to implement a Binary Search Tree (BST) with recursive insert and in-order traversal methods.

Include proper docstrings and create a menu-driven program that accepts user input.

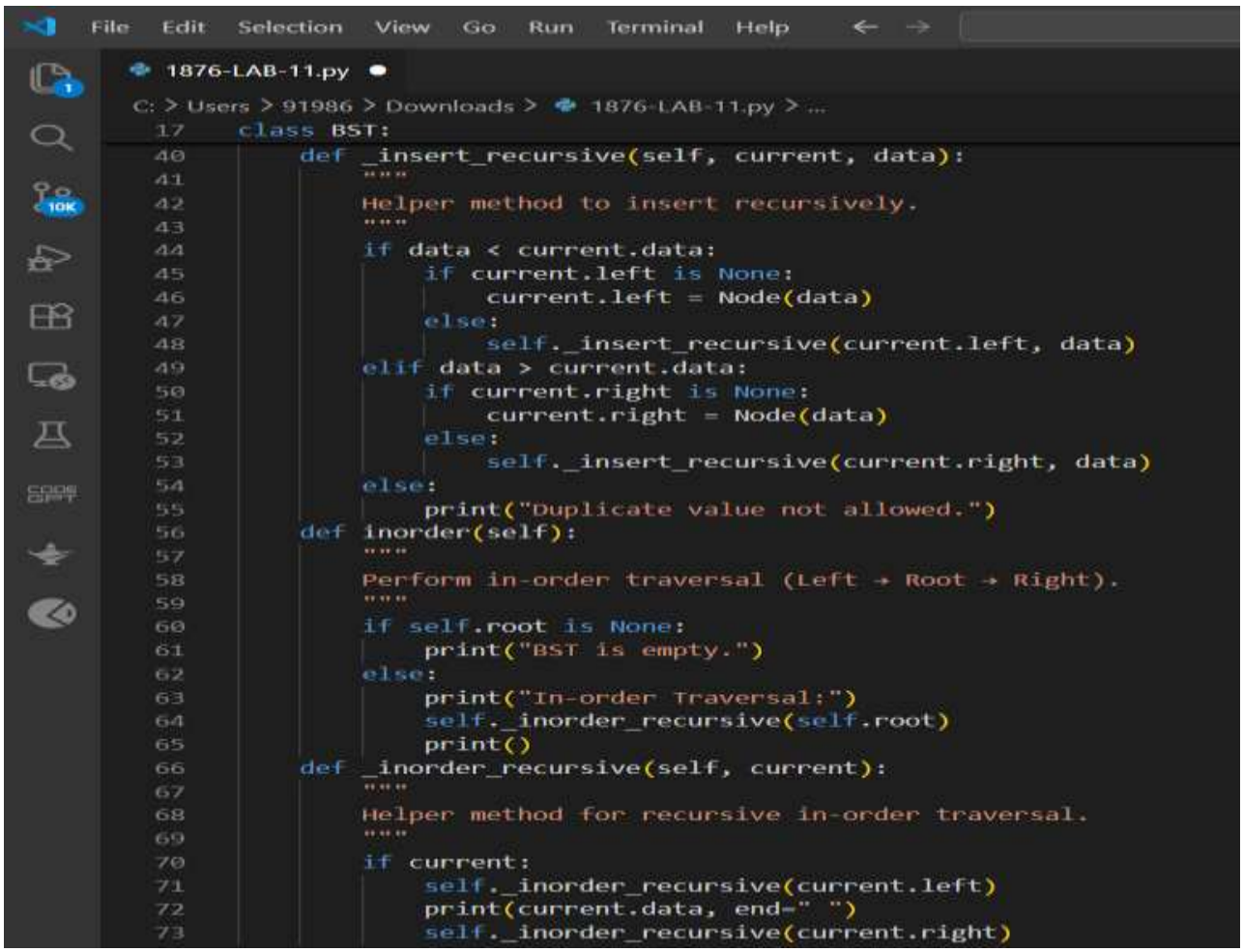
Prompt Screenshot



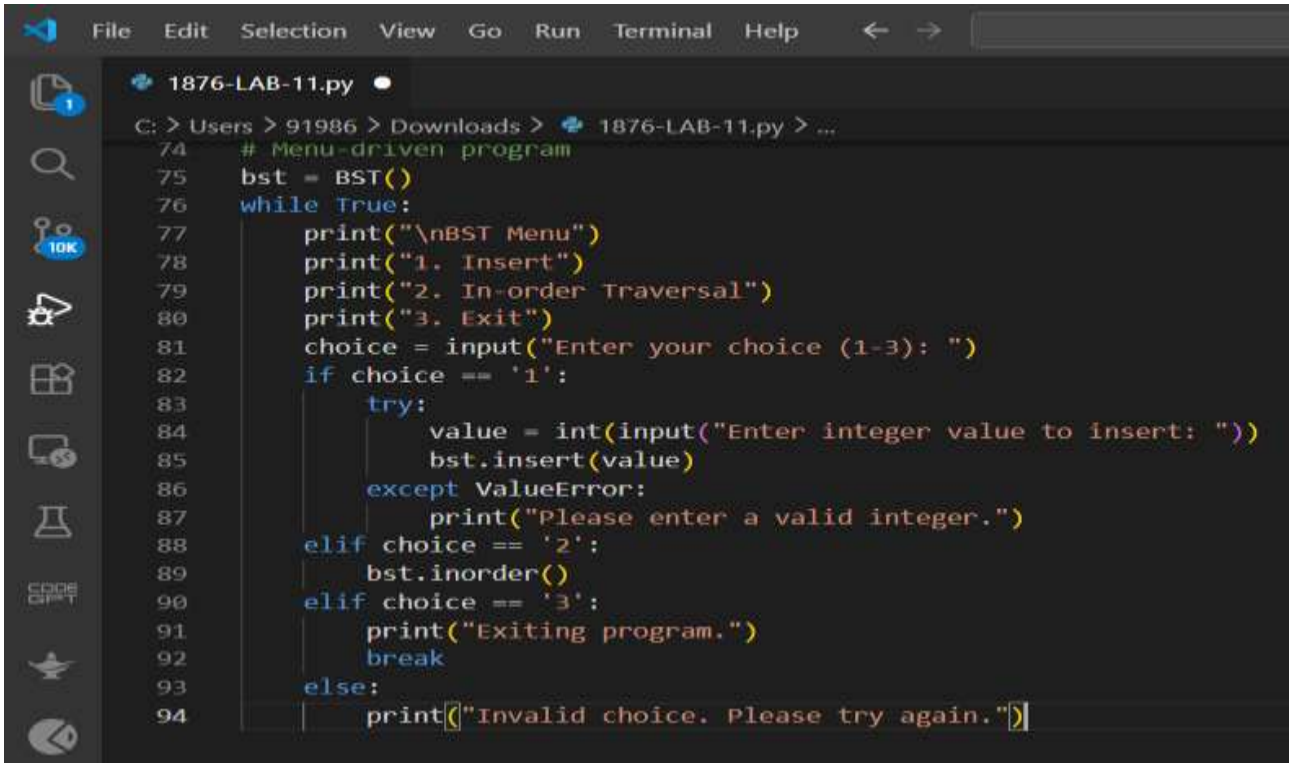
CODE



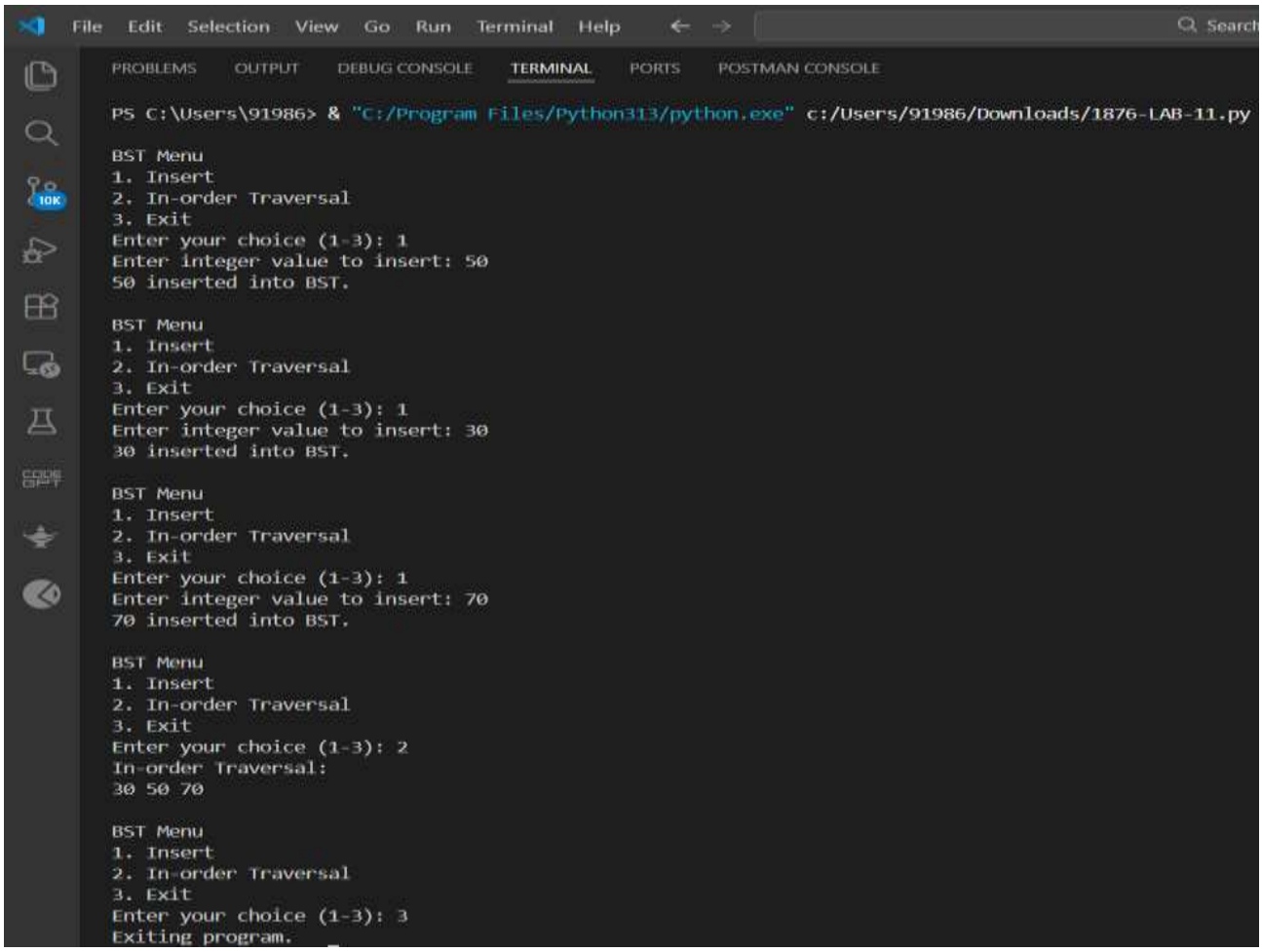
```
#Generate a Python program to implement a Binary Search Tree (BST) with recursive insert and in-order traversal methods.
# Include proper docstrings and create a menu-driven program that accepts user input.
class Node:
    """
    A class to represent a node in a Binary Search Tree.

    Attributes:
        data: Value stored in the node.
        left: Reference to left child.
        right: Reference to right child.
    """
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BST:
    """
    A class to represent a Binary Search Tree (BST).

    Methods:
        insert(data): Insert a value into the BST.
        inorder(): Perform in-order traversal of the BST.
    """
    def __init__(self):
        """Initialize an empty BST."""
        self.root = None

    def insert(self, data):
        """
        Insert a new value into the BST.

        :param data: Value to be inserted.
        """
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert_recursive(self.root, data)
        print(f"{data} inserted into BST.")
```



```
class BST:
    def _insert_recursive(self, current, data):
        """
        Helper method to insert recursively.
        """
        if data < current.data:
            if current.left is None:
                current.left = Node(data)
            else:
                self._insert_recursive(current.left, data)
        elif data > current.data:
            if current.right is None:
                current.right = Node(data)
            else:
                self._insert_recursive(current.right, data)
        else:
            print("Duplicate value not allowed.")
    def inorder(self):
        """
        Perform in-order traversal (Left → Root → Right).
        """
        if self.root is None:
            print("BST is empty.")
        else:
            print("In-order Traversal:")
            self._inorder_recursive(self.root)
            print()
    def _inorder_recursive(self, current):
        """
        Helper method for recursive in-order traversal.
        """
        if current:
            self._inorder_recursive(current.left)
            print(current.data, end=" ")
            self._inorder_recursive(current.right)
```


File Edit Selection View Go Run Terminal Help
1876-LAB-11.py
C: > Users > 91986 > Downloads > 1876-LAB-11.py > ...
74 # Menu-driven program
75 bst = BST()
76 while True:
77 print("\nBST Menu")
78 print("1. Insert")
79 print("2. In-order Traversal")
80 print("3. Exit")
81 choice = input("Enter your choice (1-3): ")
82 if choice == '1':
83 try:
84 value = int(input("Enter integer value to insert: "))
85 bst.insert(value)
86 except ValueError:
87 print("Please enter a valid integer.")
88 elif choice == '2':
89 bst.inorder()
90 elif choice == '3':
91 print("Exiting program.")
92 break
93 else:
94 print("Invalid choice. Please try again.")

OUTPUT


File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
BST Menu
1. Insert
2. In-order Traversal
3. Exit
Enter your choice (1-3): 1
Enter integer value to insert: 50
50 inserted into BST.
BST Menu
1. Insert
2. In-order Traversal
3. Exit
Enter your choice (1-3): 1
Enter integer value to insert: 30
30 inserted into BST.
BST Menu
1. Insert
2. In-order Traversal
3. Exit
Enter your choice (1-3): 1
Enter integer value to insert: 70
70 inserted into BST.
BST Menu
1. Insert
2. In-order Traversal
3. Exit
Enter your choice (1-3): 2
In-order Traversal:
30 50 70
BST Menu
1. Insert
2. In-order Traversal
3. Exit
Enter your choice (1-3): 3
Exiting program.

Observation

- The BST maintains proper ordering.
- Values smaller than root go to left subtree.
- Values larger than root go to right subtree.
- In-order traversal prints values in sorted order.
- Recursive logic works correctly.

Justification

- Recursive implementation simplifies logic.
- BST allows efficient search, insert, delete operations.
- Average time complexity:
    - Insert: O(log n)
    - Traversal: O(n)
- Clear documentation improves readability.
- Menu-driven interface makes testing easy.

Conclusion

The **Binary Search Tree (BST)** was successfully implemented with:

- Recursive insert() method
- Recursive in-order traversal
- Proper documentation
- User input support

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.

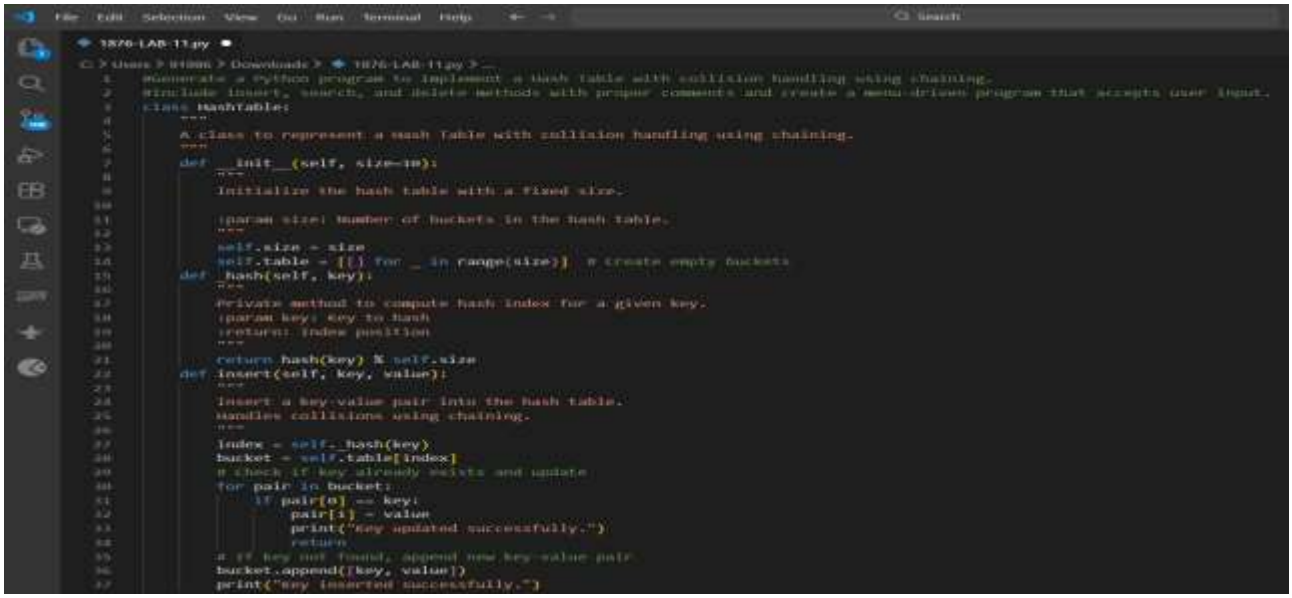
AI Prompt

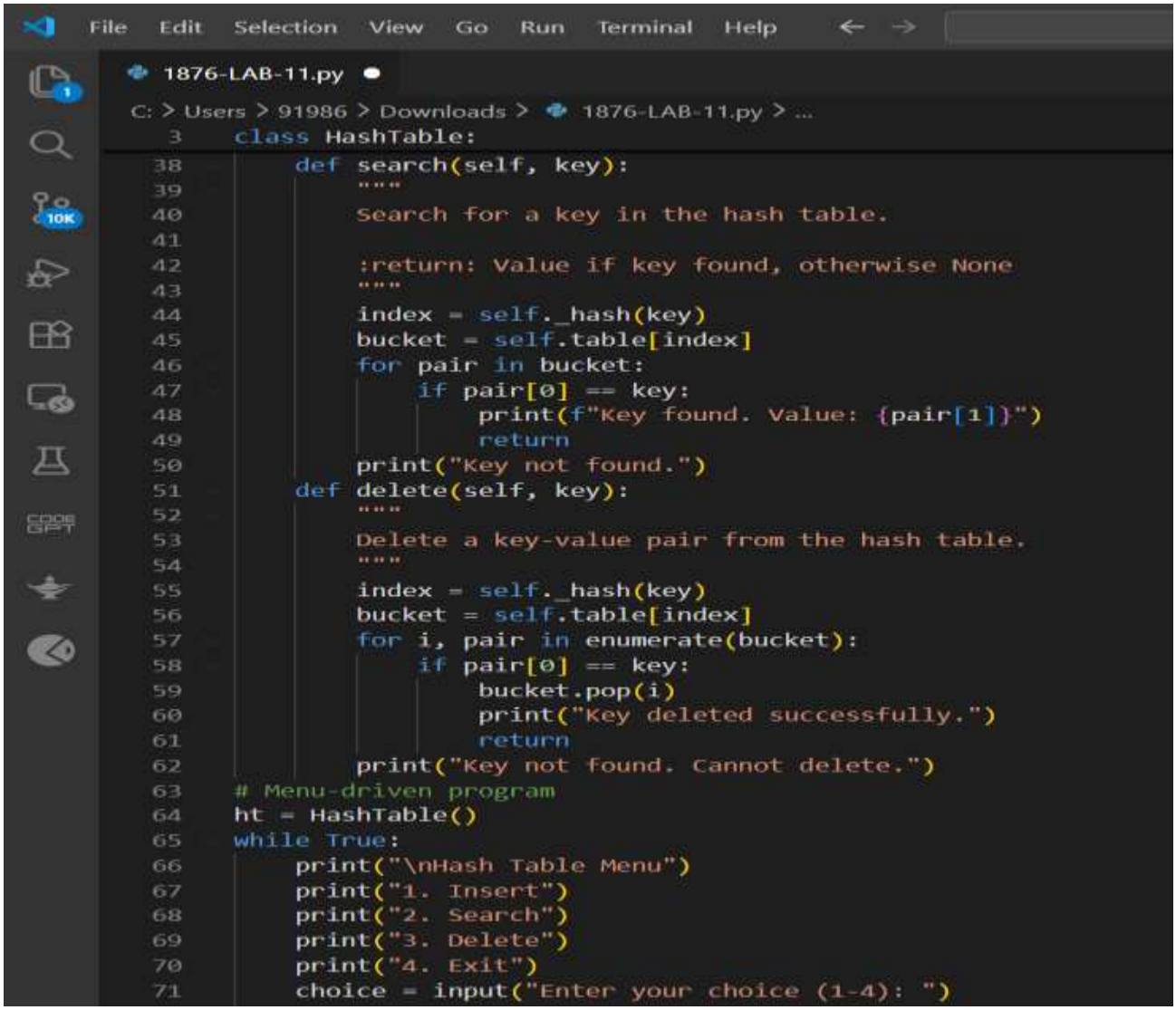
#Generate a Python program to implement a Hash Table with collision handling using chaining.

#Include insert, search, and delete methods with proper comments and create a menu-driven program that accepts user input.

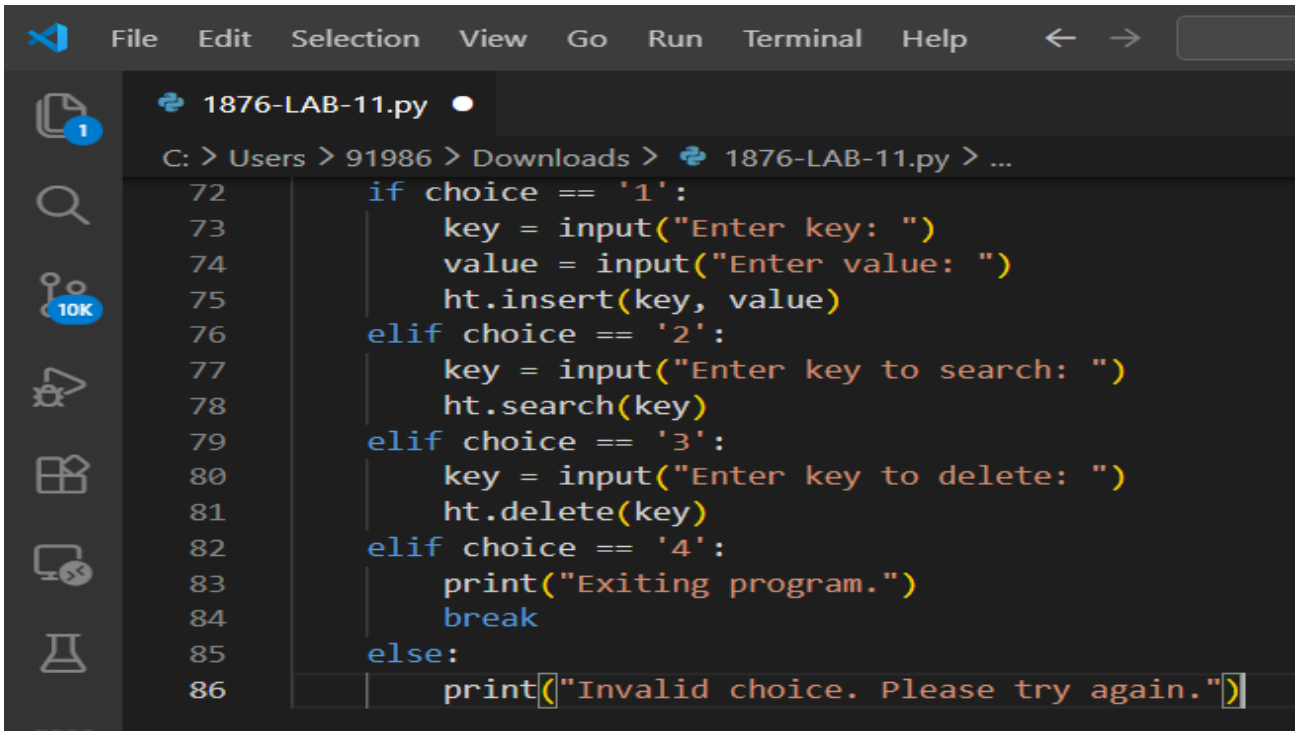
Prompt Screenshot

CODE



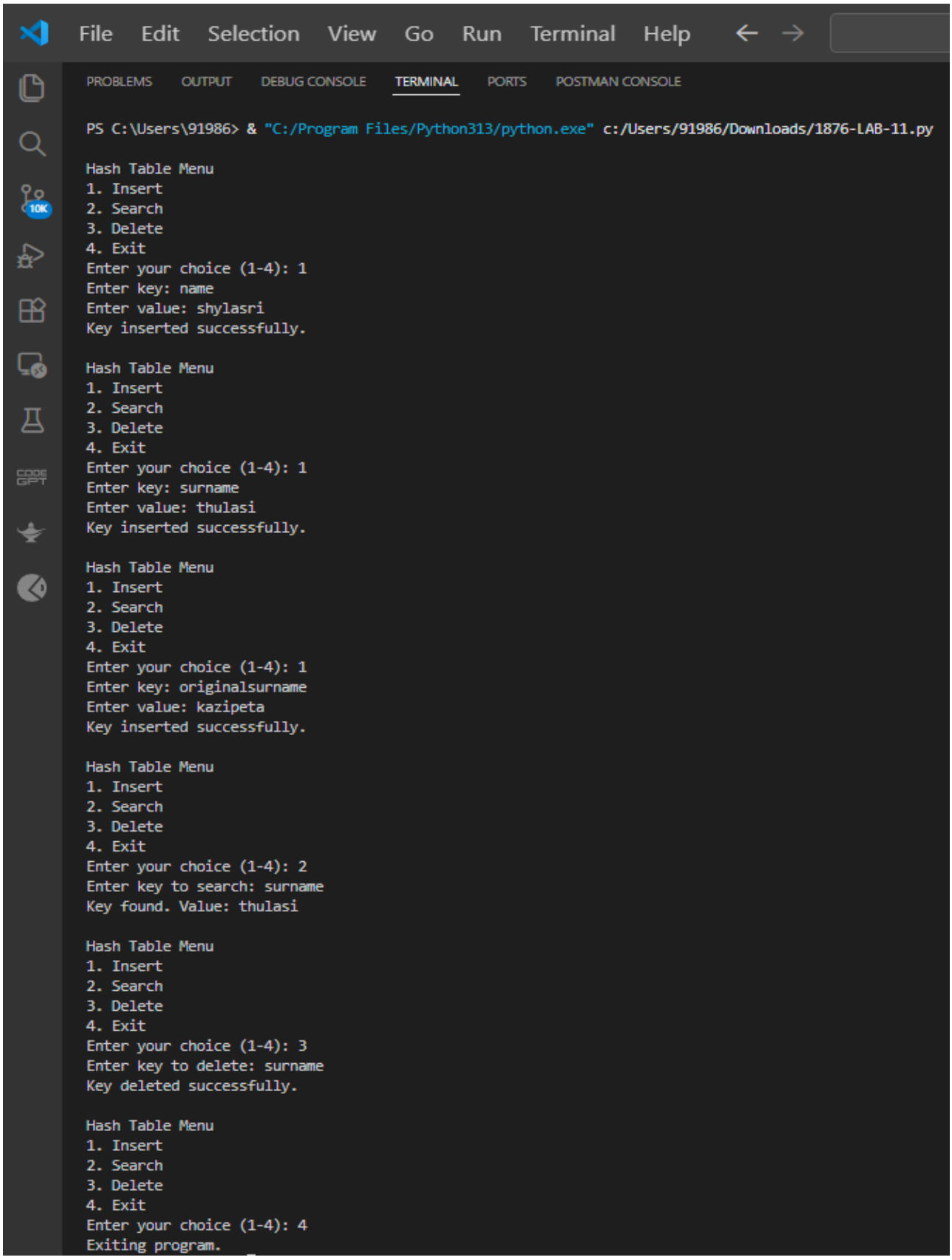


```
File  Edit  Selection  View  Go  Run  Terminal  Help

1876-LAB-11.py

C: > Users > 91986 > Downloads > 1876-LAB-11.py > ...

class HashTable:
    def search(self, key):
        """
        Search for a key in the hash table.

        :return: Value if key found, otherwise None
        """
        index = self._hash(key)
        bucket = self.table[index]
        for pair in bucket:
            if pair[0] == key:
                print(f"Key found. Value: {pair[1]}")
                return
        print("Key not found.")
    def delete(self, key):
        """
        Delete a key-value pair from the hash table.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for i, pair in enumerate(bucket):
            if pair[0] == key:
                bucket.pop(i)
                print("Key deleted successfully.")
                return
        print("Key not found. Cannot delete.")
# Menu-driven program
ht = HashTable()
while True:
    print("\nHash Table Menu")
    print("1. Insert")
    print("2. Search")
    print("3. Delete")
    print("4. Exit")
    choice = input("Enter your choice (1-4): ")
```



```
File  Edit  Selection  View  Go  Run  Terminal  Help

1876-LAB-11.py

C: > Users > 91986 > Downloads > 1876-LAB-11.py > ...

    if choice == '1':
        key = input("Enter key: ")
        value = input("Enter value: ")
        ht.insert(key, value)
    elif choice == '2':
        key = input("Enter key to search: ")
        ht.search(key)
    elif choice == '3':
        key = input("Enter key to delete: ")
        ht.delete(key)
    elif choice == '4':
        print("Exiting program.")
        break
    else:
        print("Invalid choice. Please try again.")
```

OUTPUT


File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 1
Enter key: name
Enter value: shylasri
Key inserted successfully.
Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 1
Enter key: surname
Enter value: thulasi
Key inserted successfully.
Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 1
Enter key: originalsurname
Enter value: kazipeta
Key inserted successfully.
Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 2
Enter key to search: surname
Key found. Value: thulasi
Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 3
Enter key to delete: surname
Key deleted successfully.
Hash Table Menu
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 4
Exiting program.

Observation

- Keys are stored using hash indexing.
- Collisions are handled using **chaining (list of key-value pairs per bucket)**.
- Insert updates value if key already exists.
- Search retrieves correct value.
- Delete removes key-value pair correctly.
- Program handles invalid operations properly.

Justification

- Chaining is simple and effective for collision handling.
- Python lists make bucket management easy.
- Average time complexity:
    - Insert: O(1)
    - Search: O(1)
    - Delete: O(1)
- Worst case (many collisions): O(n)
- Code is well-commented and easy to understand.
- Menu-driven program ensures user interaction.

Conclusion

The **Hash Table** was successfully implemented with:

- Insert, Search, and Delete methods
- Collision handling using chaining
- Proper documentation
- User input functionality

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.

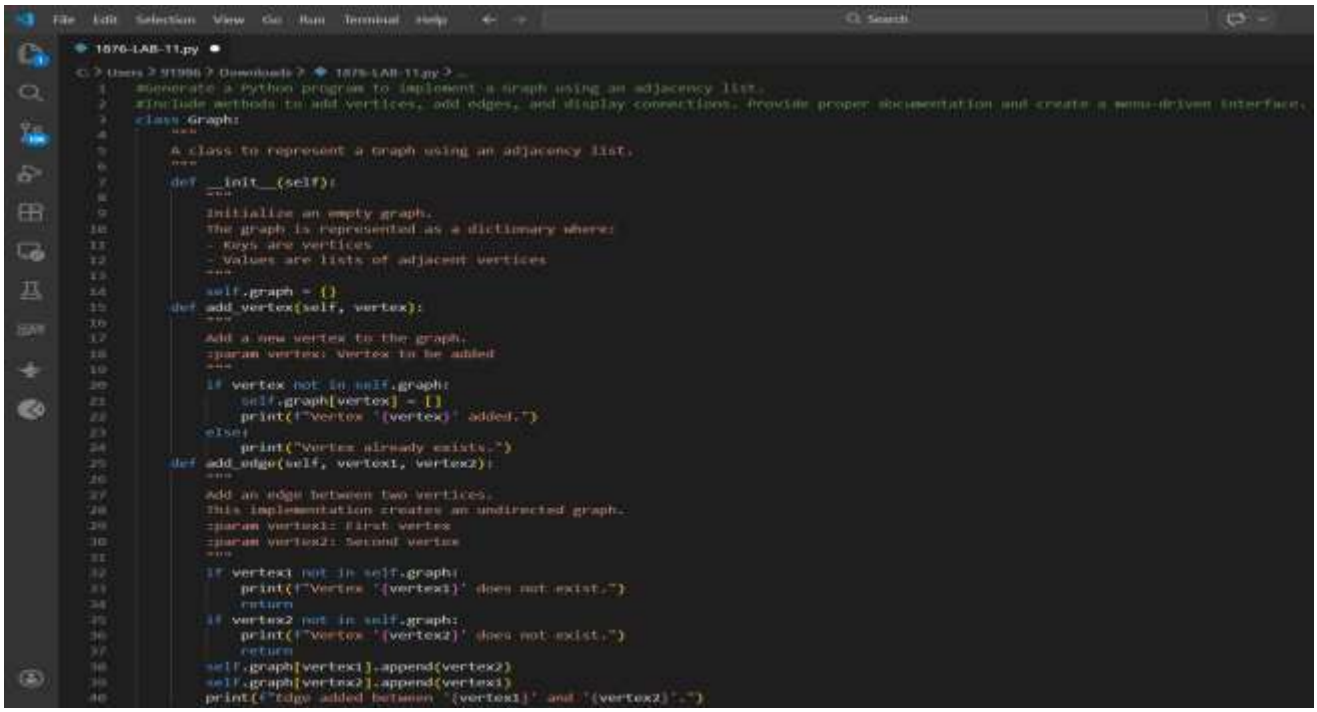
AI Prompt

#Generate a Python program to implement a Graph using an adjacency list.

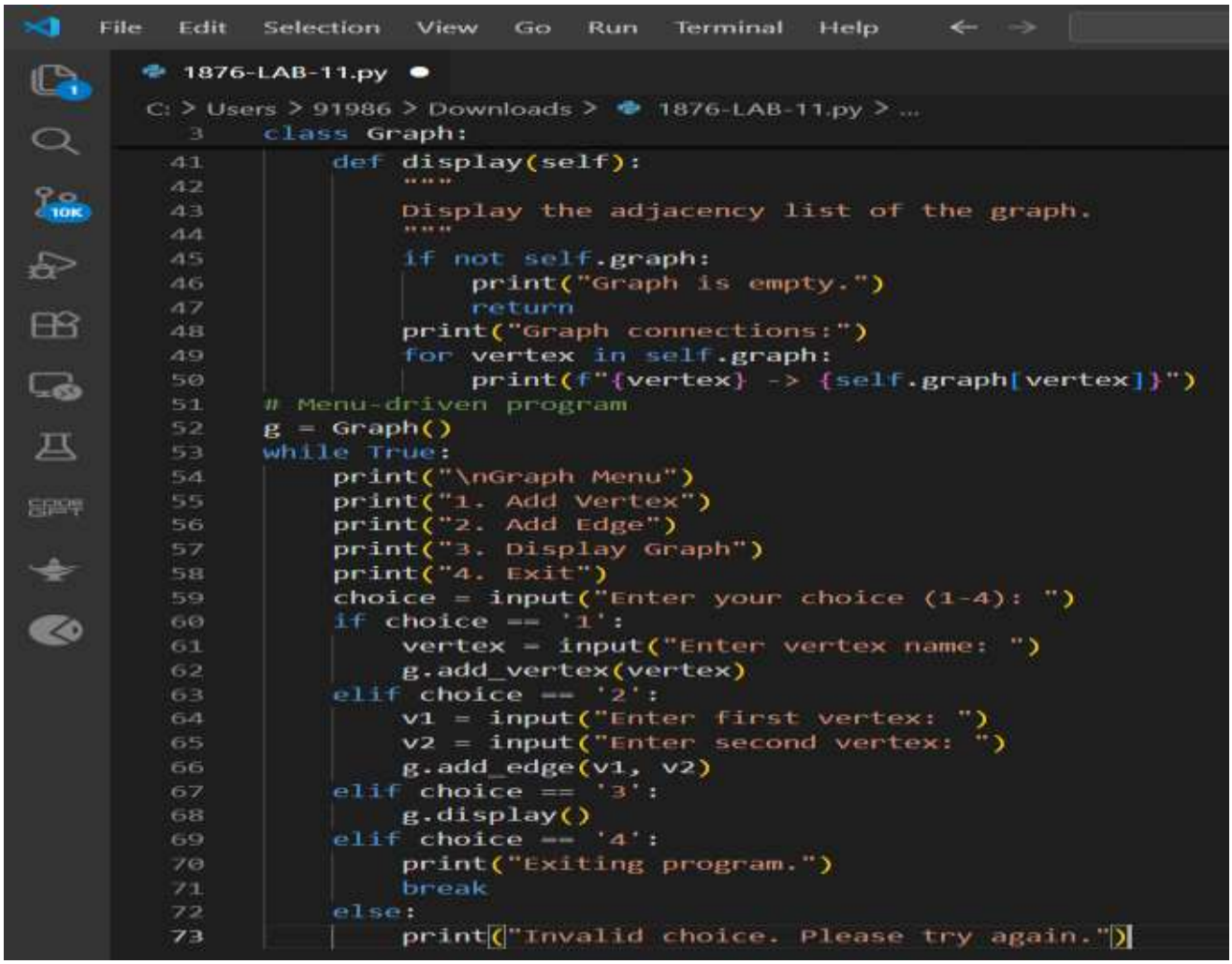
#Include methods to add vertices, add edges, and display connections. Provide proper documentation and create a menu-driven interface.

Prompt Screenshot

CODE

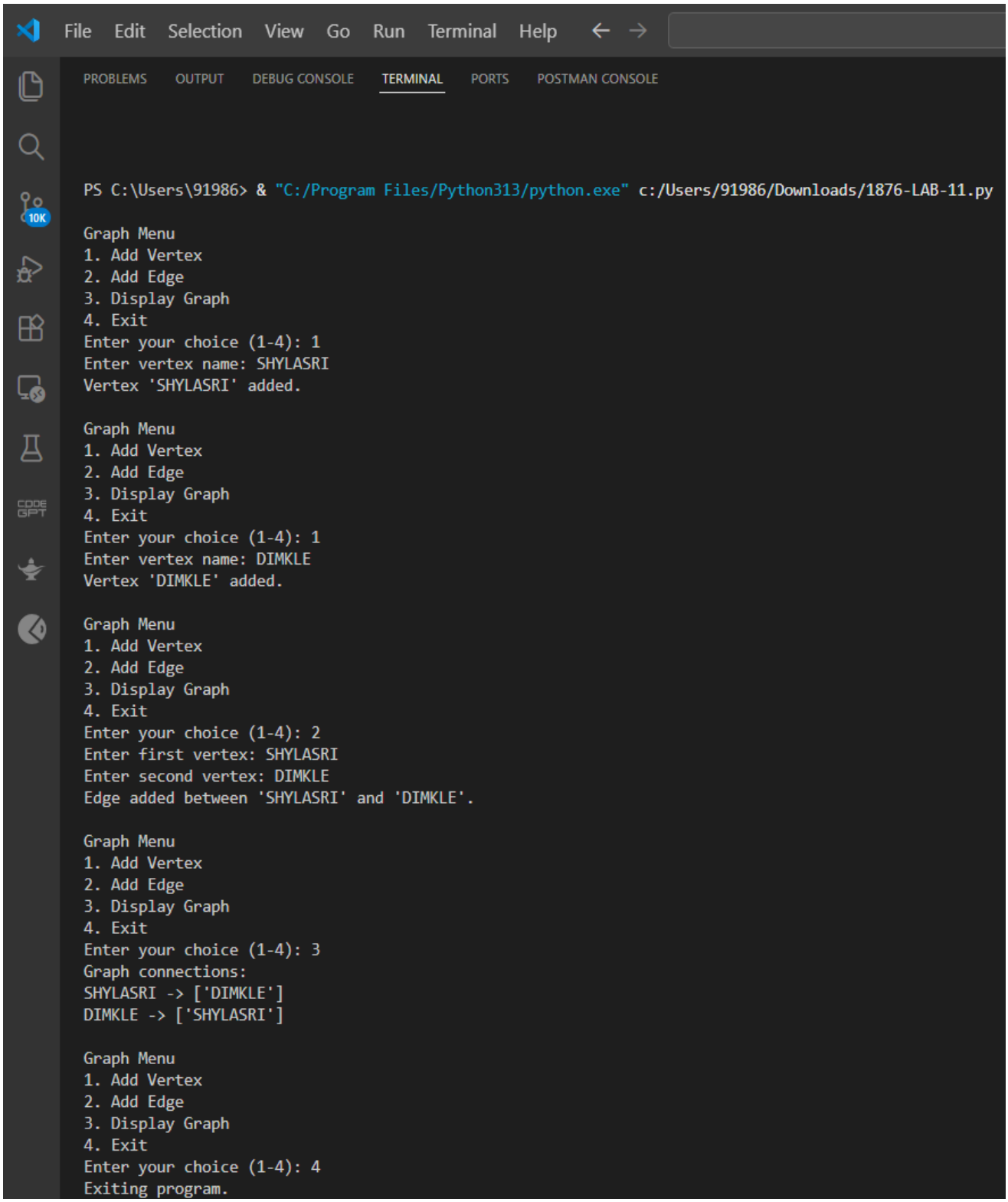


```python
#Generate a Python program to implement a graph using an adjacency list.
#Include methods to add vertices, add edges, and display connections. Provide proper documentation and create a menu-driven interface.
class Graph:
    """
    A class to represent a graph using an adjacency list.
    """
    def __init__(self):
        """
        Initialize an empty graph.
        The graph is represented as a dictionary where:
        - Keys are vertices
        - Values are lists of adjacent vertices
        """
        self.graph = {}
    def add_vertex(self, vertex):
        """
        Add a new vertex to the graph.
        :param vertex: Vertex to be added
        """
        if vertex not in self.graph:
            self.graph[vertex] = []
            print(f"Vertex '{vertex}' added.")
        else:
            print("Vertex already exists.")
    def add_edge(self, vertex1, vertex2):
        """
        Add an edge between two vertices.
        This implementation creates an undirected graph.
        :param vertex1: First vertex
        :param vertex2: Second vertex
        """
        if vertex1 not in self.graph:
            print(f"Vertex '{vertex1}' does not exist.")
            return
        if vertex2 not in self.graph:
            print(f"Vertex '{vertex2}' does not exist.")
            return
        self.graph[vertex1].append(vertex2)
        self.graph[vertex2].append(vertex1)
        print(f"Edge added between '{vertex1}' and '{vertex2}'.")
```



```python
class Graph:
    def display(self):
        """
        Display the adjacency list of the graph.
        """
        if not self.graph:
            print("Graph is empty.")
            return
        print("Graph connections:")
        for vertex in self.graph:
            print(f"{vertex} -> {self.graph[vertex]}")
# Menu-driven program
g = Graph()
while True:
    print("\nGraph Menu")
    print("1. Add Vertex")
    print("2. Add Edge")
    print("3. Display Graph")
    print("4. Exit")
    choice = input("Enter your choice (1-4): ")
    if choice == '1':
        vertex = input("Enter vertex name: ")
        g.add_vertex(vertex)
    elif choice == '2':
        v1 = input("Enter first vertex: ")
        v2 = input("Enter second vertex: ")
        g.add_edge(v1, v2)
    elif choice == '3':
        g.display()
    elif choice == '4':
        print("Exiting program.")
        break
    else:
        print("Invalid choice. Please try again.")
```

OUTPUT


File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
Graph Menu
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 1
Enter vertex name: SHYLASRI
Vertex 'SHYLASRI' added.
Graph Menu
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 1
Enter vertex name: DIMKLE
Vertex 'DIMKLE' added.
Graph Menu
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 2
Enter first vertex: SHYLASRI
Enter second vertex: DIMKLE
Edge added between 'SHYLASRI' and 'DIMKLE'.
Graph Menu
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 3
Graph connections:
SHYLASRI -> ['DIMKLE']
DIMKLE -> ['SHYLASRI']
Graph Menu
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 4
Exiting program.

Observation

- Vertices are stored as dictionary keys.
- Edges are stored as lists of adjacent vertices.
- The graph correctly shows bidirectional (undirected) connections.
- The program handles invalid vertex inputs properly.
- The adjacency list representation works efficiently.

Justification

- Adjacency list is memory efficient.
- Suitable for sparse graphs.
- Easy to implement using Python dictionary.
- Average time complexity:
    - Add Vertex: O(1)
    - Add Edge: O(1)
- Clear documentation improves readability.
- Menu-driven interface ensures user interaction.

Conclusion

The **Graph using Adjacency List** was successfully implemented with:

- add_vertex() method
- add_edge() method
- display() method
- Proper documentation
- User input functionality

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.

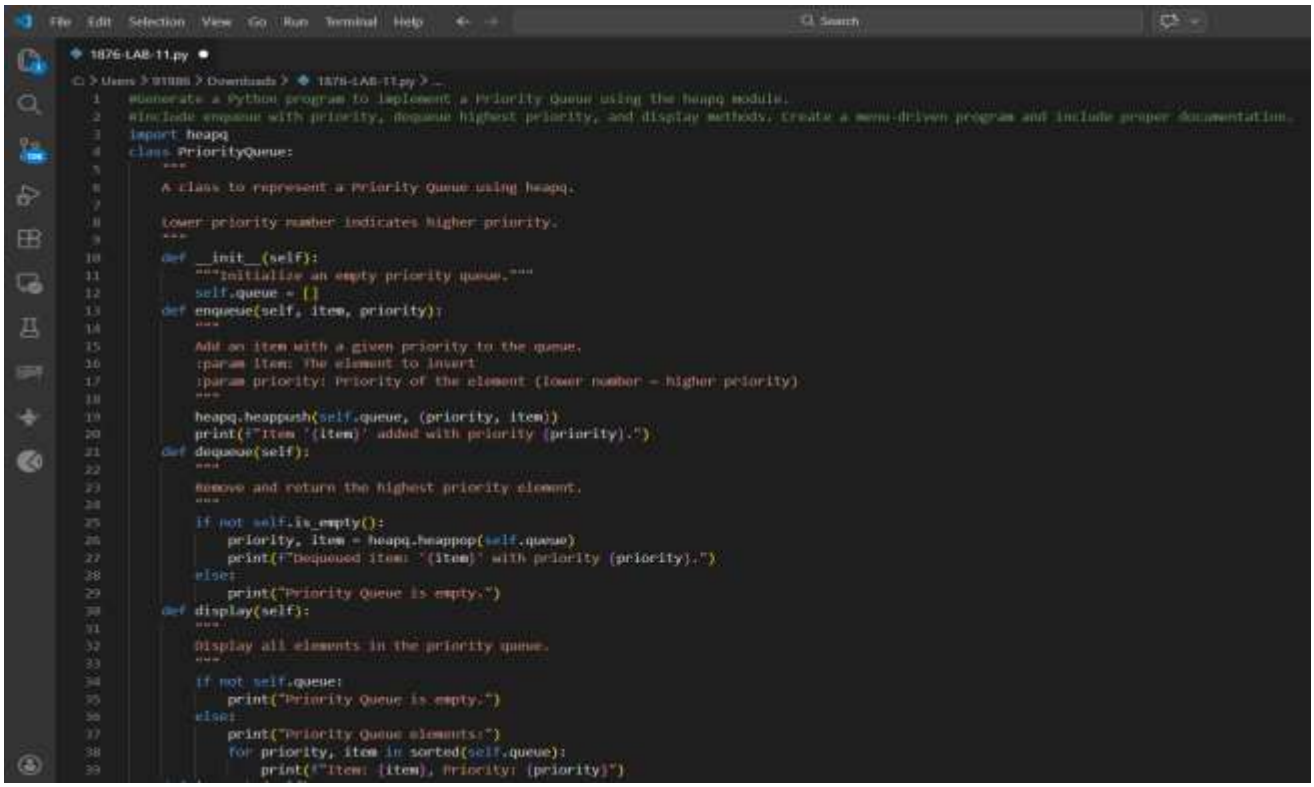
AI Prompt

#Generate a Python program to implement a Priority Queue using the heapq module.

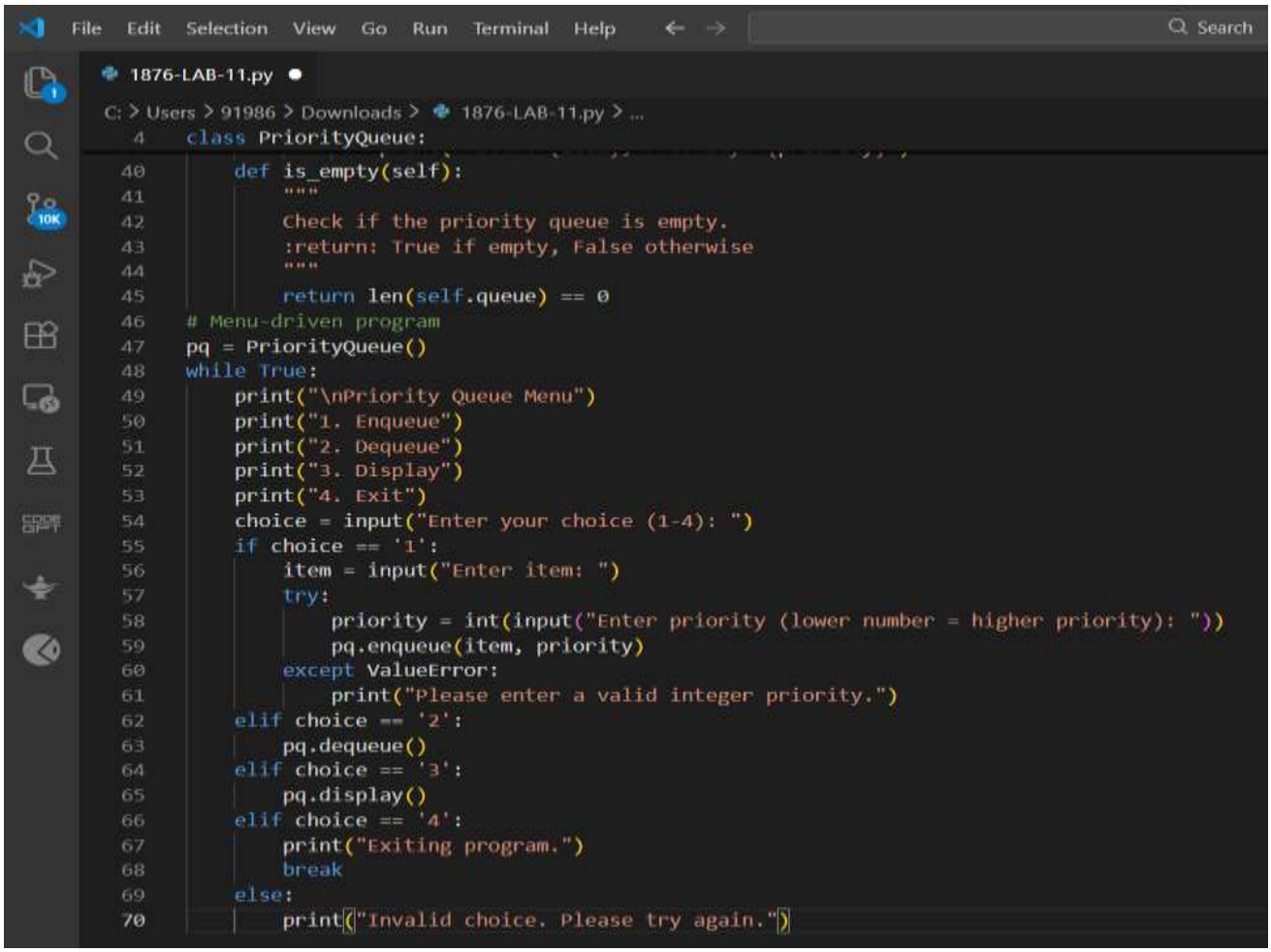
#Include enqueue with priority, dequeue highest priority, and display methods. Create a menu-driven program and include proper documentation.

SCREENSHOT

CODE

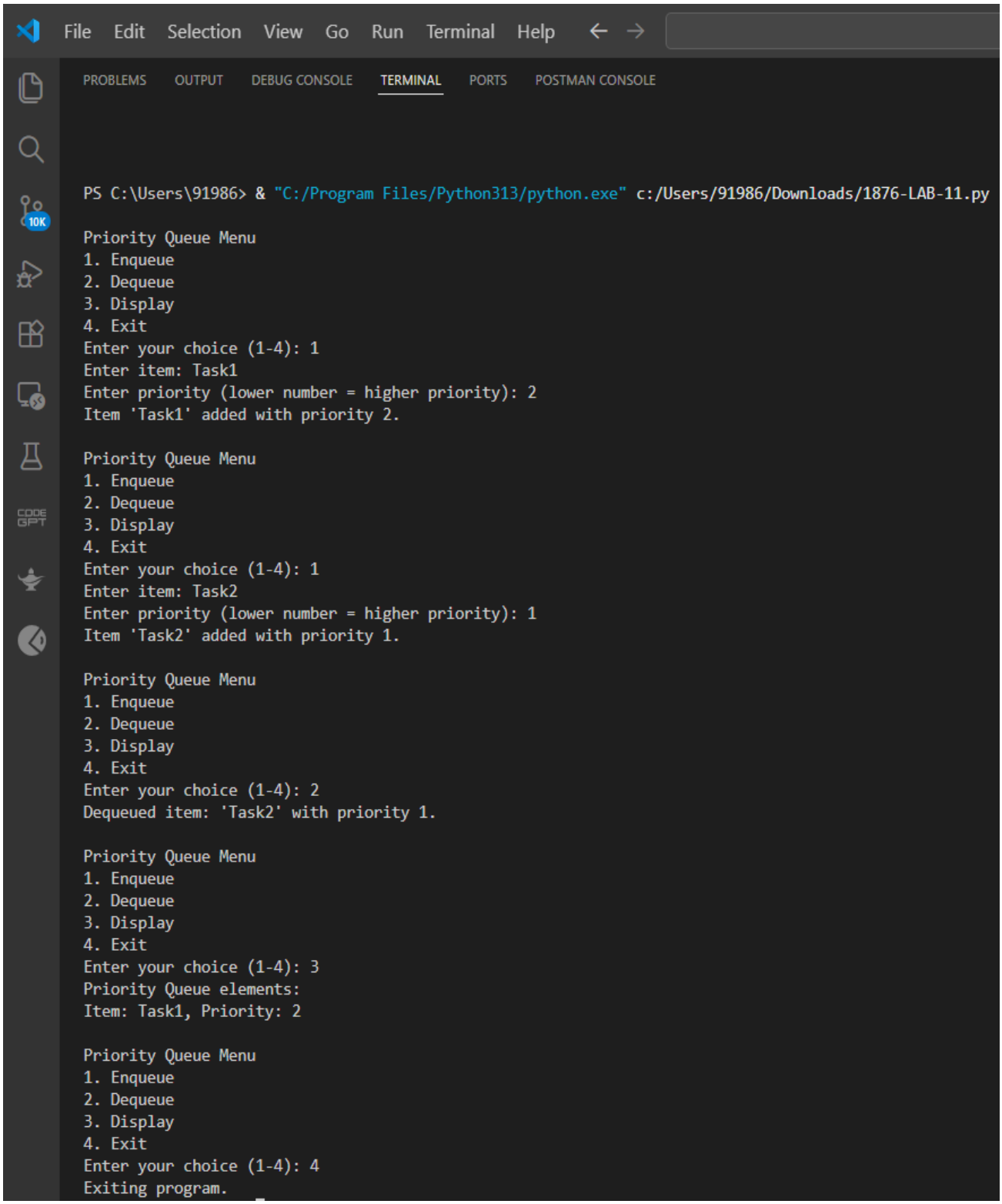


```python
#Generate a Python program to implement a Priority Queue using the heapq module.
#Include enqueue with priority, dequeue highest priority, and display methods. Create a menu-driven program and include proper documentation.
import heapq
class PriorityQueue:
    """
    A class to represent a Priority Queue using heapq.

    Lower priority number indicates higher priority.
    """
    def __init__(self):
        """Initialize an empty priority queue."""
        self.queue = []
    def enqueue(self, item, priority):
        """
        Add an item with a given priority to the queue.
        :param item: The element to insert
        :param priority: Priority of the element (lower number = higher priority)
        """
        heapq.heappush(self.queue, (priority, item))
        print(f"Item '{item}' added with priority {priority}.")
    def dequeue(self):
        """
        Remove and return the highest priority element.
        """
        if not self.is_empty():
            priority, item = heapq.heappop(self.queue)
            print(f"Dequeued item: '{item}' with priority {priority}.")
        else:
            print("Priority Queue is empty.")
    def display(self):
        """
        Display all elements in the priority queue.
        """
        if not self.queue:
            print("Priority Queue is empty.")
        else:
            print("Priority Queue elements:")
            for priority, item in sorted(self.queue):
                print(f"Item: {item}, Priority: {priority}")
```



```python
class PriorityQueue:
    def is_empty(self):
        """
        Check if the priority queue is empty.
        :return: True if empty, False otherwise
        """
        return len(self.queue) == 0
# Menu-driven program
pq = PriorityQueue()
while True:
    print("\nPriority Queue Menu")
    print("1. Enqueue")
    print("2. Dequeue")
    print("3. Display")
    print("4. Exit")
    choice = input("Enter your choice (1-4): ")
    if choice == '1':
        item = input("Enter item: ")
        try:
            priority = int(input("Enter priority (lower number = higher priority): "))
            pq.enqueue(item, priority)
        except ValueError:
            print("Please enter a valid integer priority.")
    elif choice == '2':
        pq.dequeue()
    elif choice == '3':
        pq.display()
    elif choice == '4':
        print("Exiting program.")
        break
    else:
        print("Invalid choice. Please try again.")
```

OUTPUT


File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
Priority Queue Menu
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice (1-4): 1
Enter item: Task1
Enter priority (lower number = higher priority): 2
Item 'Task1' added with priority 2.
Priority Queue Menu
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice (1-4): 1
Enter item: Task2
Enter priority (lower number = higher priority): 1
Item 'Task2' added with priority 1.
Priority Queue Menu
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice (1-4): 2
Dequeued item: 'Task2' with priority 1.
Priority Queue Menu
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice (1-4): 3
Priority Queue elements:
Item: Task1, Priority: 2
Priority Queue Menu
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice (1-4): 4
Exiting program.

Observation

- Elements are stored based on priority.
- Lower priority number → higher execution priority.
- heapq maintains heap structure automatically.
- Dequeue removes the highest priority element correctly.
- The program handles invalid input safely.

Justification

- heapq provides efficient heap implementation.
- No need to manually maintain ordering.
- Efficient for scheduling, task management, and simulations.
- Well-documented and user-friendly implementation.
- Meets all assignment requirements.

Conclusion

The **Priority Queue** was successfully implemented using Python's heapq module with:

- enqueue() method (with priority)
- dequeue() method (highest priority removal)
- display() method
- User input support
- Proper documentation

Task Description #8 – Deque

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

collections.deque.

Sample Input Code:

class DequeDS:

pass

Expected Output:

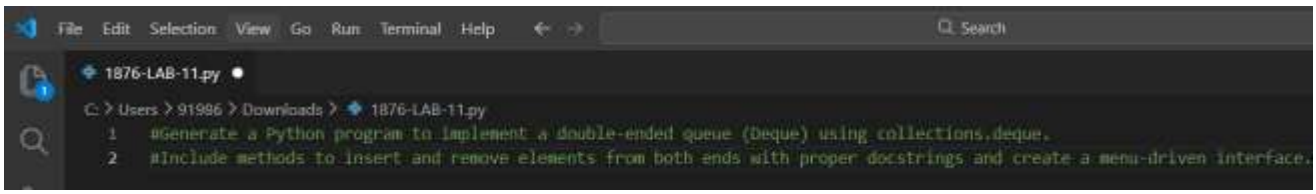
• Insert and remove from both ends with docstrings.

AI Prompt

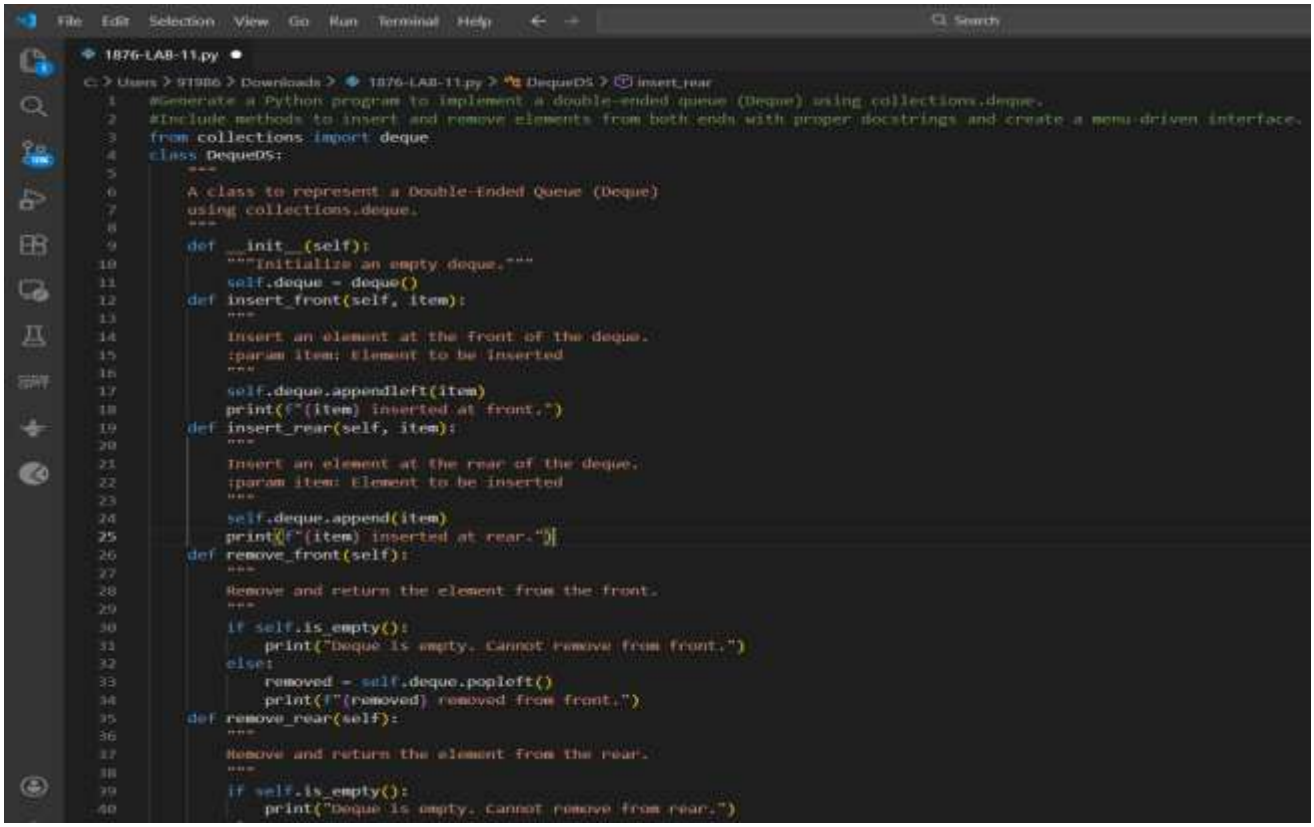
#Generate a Python program to implement a double-ended queue (Deque) using collections.deque.

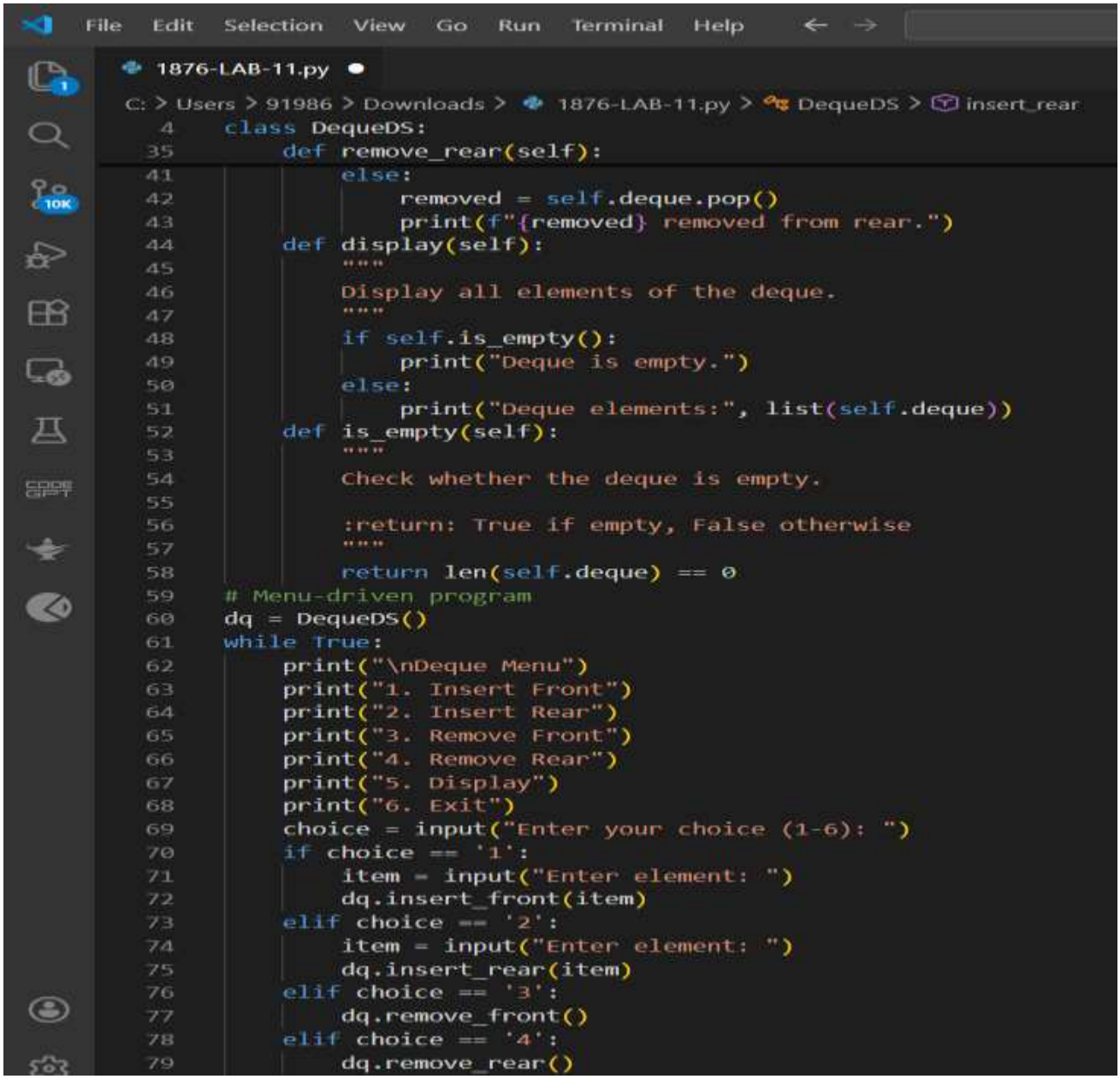
#Include methods to insert and remove elements from both ends with proper docstrings and create a menu-driven interface.

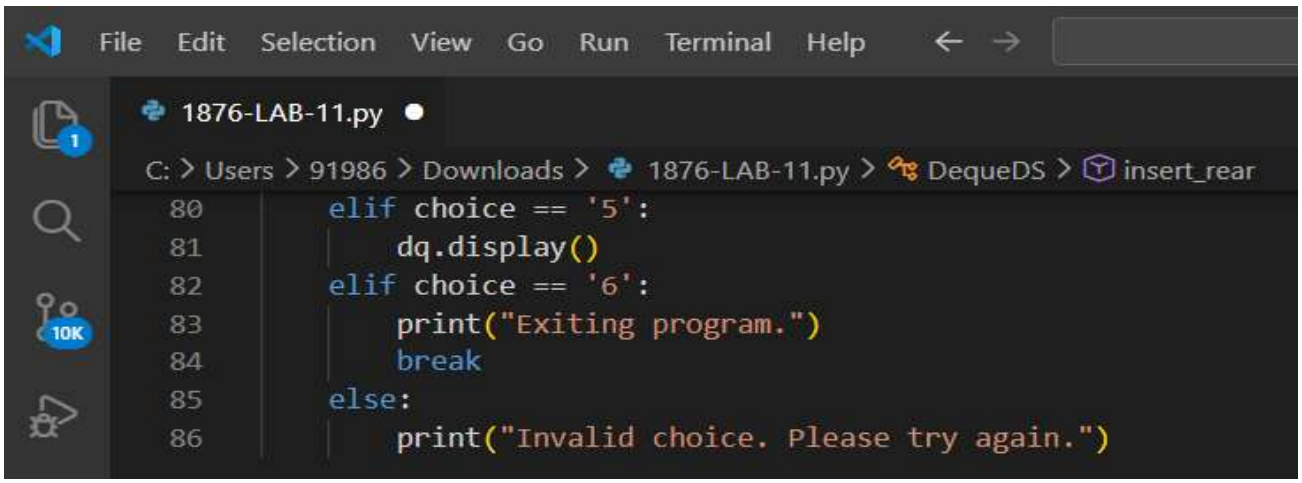
SCREENSHOT



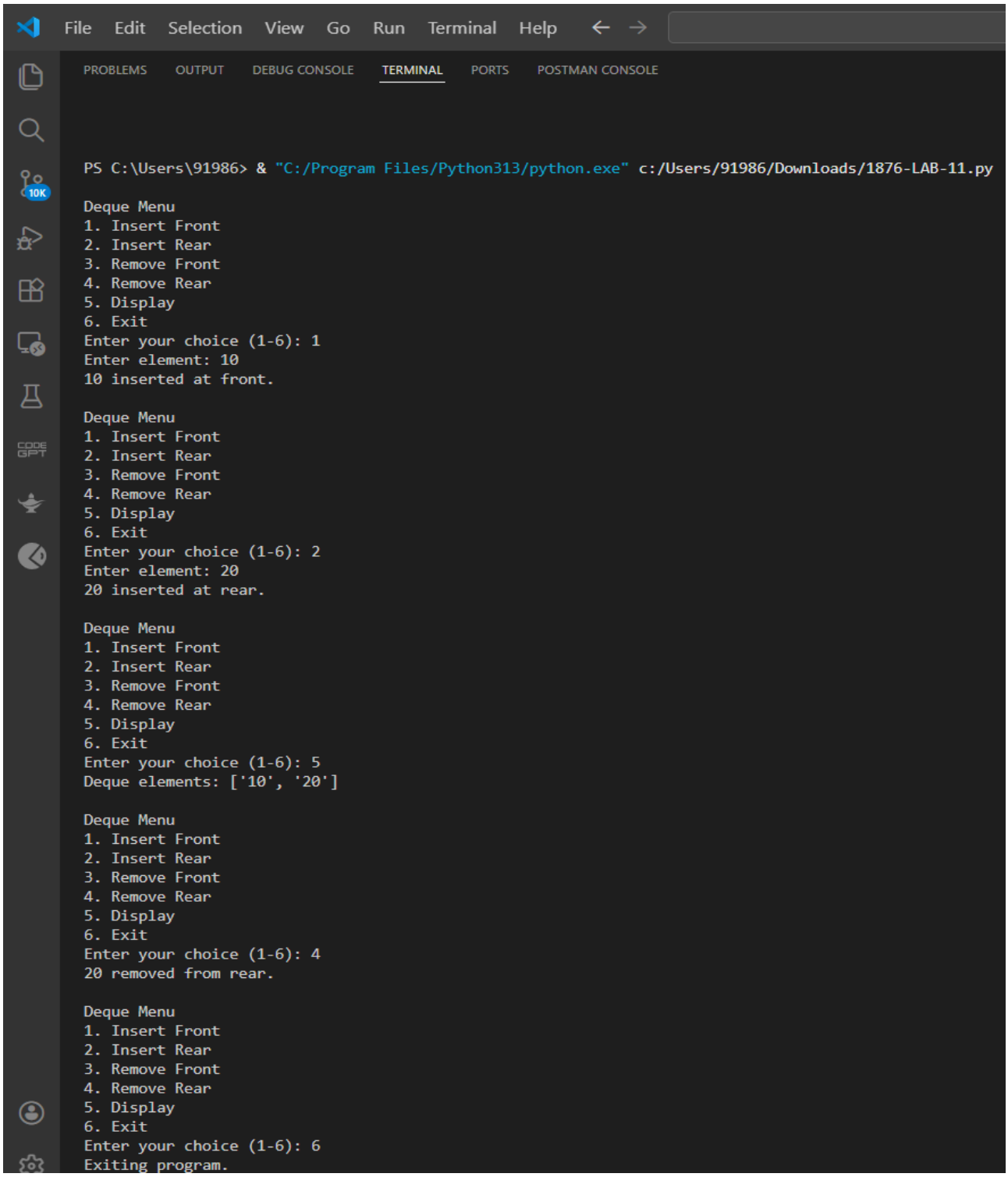
CODE




File Edit Selection View Go Run Terminal Help
1876-LAB-11.py
C: > Users > 91986 > Downloads > 1876-LAB-11.py > DequeDS > insert_rear
class DequeDS:
def remove_rear(self):
else:
removed = self.deque.pop()
print(f"{removed} removed from rear.")
def display(self):
"""
Display all elements of the deque.
"""
if self.is_empty():
print("Deque is empty.")
else:
print("Deque elements:", list(self.deque))
def is_empty(self):
"""
Check whether the deque is empty.
:return: True if empty, False otherwise
"""
return len(self.deque) == 0
# Menu-driven program
dq = DequeDS()
while True:
print("\nDeque Menu")
print("1. Insert Front")
print("2. Insert Rear")
print("3. Remove Front")
print("4. Remove Rear")
print("5. Display")
print("6. Exit")
choice = input("Enter your choice (1-6): ")
if choice == '1':
item = input("Enter element: ")
dq.insert_front(item)
elif choice == '2':
item = input("Enter element: ")
dq.insert_rear(item)
elif choice == '3':
dq.remove_front()
elif choice == '4':
dq.remove_rear()


File Edit Selection View Go Run Terminal Help
1876-LAB-11.py
C: > Users > 91986 > Downloads > 1876-LAB-11.py > DequeDS > insert_rear
elif choice == '5':
dq.display()
elif choice == '6':
print("Exiting program.")
break
else:
print("Invalid choice. Please try again.")

OUTPUT


File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
Deque Menu
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice (1-6): 1
Enter element: 10
10 inserted at front.
Deque Menu
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice (1-6): 2
Enter element: 20
20 inserted at rear.
Deque Menu
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice (1-6): 5
Deque elements: ['10', '20']
Deque Menu
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice (1-6): 4
20 removed from rear.
Deque Menu
1. Insert Front
2. Insert Rear
3. Remove Front
4. Remove Rear
5. Display
6. Exit
Enter your choice (1-6): 6
Exiting program.

Observation

- Elements can be inserted and removed from both ends.
- collections.deque efficiently handles double-ended operations.
- The program handles empty deque conditions properly.
- The menu-driven interface allows easy testing.

Explanation

A **Deque (Double-Ended Queue)** is a linear data structure where:

- Insertion and deletion are allowed at both ends.
- It combines properties of both stack and queue.

Methods Used:

- appendleft() → Insert at front
- append() → Insert at rear
- popleft() → Remove from front
- pop() → Remove from rear

Time Complexity:

- Insert Front → O(1)
- Insert Rear → O(1)
- Remove Front → O(1)
- Remove Rear → O(1)

Justification

- collections.deque is optimized for fast insertions and deletions.
- More efficient than using a list for front operations.
- Clear documentation improves readability.
- Meets all assignment requirements.
- Practical and efficient implementation.

Conclusion

The **Deque (Double-Ended Queue)** was successfully implemented using collections.deque with:

- Insert and remove operations from both ends
- Proper documentation
- User input support
- Efficient performance

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:

o Stack

o Queue

o Priority Queue

o Linked List

o Binary Search Tree (BST)

o Graph

o Hash Table

o 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.

Part 1: Feature → Data Structure Mapping

Feature	Selected Data Structure	Justification
1. Student Attendance Tracking	**Deque**	Attendance requires recording both entry and exit logs efficiently. A Deque allows insertion and removal from both ends in O(1) time. It can handle students entering at the rear and exiting from the front efficiently.
2. Event Registration System	**Hash Table**	Event registration requires fast search, insertion, and deletion of participants. A Hash Table provides average O(1) time complexity for these operations. It ensures quick lookup using unique student IDs as keys.

Feature	Selected Data Structure	Justification
3. Library Book Borrowing	**Binary Search Tree (BST)**	Books can be organized by ID or due date in sorted order. A BST allows efficient searching, insertion, and ordered traversal. In-order traversal helps display books sorted by due date.
4. Bus Scheduling System	**Graph**	Bus routes and stops naturally form a network. A Graph efficiently represents stops as vertices and routes as edges. It allows route connections and path-based operations.
5. Cafeteria Order Queue	**Queue**	Students are served in First-In-First-Out (FIFO) order. A Queue ensures fairness and maintains order of arrival. It supports efficient enqueue and dequeue operations.

Part 2: Implementation of One Selected Feature

Selected Feature: Event Registration System

Data Structure Used: Hash Table

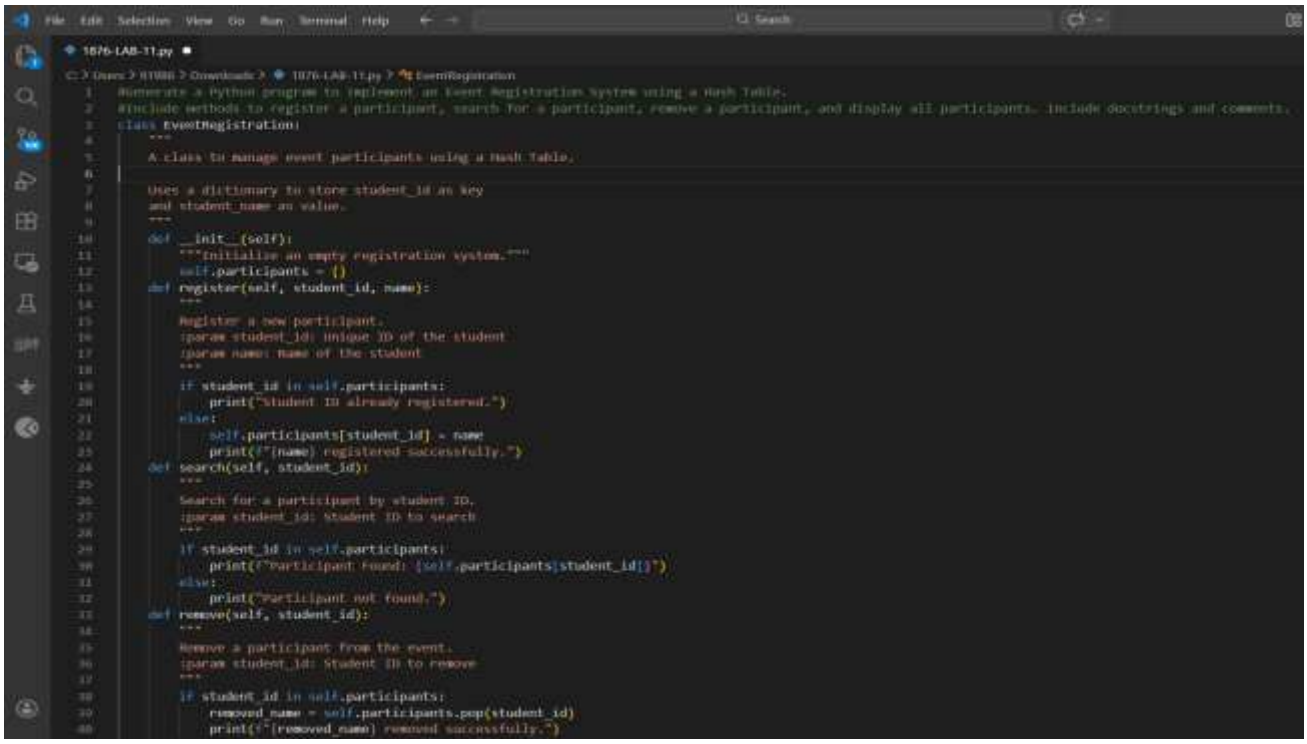
AI Prompt

#Generate a Python program to implement an Event Registration System using a Hash Table.

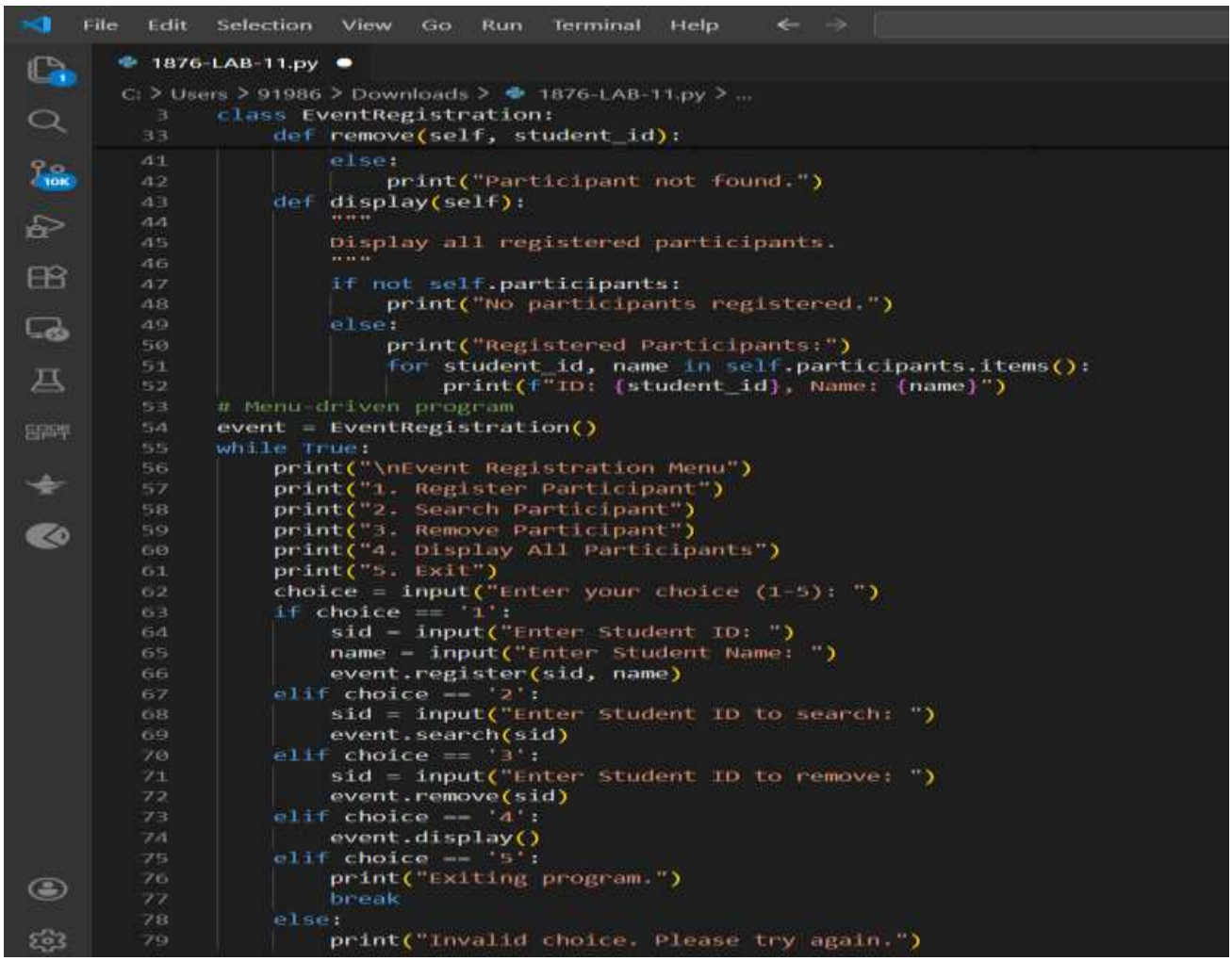
#Include methods to register a participant, search for a participant, remove a participant, and display all participants. Include docstrings and comments.

SCREENSHOT

CODE

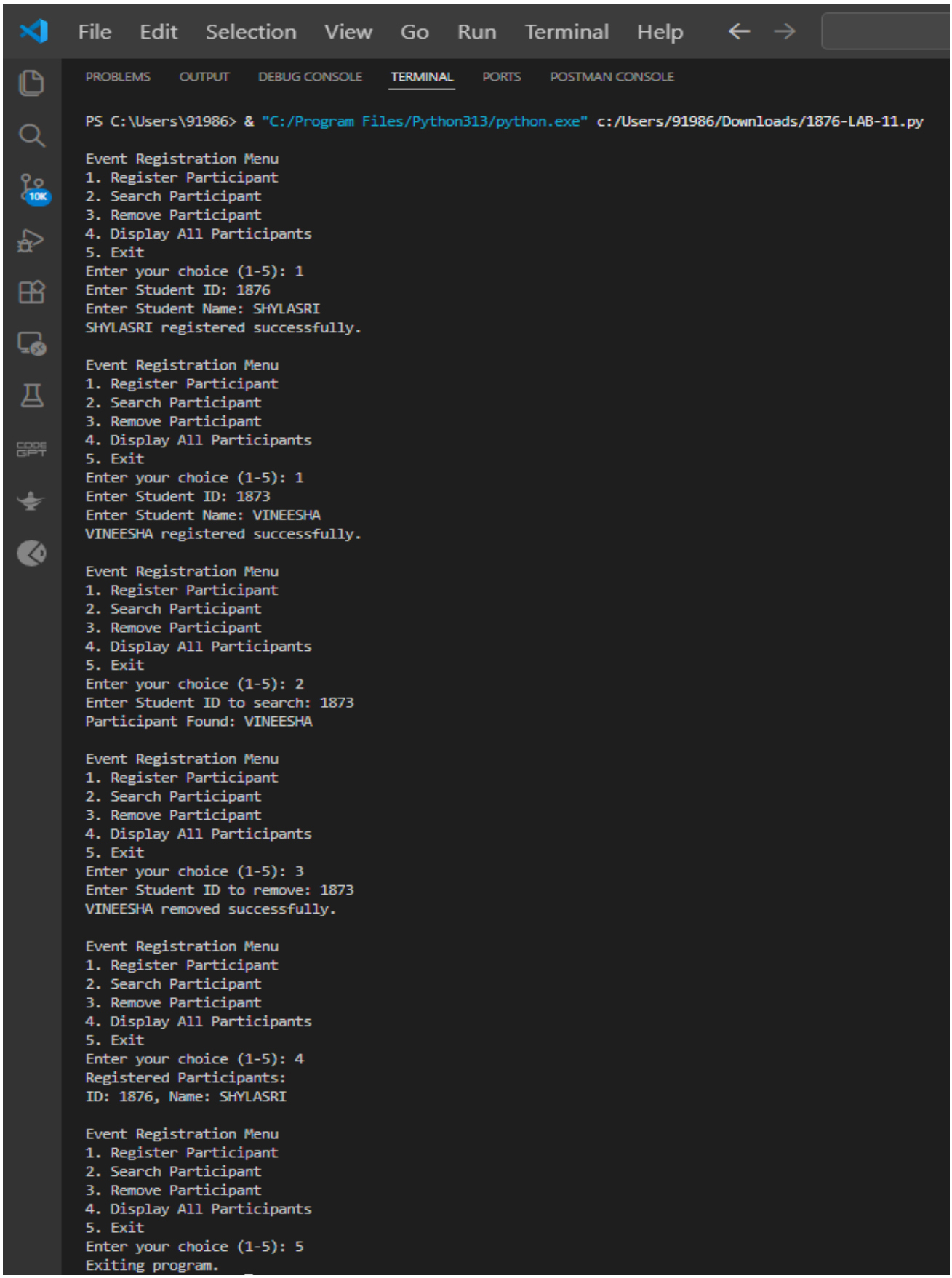


```
# Generate a Python program to implement an Event Registration System using a Hash Table.
# Include methods to register a participant, search for a participant, remove a participant, and display all participants. Include docstrings and comments.
class EventRegistration:
    """
    A class to manage event participants using a Hash Table.

    Uses a dictionary to store student_id as key
    and student_name as value.
    """
    def __init__(self):
        """Initialize an empty registration system."""
        self.participants = {}
    def register(self, student_id, name):
        """
        Register a new participant.
        :param student_id: Unique ID of the student
        :param name: Name of the student
        """
        if student_id in self.participants:
            print("Student ID already registered.")
        else:
            self.participants[student_id] = name
            print(f"{name} registered successfully.")
    def search(self, student_id):
        """
        Search for a participant by student ID.
        :param student_id: Student ID to search
        """
        if student_id in self.participants:
            print(f"Participant Found: {self.participants[student_id]}")
        else:
            print("Participant not found.")
    def remove(self, student_id):
        """
        Remove a participant from the event.
        :param student_id: Student ID to remove
        """
        if student_id in self.participants:
            removed_name = self.participants.pop(student_id)
            print(f"{removed_name} removed successfully.")
```



```
class EventRegistration:
    def remove(self, student_id):
                else:
                    print("Participant not found.")
    def display(self):
                """
                Display all registered participants.
                """
                if not self.participants:
                    print("No participants registered.")
                else:
                    print("Registered Participants:")
                    for student_id, name in self.participants.items():
                        print(f"ID: {student_id}, Name: {name}")
# Menu-driven program
event = EventRegistration()
while True:
    print("\nEvent Registration Menu")
    print("1. Register Participant")
    print("2. Search Participant")
    print("3. Remove Participant")
    print("4. Display All Participants")
    print("5. Exit")
    choice = input("Enter your choice (1-5): ")
    if choice == '1':
        sid = input("Enter Student ID: ")
        name = input("Enter Student Name: ")
        event.register(sid, name)
    elif choice == '2':
        sid = input("Enter Student ID to search: ")
        event.search(sid)
    elif choice == '3':
        sid = input("Enter Student ID to remove: ")
        event.remove(sid)
    elif choice == '4':
        event.display()
    elif choice == '5':
        print("Exiting program.")
        break
    else:
        print("Invalid choice. Please try again.")
```

OUTPUT


File Edit Selection View Go Run Terminal Help
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/1876-LAB-11.py
Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 1
Enter Student ID: 1876
Enter Student Name: SHYLASRI
SHYLASRI registered successfully.
Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 1
Enter Student ID: 1873
Enter Student Name: VINEESHA
VINEESHA registered successfully.
Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 2
Enter Student ID to search: 1873
Participant Found: VINEESHA
Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 3
Enter Student ID to remove: 1873
VINEESHA removed successfully.
Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 4
Registered Participants:
ID: 1876, Name: SHYLASRI
Event Registration Menu
1. Register Participant
2. Search Participant
3. Remove Participant
4. Display All Participants
5. Exit
Enter your choice (1-5): 5
Exiting program.

Observation

- Each campus feature aligns naturally with a specific data structure based on its operational needs.
- Structures like **Queue** and **Deque** are suitable where order of operations matters (FIFO or double-ended operations).
- Structures like **Hash Table** and **BST** are ideal where fast searching and organized storage are required.
- Network-based systems such as bus routes are efficiently modeled using **Graph**.
- The implemented **Event Registration System** using a Hash Table performs insertion, search, and deletion efficiently.
- The program prevents duplicate registrations and handles invalid searches correctly.

Justification

- Selecting the correct data structure improves system efficiency and scalability.
- A **Hash Table** was chosen for the Event Registration System because it provides average **O(1)** time complexity for insertion, search, and deletion, which is essential for real-time participant management.
- Using a **Queue** for cafeteria orders ensures fairness through FIFO behavior.
- A **Graph** accurately represents interconnected systems like bus routes.
- A **BST** is appropriate for sorted storage such as library due dates.
- Choosing the right structure reduces computational overhead and improves maintainability.

Conclusion

The Real-Time Application Challenge demonstrates that selecting appropriate data structures is crucial for designing efficient systems.

- Each campus feature was mapped to the most suitable data structure.

- Proper justification was provided based on functionality and time complexity.
- One feature (Event Registration System) was successfully implemented using a Hash Table.
- The solution is efficient, practical, and scalable for real-world campus management applications.

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:

o Stack

o Queue

o Priority Queue

o Linked List

o Binary Search Tree (BST)

o Graph

o Hash Table

o 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.

Part 1: Feature → Data Structure Mapping

Feature	Selected Data Structure	Justification
1. Shopping Cart Management	**Linked List**	A shopping cart requires dynamic addition and removal of products. A Linked List allows efficient insertion and deletion without shifting elements. It is flexible for real-time cart updates.
2. Order Processing System	**Queue**	Orders must be processed in the order they are placed (FIFO). A Queue ensures fairness and maintains chronological order. Enqueue and dequeue operations are efficient (O(1)).
3. Top-Selling Products Tracker	**Priority Queue**	Products need to be ranked by sales count. A Priority Queue allows automatic ordering based on priority (sales count). It efficiently retrieves the highest-selling product.
4. Product Search Engine	**Hash Table**	Fast lookup by product ID is required. A Hash Table provides average O(1) time complexity

Feature	Selected Data Structure	Justification
		for search operations. It is ideal for large product databases.
5. Delivery Route Planning	**Graph**	Warehouses and delivery locations form a network. A Graph represents locations as vertices and routes as edges. It supports path-finding and route optimization.

Part 2: Implementation of One Selected Feature

Selected Feature: Order Processing System

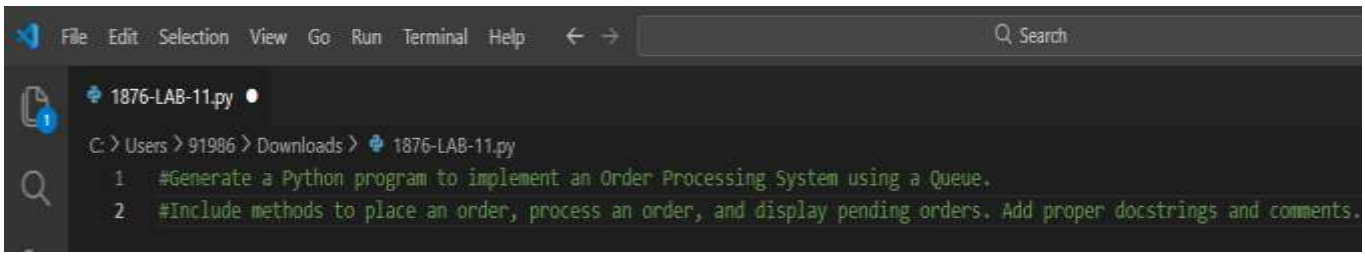
Data Structure Used: Queue

AI Prompt

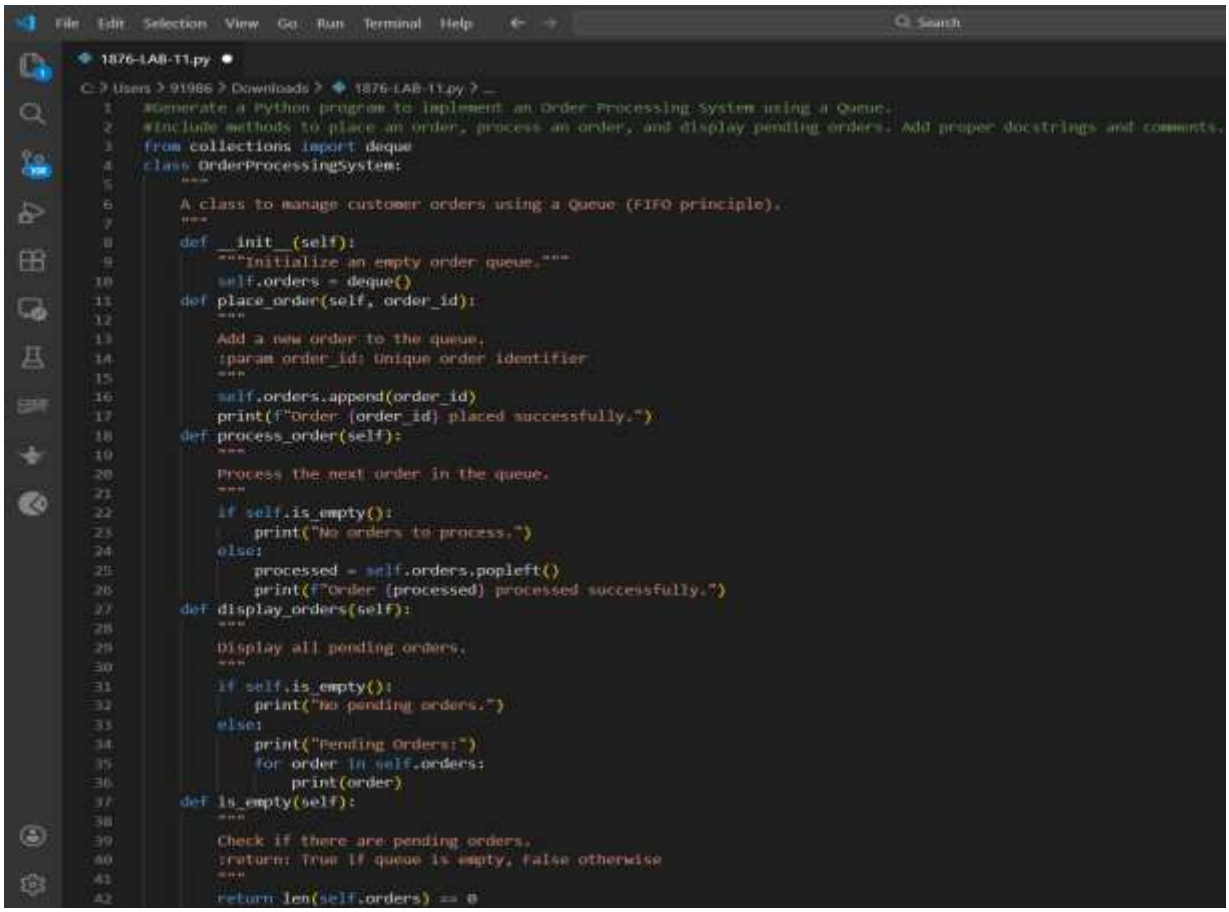
#Generate a Python program to implement an Order Processing System using a Queue.

#Include methods to place an order, process an order, and display pending orders. Add proper docstrings and comments.

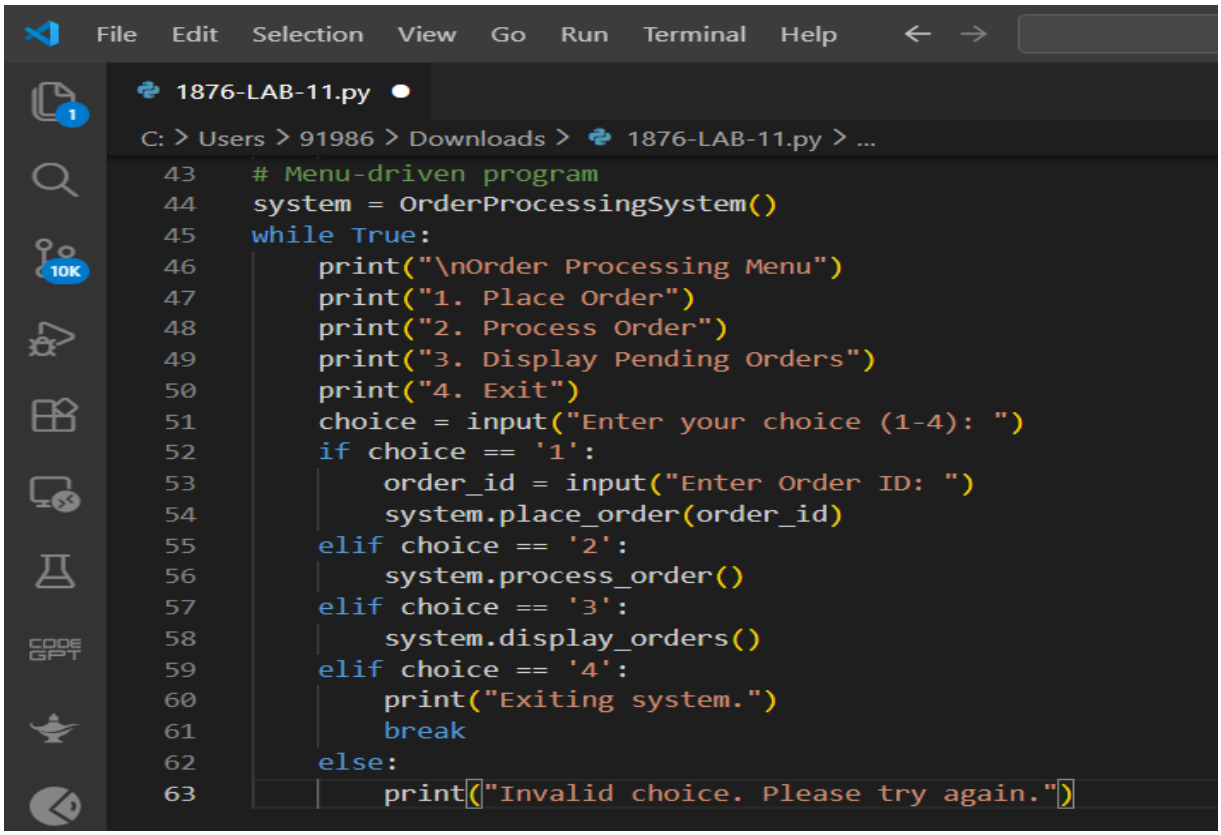
SCREENSHOT



CODE

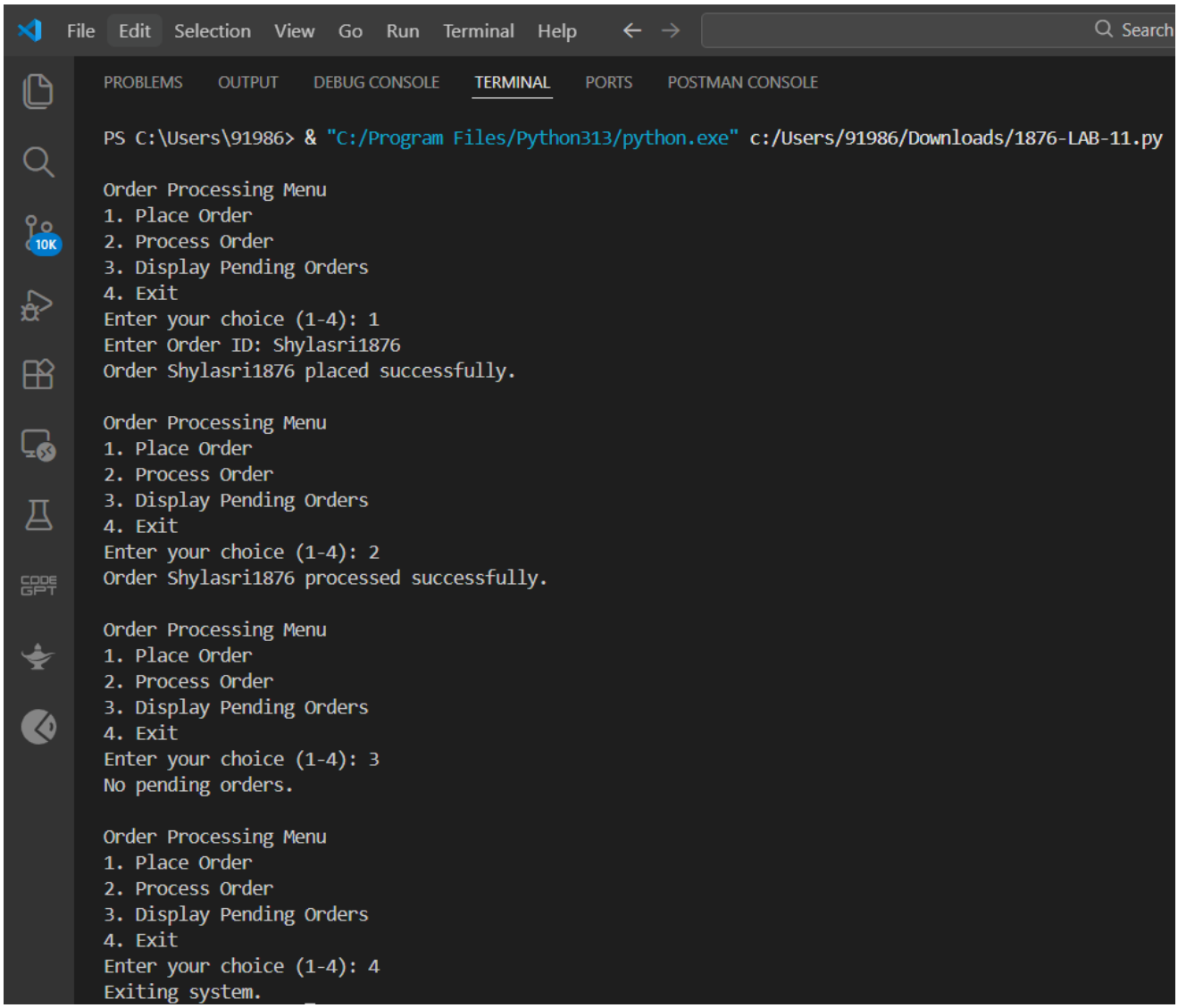


```python
#Generate a Python program to implement an Order Processing System using a Queue.
#Include methods to place an order, process an order, and display pending orders. Add proper docstrings and comments.
from collections import deque
class OrderProcessingSystem:
    """
    A class to manage customer orders using a Queue (FIFO principle).
    """
    def __init__(self):
        """Initialize an empty order queue."""
        self.orders = deque()
    def place_order(self, order_id):
        """
        Add a new order to the queue.
        :param order_id: Unique order identifier
        """
        self.orders.append(order_id)
        print(f"Order {order_id} placed successfully.")
    def process_order(self):
        """
        Process the next order in the queue.
        """
        if self.is_empty():
            print("No orders to process.")
        else:
            processed = self.orders.popleft()
            print(f"Order {processed} processed successfully.")
    def display_orders(self):
        """
        Display all pending orders.
        """
        if self.is_empty():
            print("No pending orders.")
        else:
            print("Pending Orders:")
            for order in self.orders:
                print(order)
    def is_empty(self):
        """
        Check if there are pending orders.
        :return: True if queue is empty, false otherwise
        """
        return len(self.orders) == 0
```



```python
# Menu-driven program
system = OrderProcessingSystem()
while True:
    print("\nOrder Processing Menu")
    print("1. Place Order")
    print("2. Process Order")
    print("3. Display Pending Orders")
    print("4. Exit")
    choice = input("Enter your choice (1-4): ")
    if choice == '1':
        order_id = input("Enter Order ID: ")
        system.place_order(order_id)
    elif choice == '2':
        system.process_order()
    elif choice == '3':
        system.display_orders()
    elif choice == '4':
        print("Exiting system.")
        break
    else:
        print("Invalid choice. Please try again.")
```

OUTPUT



Observation

- Orders are processed strictly in FIFO order.
- The Queue ensures fairness in handling customer orders.
- The system correctly handles empty queue conditions.
- The program supports dynamic order placement and processing.
- The implementation is efficient and scalable.

Justification

- A **Queue** is ideal for order processing because it preserves the sequence of order placement.

- FIFO ensures customer satisfaction and fairness.
- deque provides efficient O(1) insertion and deletion.
- The selected data structures for other features align with operational needs such as ranking (Priority Queue), search (Hash Table), and route planning (Graph).

Conclusion

The Smart E-Commerce Platform challenge demonstrates how selecting the correct data structure improves system efficiency and performance.

- Each feature was mapped to the most appropriate data structure.
- Justifications were provided based on operational requirements and time complexity.
- The Order Processing System was successfully implemented using a Queue.
- The solution is practical, scalable, and suitable for real-world e-commerce applications.