ASSIGNMENT-11.1

Name :N.Soumya

H NO:2303A51869

Batch-13

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

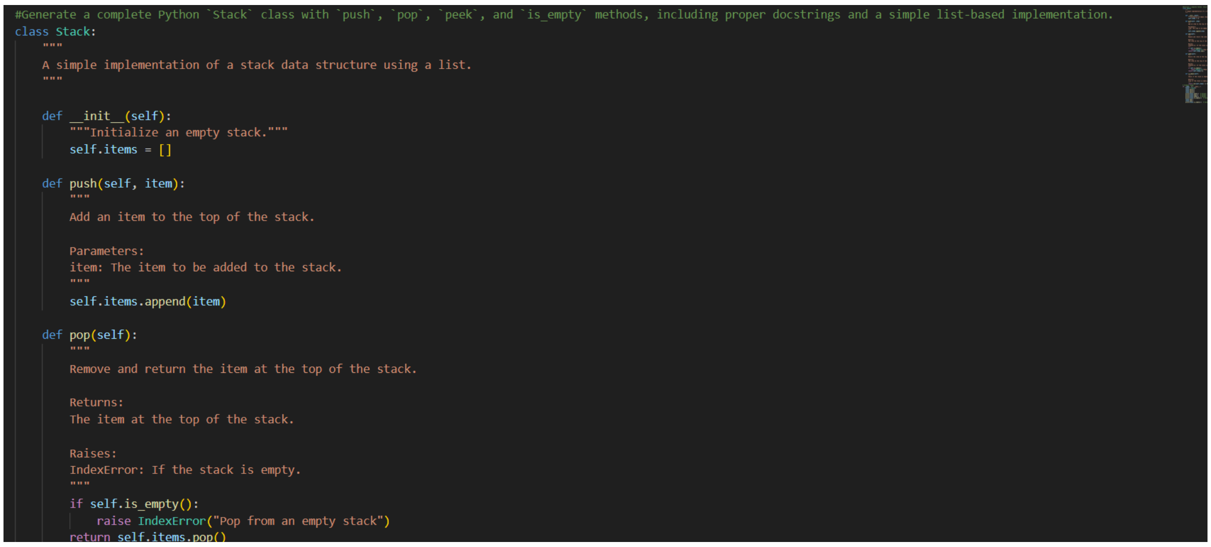
Expected Output:

• A functional stack implementation with all required methods and docstrings.

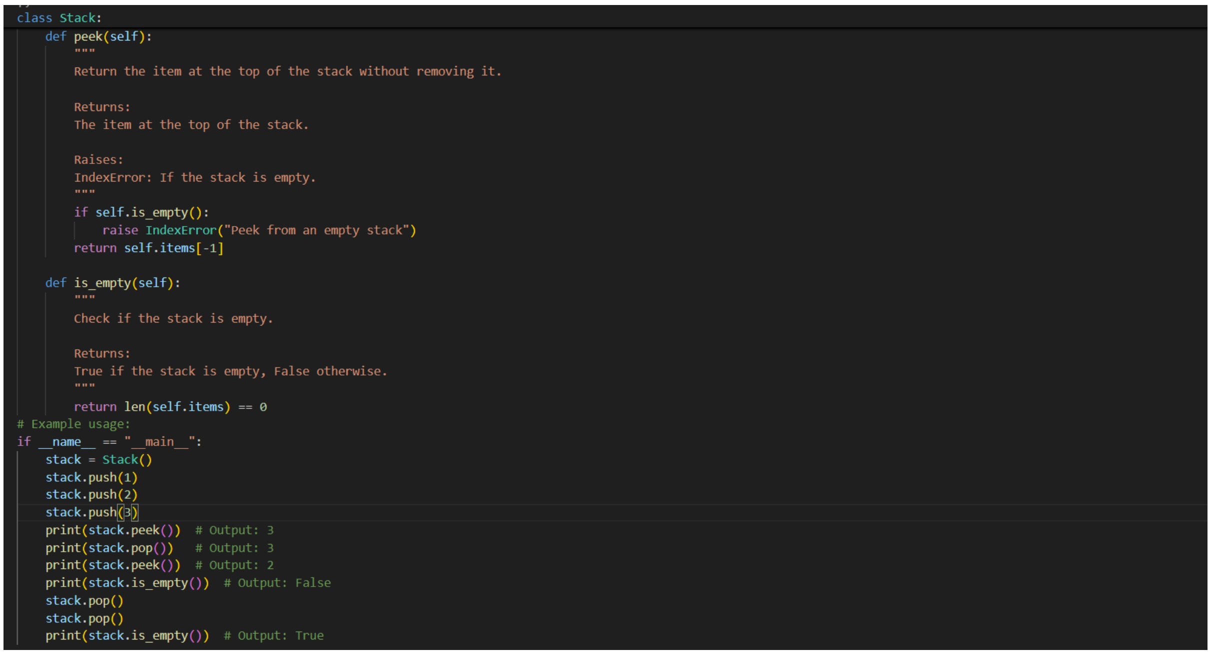
PROMPT:

Generate a complete Python Stack class with push, pop, peek, and is_empty methods, including proper docstrings and a simple list-based implementation.

CODE:

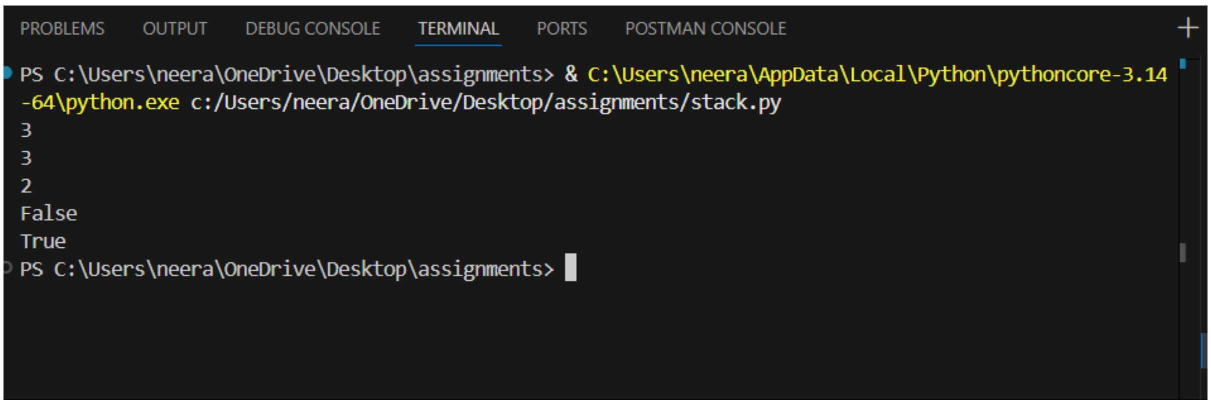


```python
#Generate a complete Python `Stack` class with `push`, `pop`, `peek`, and `is_empty` methods, including proper docstrings and a simple list-based implementation.
class Stack:
    """
    A simple implementation of a stack data structure using a list.
    """

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

    def push(self, item):
        """
        Add an item to the top of the stack.

        Parameters:
        item: The item to be added to the stack.
        """
        self.items.append(item)

    def pop(self):
        """
        Remove and return the item at the top of the stack.

        Returns:
        The item at the top of the stack.

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



```python
class Stack:
    def peek(self):
        """
        Return the item at the top of the stack without removing it.

        Returns:
        The item at the top of the stack.

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

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

        Returns:
        True if the stack is empty, False otherwise.
        """
        return len(self.items) == 0
# Example usage:
if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(stack.peek())  # Output: 3
    print(stack.pop())   # Output: 3
    print(stack.peek())  # Output: 2
    print(stack.is_empty())  # Output: False
    stack.pop()
    stack.pop()
    print(stack.is_empty())  # Output: True
```

OUTPUT:



```
PS C:\Users\neera\OneDrive\Desktop\assignments> & C:\Users\neera\AppData\Local\Python\pythoncore-3.14
-64\python.exe c:/Users/neera/OneDrive/Desktop/assignments/stack.py
3
3
2
False
True
PS C:\Users\neera\OneDrive\Desktop\assignments>
```

#JUSTIFICATION

The Stack class uses a list to store its elements. The push method adds an item to the top, and the pop method removes and returns the top item. The peek method shows the top element without removing it, while is_empty checks if the stack has any items. This makes the stack simple and efficient to use in Python.

Task Description #2 – Queue Implementation

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

Sample Input Code:

class Queue:

pass

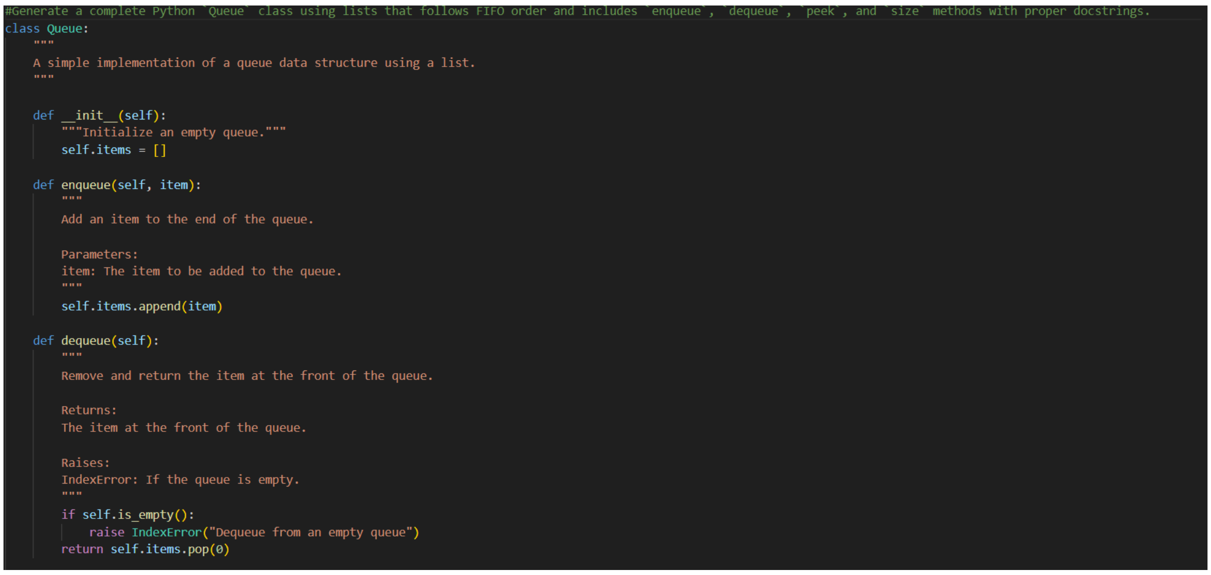
Expected Output:

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

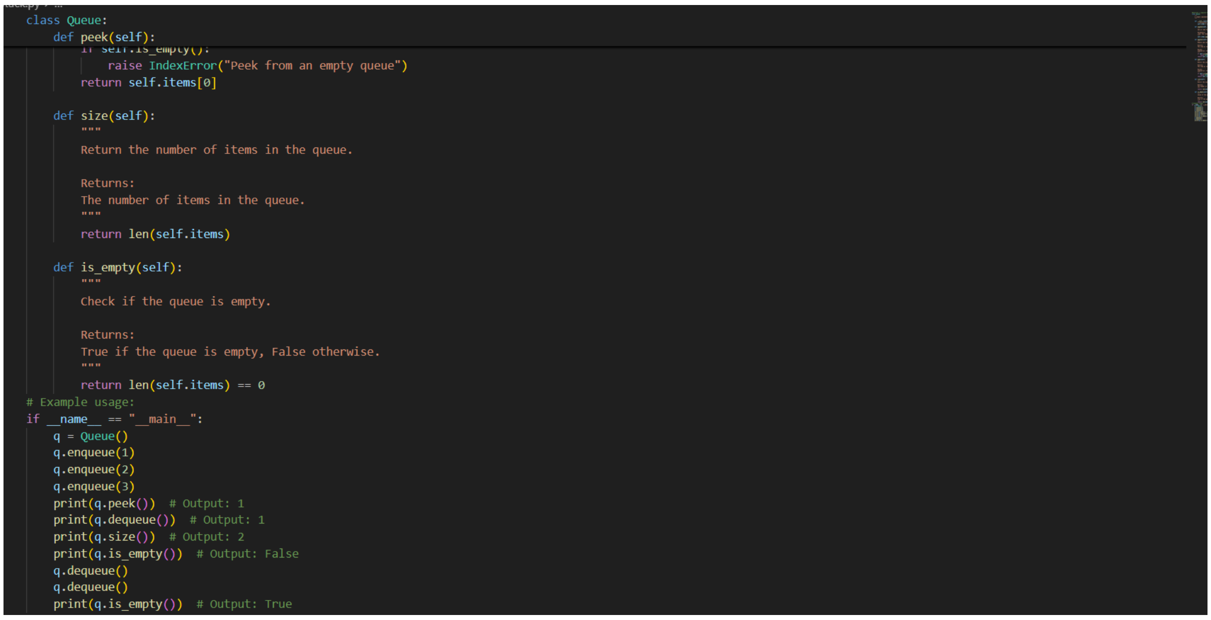
PROMPT:

Generate a complete Python Queue class using lists that follows FIFO order and includes enqueue, dequeue, peek, and size methods with proper docstrings.

CODE:

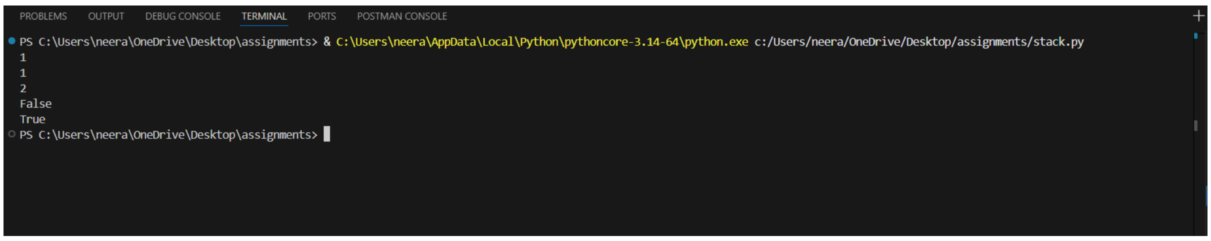


```python
#Generate a complete Python `Queue` class using lists that follows FIFO order and includes `enqueue`, `dequeue`, `peek`, and `size` methods with proper docstrings.
class Queue:
    """
    A simple implementation of a queue data structure using a list.
    """

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

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

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

    def dequeue(self):
        """
        Remove and return the item at the front of the queue.

        Returns:
        The item at the front of the queue.

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



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

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

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

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

        Returns:
        True if the queue is empty, False otherwise.
        """
        return len(self.items) == 0
# Example usage:
if __name__ == "__main__":
    q = Queue()
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)
    print(q.peek())  # Output: 1
    print(q.dequeue())  # Output: 1
    print(q.size())  # Output: 2
    print(q.is_empty())  # Output: False
    q.dequeue()
    q.dequeue()
    print(q.is_empty())  # Output: True
```

OUTPUT:



```
PS C:\Users\neera\OneDrive\Desktop\assignments> & C:\Users\neera\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/neera/OneDrive/Desktop/assignments/stack.py
1
1
2
False
True
PS C:\Users\neera\OneDrive\Desktop\assignments>
```

#JUSTIFICATION

The enqueue method adds an item to the end, and dequeue removes the item from the front. The peek method lets you see the front item without removing it, while size returns the number of items in the queue. The is_empty method checks whether the queue has any elements to avoid errors.

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:

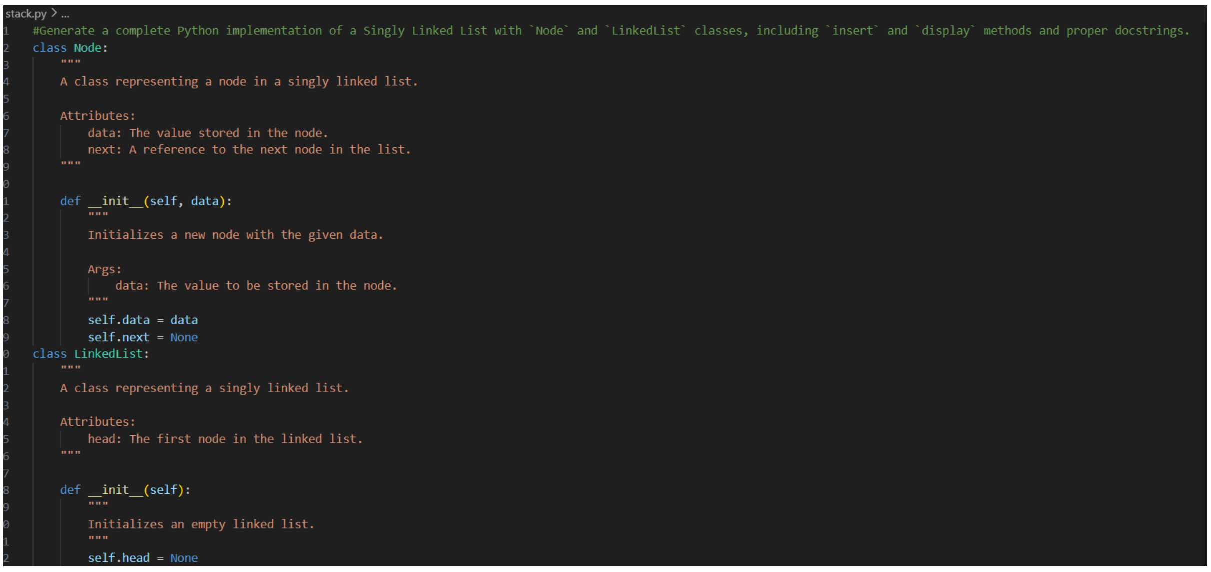
• A working linked list implementation with clear method

documentation.

PROMPT:

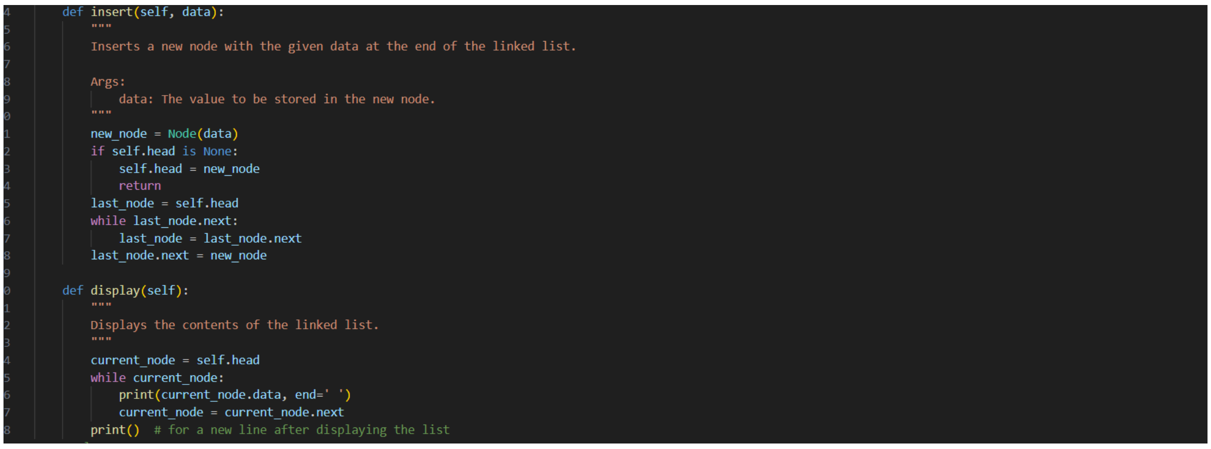
Generate a complete Python implementation of a Singly Linked List with Node and LinkedList classes, including insert and display methods and proper docstrings**.**

CODE

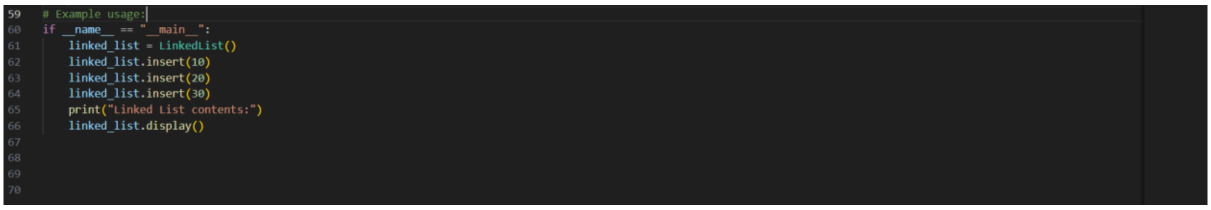


stack.py > ...

```python
#Generate a complete Python implementation of a Singly Linked List with `Node` and `LinkedList` classes, including `insert` and `display` methods and proper docstrings.
class Node:
    """
    A class representing a node in a singly linked list.

    Attributes:
        data: The value stored in the node.
        next: A reference to the next node in the list.
    """

    def __init__(self, data):
        """
        Initializes a new node with the given data.

        Args:
            data: The value to be stored in the node.
        """
        self.data = data
        self.next = None
class LinkedList:
    """
    A class representing a singly linked list.

    Attributes:
        head: The first node in the linked list.
    """

    def __init__(self):
        """
        Initializes an empty linked list.
        """
        self.head = None
```

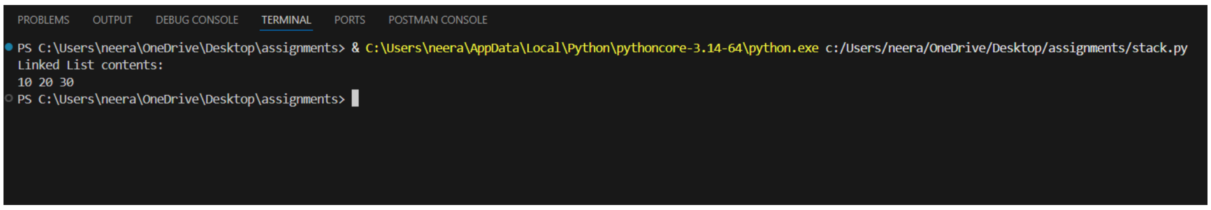


```python
    def insert(self, data):
        """
        Inserts a new node with the given data at the end of the linked list.

        Args:
            data: The value to be stored in the new node.
        """
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def display(self):
        """
        Displays the contents of the linked list.
        """
        current_node = self.head
        while current_node:
            print(current_node.data, end=' ')
            current_node = current_node.next
        print()  # for a new line after displaying the list
```



```python
# Example usage:
if __name__ == "__main__":
    linked_list = LinkedList()
    linked_list.insert(10)
    linked_list.insert(20)
    linked_list.insert(30)
    print("Linked List contents:")
    linked_list.display()
```

OUTPUT:



```
PS C:\Users\neera\OneDrive\Desktop\assignments> & C:\Users\neera\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/neera/OneDrive/Desktop/assignments/stack.py
Linked List contents:
10 20 30
PS C:\Users\neera\OneDrive\Desktop\assignments>
```

#JUSTIFICATION:

The implementation of the `Node` and `LinkedList` classes provides a basic structure for a singly linked list. The `Node` class encapsulates the data and the reference to the next node, while the `LinkedList` class manages the linked list, allowing for insertion of new nodes and displaying the contents of the list.

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

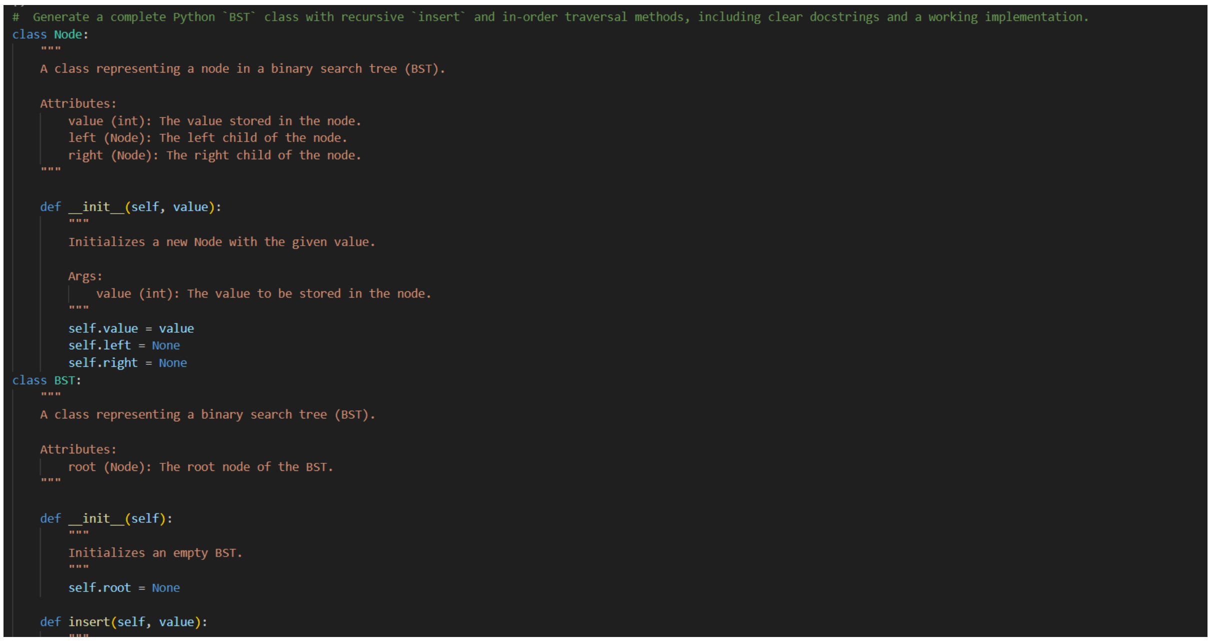
Expected Output:

• BST implementation with recursive insert and traversal methods.

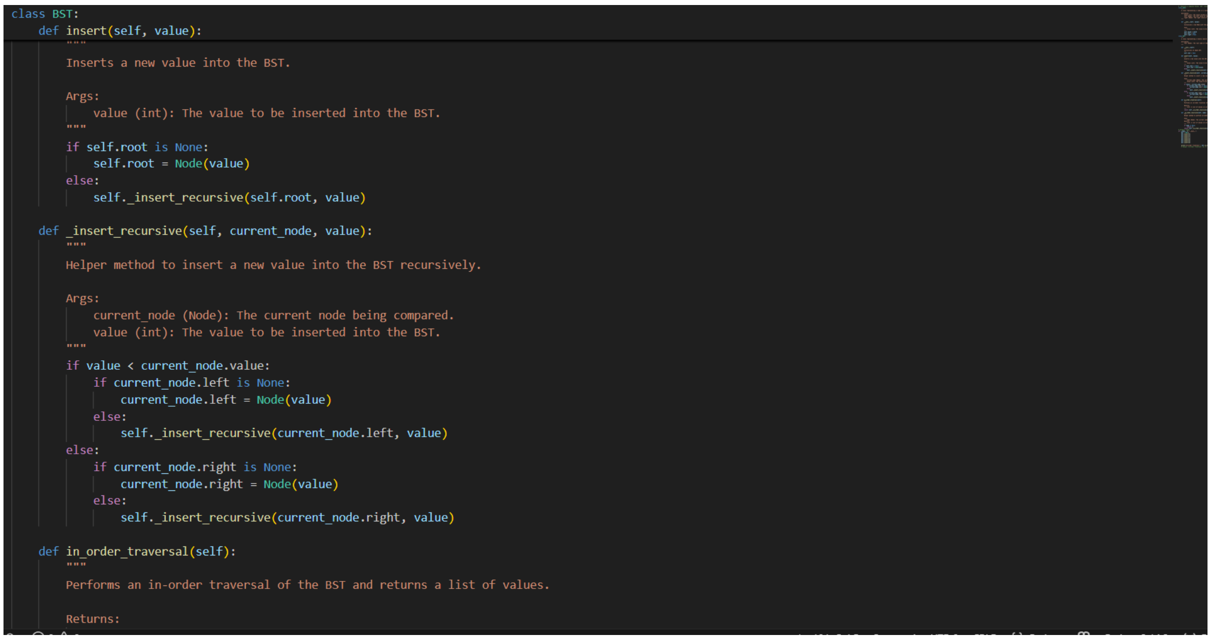
PROMPT:

Generate a complete Python BST class with recursive insert and in-order traversal methods, including clear docstrings and a working implementation.

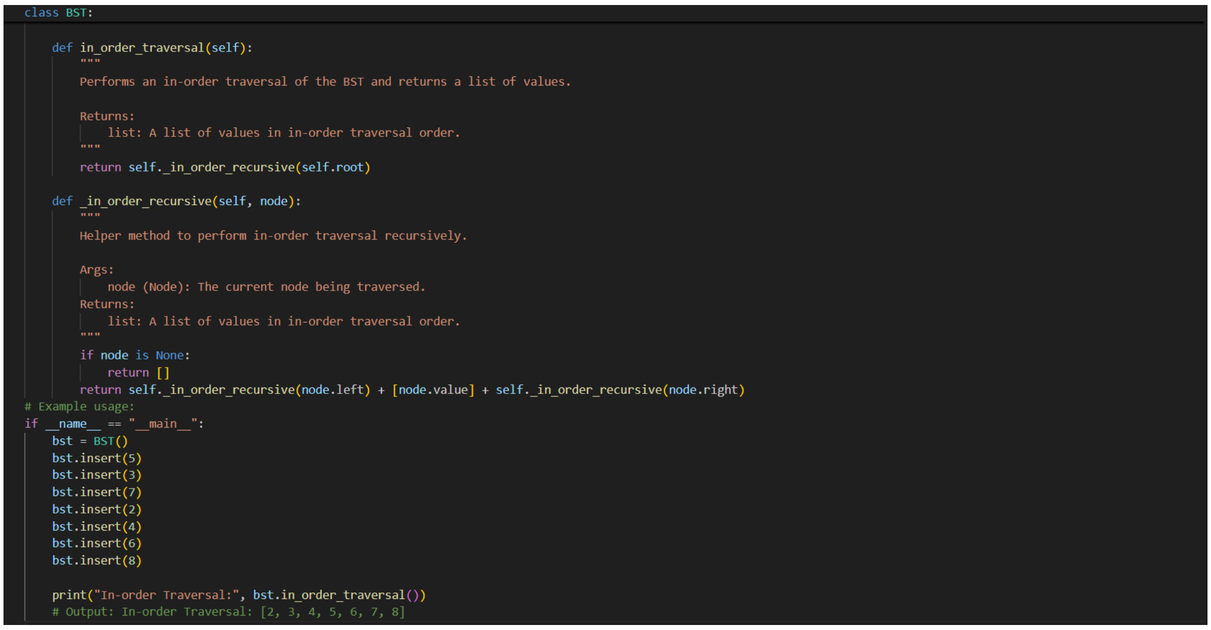
CODE:



```python
#  Generate a complete Python `BST` class with recursive `insert` and in-order traversal methods, including clear docstrings and a working implementation.
class Node:
    """
    A class representing a node in a binary search tree (BST).

    Attributes:
        value (int): The value stored in the node.
        left (Node): The left child of the node.
        right (Node): The right child of the node.
    """

    def __init__(self, value):
        """
        Initializes a new Node with the given value.

        Args:
            value (int): The value to be stored in the node.
        """
        self.value = value
        self.left = None
        self.right = None
class BST:
    """
    A class representing a binary search tree (BST).

    Attributes:
        root (Node): The root node of the BST.
    """

    def __init__(self):
        """
        Initializes an empty BST.
        """
        self.root = None

    def insert(self, value):
```

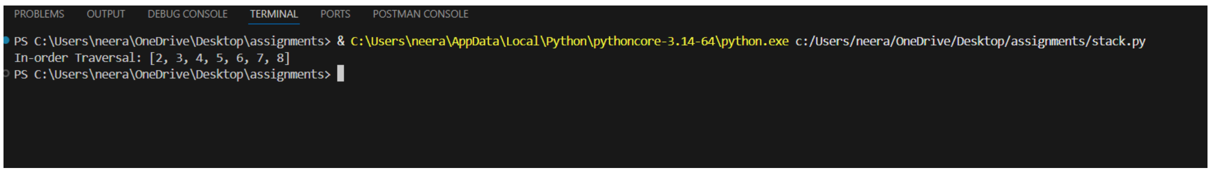


```python
class BST:
    def insert(self, value):
        """
        Inserts a new value into the BST.

        Args:
            value (int): The value to be inserted into the BST.
        """
        if self.root is None:
            self.root = Node(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, current_node, value):
        """
        Helper method to insert a new value into the BST recursively.

        Args:
            current_node (Node): The current node being compared.
            value (int): The value to be inserted into the BST.
        """
        if value < current_node.value:
            if current_node.left is None:
                current_node.left = Node(value)
            else:
                self._insert_recursive(current_node.left, value)
        else:
            if current_node.right is None:
                current_node.right = Node(value)
            else:
                self._insert_recursive(current_node.right, value)

    def in_order_traversal(self):
        """
        Performs an in-order traversal of the BST and returns a list of values.

        Returns:
```



```
class BST:

    def in_order_traversal(self):
        """
        Performs an in-order traversal of the BST and returns a list of values.

        Returns:
            list: A list of values in in-order traversal order.
        """
        return self._in_order_recursive(self.root)

    def _in_order_recursive(self, node):
        """
        Helper method to perform in-order traversal recursively.

        Args:
            node (Node): The current node being traversed.
        Returns:
            list: A list of values in in-order traversal order.
        """
        if node is None:
            return []
        return self._in_order_recursive(node.left) + [node.value] + self._in_order_recursive(node.right)
# Example usage:
if __name__ == "__main__":
    bst = BST()
    bst.insert(5)
    bst.insert(3)
    bst.insert(7)
    bst.insert(2)
    bst.insert(4)
    bst.insert(6)
    bst.insert(8)

    print("In-order Traversal:", bst.in_order_traversal())
    # Output: In-order Traversal: [2, 3, 4, 5, 6, 7, 8]
```

OUTPUT:



```
PS C:\Users\neera\OneDrive\Desktop\assignments> & C:\Users\neera\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/neera/OneDrive/Desktop/assignments/stack.py
In-order Traversal: [2, 3, 4, 5, 6, 7, 8]
PS C:\Users\neera\OneDrive\Desktop\assignments> 
```

#JUSTIFICATION:

The BST` class implements a binary search tree with methods for inserting values and performing in-order traversal. The `insert` method uses recursion to find the correct position for the new value, ensuring that the properties of the BST are maintained.The `in_order_traversal` method also uses recursion to visit nodes in the correct order (left, root, right), which results in a sorted list of values when traversed. This implementation is efficient and follows standard practices for BST operations.

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

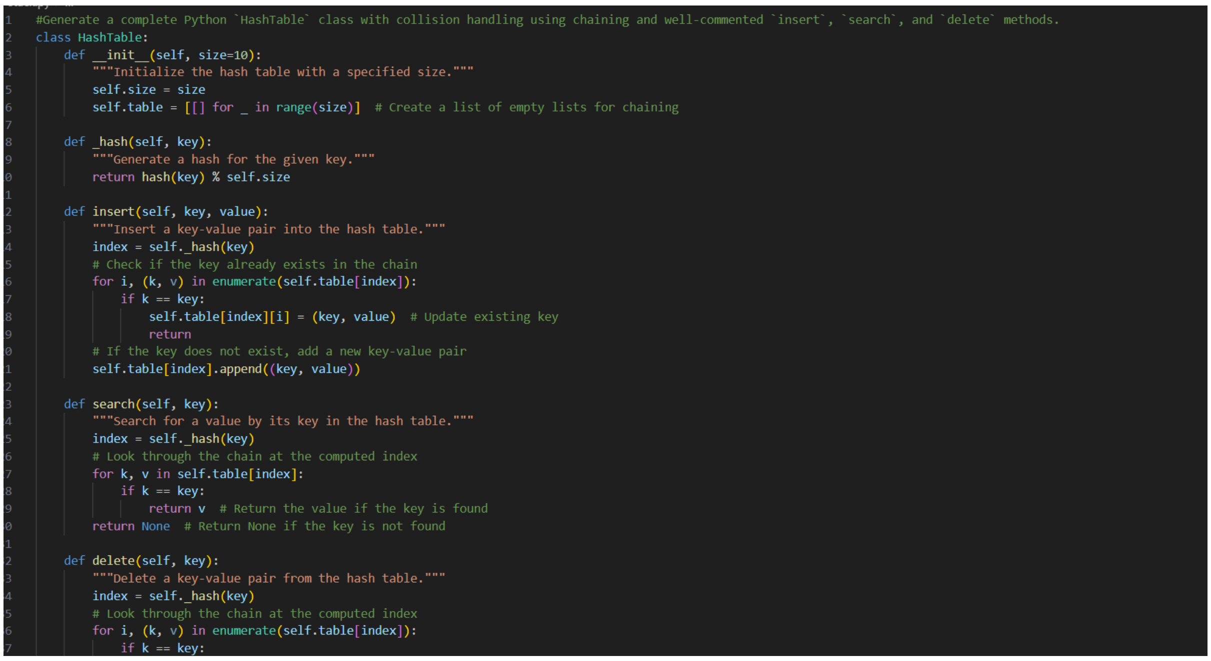
Expected Output:

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

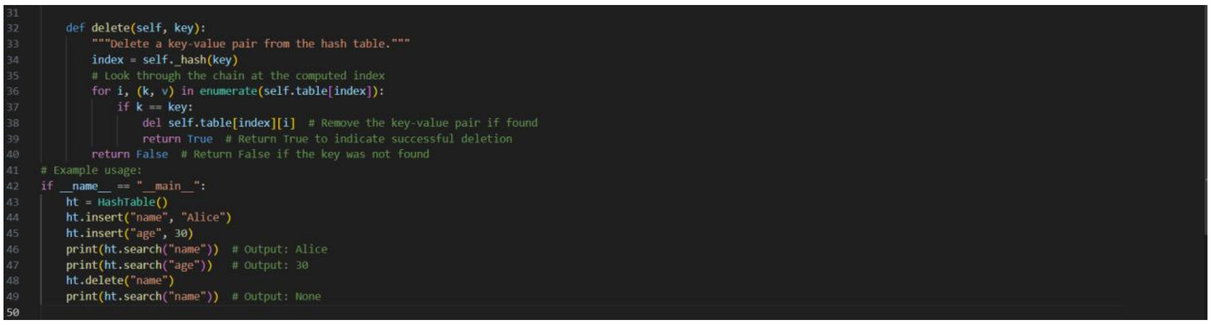
PROMPT:

Generate a complete Python HashTable class with collision handling using chaining and well-commented insert, search, and delete methods.

CODE:

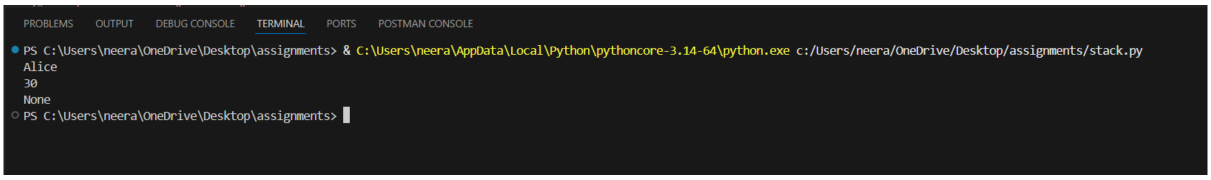


```python
#Generate a complete Python `HashTable` class with collision handling using chaining and well-commented `insert`, `search`, and `delete` methods.
class HashTable:
    def __init__(self, size=10):
        """Initialize the hash table with a specified size."""
        self.size = size
        self.table = [[] for _ in range(size)]  # Create a list of empty lists for chaining

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

    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
        index = self._hash(key)
        # Check if the key already exists in the chain
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        # If the key does not exist, add a new key-value pair
        self.table[index].append((key, value))

    def search(self, key):
        """Search for a value by its key in the hash table."""
        index = self._hash(key)
        # Look through the chain at the computed index
        for k, v in self.table[index]:
            if k == key:
                return v  # Return the value if the key is found
        return None  # Return None if the key is not found

    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        index = self._hash(key)
        # Look through the chain at the computed index
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
```



```python

    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        index = self._hash(key)
        # Look through the chain at the computed index
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]  # Remove the key-value pair if found
                return True  # Return True to indicate successful deletion
        return False  # Return False if the key was not found
# Example usage:
if __name__ == "__main__":
    ht = HashTable()
    ht.insert("name", "Alice")
    ht.insert("age", 30)
    print(ht.search("name"))  # Output: Alice
    print(ht.search("age"))   # Output: 30
    ht.delete("name")
    print(ht.search("name"))  # Output: None

```

OUTPUT:



```
PS C:\Users\neera\OneDrive\Desktop\assignments> & C:\Users\neera\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/neera/OneDrive/Desktop/assignments/stack.py
Alice
30
None
PS C:\Users\neera\OneDrive\Desktop\assignments> 
```

#JUSTIFICATION

This “HashTable” class implements a basic hash table with collision handling using chaining. The `insert` method allows adding key-value pairs, while the search method retrieves values based on keys. The `delete` method removes key-value pairs from the table. The use of chaining (lists) to handle collisions ensures that multiple key-value pairs can be stored at the same index without overwriting each other, making the implementation robust and efficient for typical use cases.

Task Description #6 – Graph Representation

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

Sample Input Code:

class Graph:

pass

Expected Output:

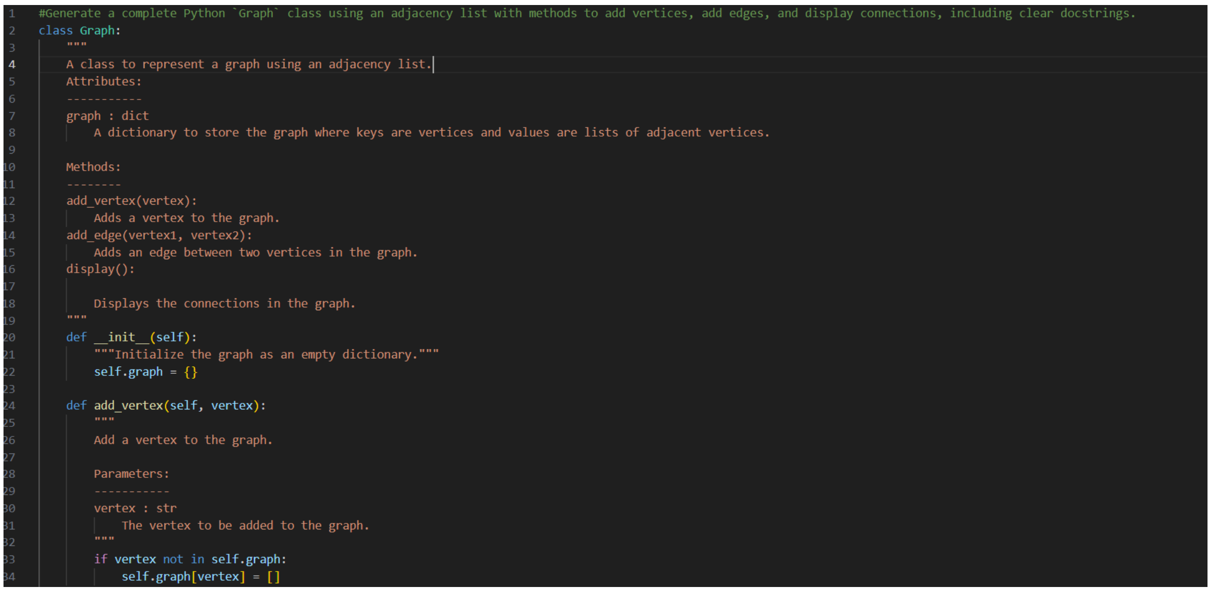
• Graph with methods to add vertices, add edges, and display

connections.

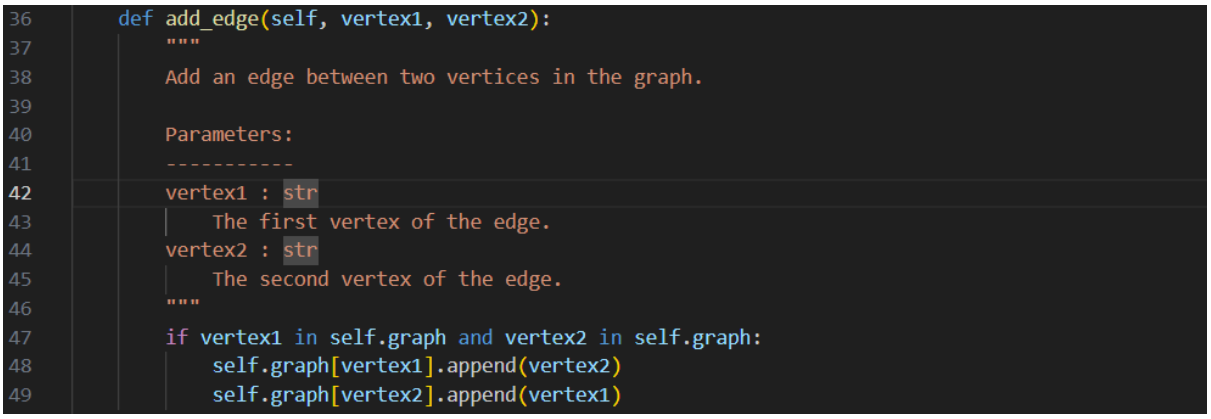
PROMPT:

Generate a complete Python Graph class using an adjacency list with methods to add vertices, add edges, and display connections, including clear docstrings.

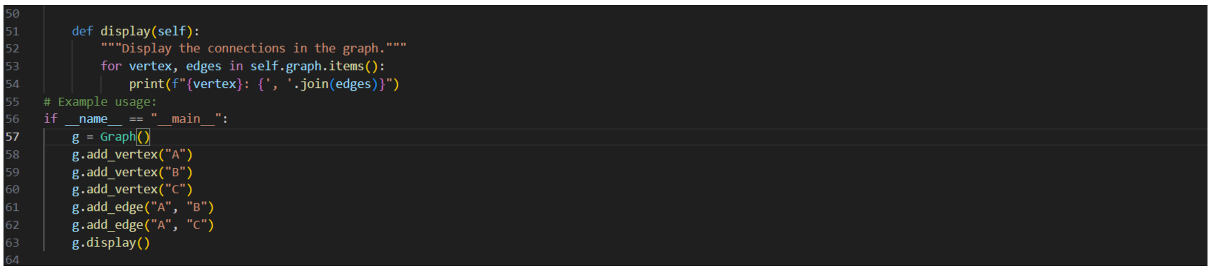
CODE:



```python
#Generate a complete Python `Graph` class using an adjacency list with methods to add vertices, add edges, and display connections, including clear docstrings.
class Graph:
    """
    A class to represent a graph using an adjacency list.
    Attributes:
    -----------
    graph : dict
        A dictionary to store the graph where keys are vertices and values are lists of adjacent vertices.

    Methods:
    --------
    add_vertex(vertex):
        Adds a vertex to the graph.
    add_edge(vertex1, vertex2):
        Adds an edge between two vertices in the graph.
    display():

        Displays the connections in the graph.
    """
    def __init__(self):
        """Initialize the graph as an empty dictionary."""
        self.graph = {}

    def add_vertex(self, vertex):
        """
        Add a vertex to the graph.

        Parameters:
        -----------
        vertex : str
            The vertex to be added to the graph.
        """
        if vertex not in self.graph:
            self.graph[vertex] = []
```

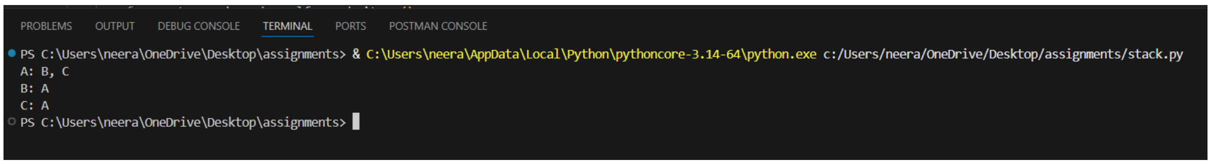


```python
    def add_edge(self, vertex1, vertex2):
        """
        Add an edge between two vertices in the graph.

        Parameters:
        -----------
        vertex1 : str
            The first vertex of the edge.
        vertex2 : str
            The second vertex of the edge.
        """
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertex1)
```



```python

    def display(self):
        """Display the connections in the graph."""
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {', '.join(edges)}")
# Example usage:
if __name__ == "__main__":
    g = Graph()
    g.add_vertex("A")
    g.add_vertex("B")
    g.add_vertex("C")
    g.add_edge("A", "B")
    g.add_edge("A", "C")
    g.display()

```

OUTPUT:



```
PS C:\Users\neera\OneDrive\Desktop\assignments> & C:\Users\neera\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/neera/OneDrive/Desktop/assignments/stack.py
A: B, C
B: A
C: A
PS C:\Users\neera\OneDrive\Desktop\assignments> 
```

#JUSTIFICATION:

This “Graph “ class is implemented using an adjacency list, which is efficient for representing sparse graphs. The methods provided allow for easy addition of vertices and edges, as well as a clear way to display the graph's connections. The use of docstrings enhances code readability and provides clear documentation for each method.

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:

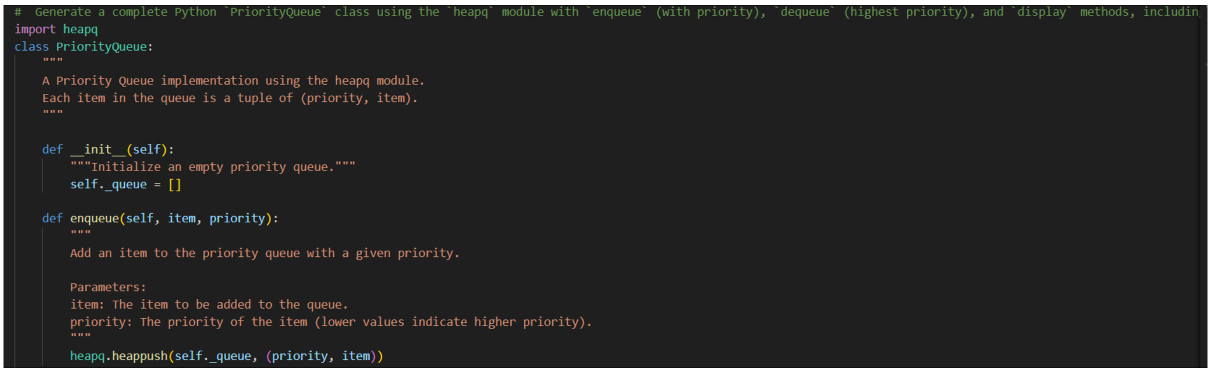
• Implementation with enqueue (priority), dequeue (highest priority),

and display methods.

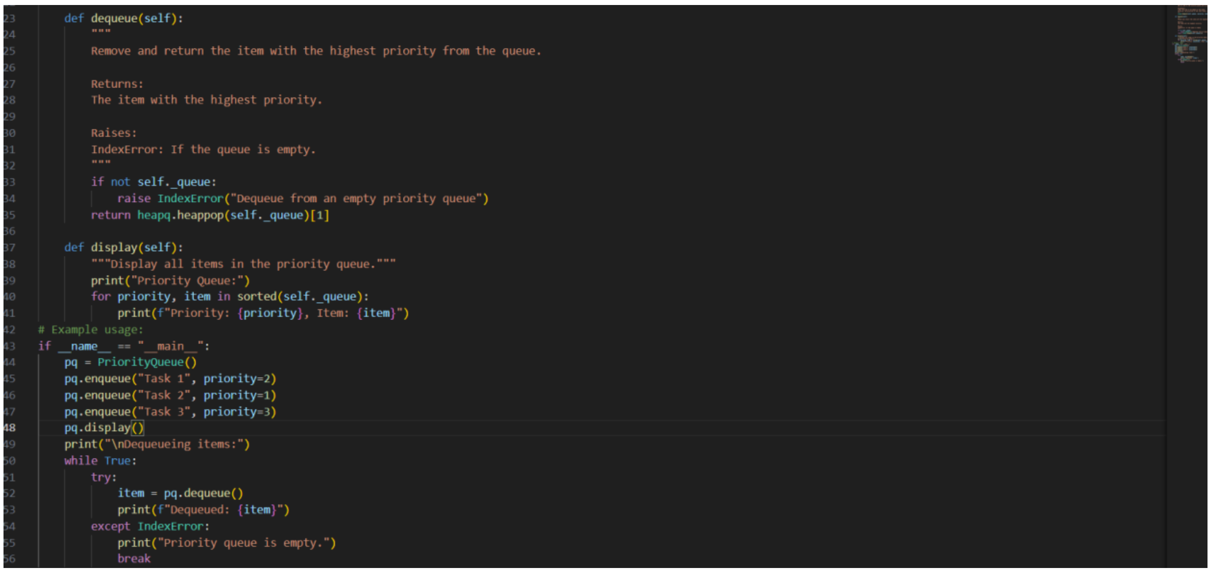
PROMPT:

Generate a complete Python PriorityQueue class using the heapq module with enqueue (with priority), dequeue (highest priority), and display methods, including proper docstrings.

CODE:

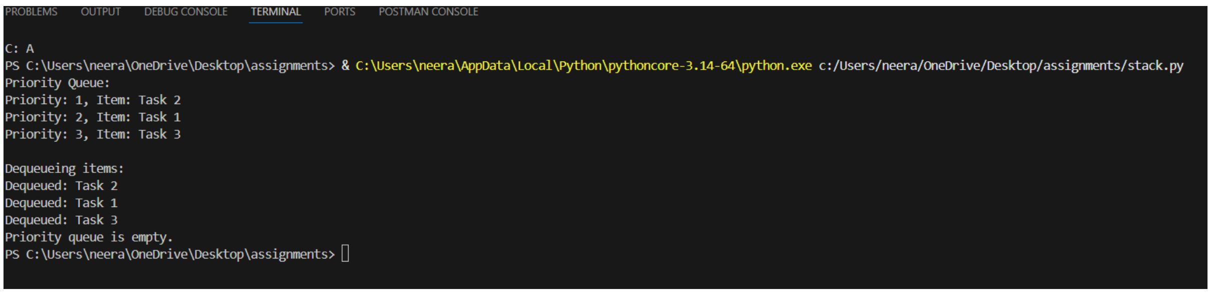


```python
#  Generate a complete Python `PriorityQueue` class using the `heapq` module with `enqueue` (with priority), `dequeue` (highest priority), and `display` methods, includin
import heapq
class PriorityQueue:
    """
    A Priority Queue implementation using the heapq module.
    Each item in the queue is a tuple of (priority, item).
    """

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

    def enqueue(self, item, priority):
        """
        Add an item to the priority queue with a given priority.

        Parameters:
        item: The item to be added to the queue.
        priority: The priority of the item (lower values indicate higher priority).
        """
        heapq.heappush(self._queue, (priority, item))
```



```python
    def dequeue(self):
        """
        Remove and return the item with the highest priority from the queue.

        Returns:
        The item with the highest priority.

        Raises:
        IndexError: If the queue is empty.
        """
        if not self._queue:
            raise IndexError("Dequeue from an empty priority queue")
        return heapq.heappop(self._queue)[1]

    def display(self):
        """Display all items in the priority queue."""
        print("Priority Queue:")
        for priority, item in sorted(self._queue):
            print(f"Priority: {priority}, Item: {item}")
# Example usage:
if __name__ == "__main__":
    pq = PriorityQueue()
    pq.enqueue("Task 1", priority=2)
    pq.enqueue("Task 2", priority=1)
    pq.enqueue("Task 3", priority=3)
    pq.display()
    print("\nDequeueing items:")
    while True:
        try:
            item = pq.dequeue()
            print(f"Dequeued: {item}")
        except IndexError:
            print("Priority queue is empty.")
            break
```

OUTPUT:



```
C: A
PS C:\Users\neera\OneDrive\Desktop\assignments> & C:\Users\neera\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/neera/OneDrive/Desktop/assignments/stack.py
Priority Queue:
Priority: 1, Item: Task 2
Priority: 2, Item: Task 1
Priority: 3, Item: Task 3

Dequeueing items:
Dequeued: Task 2
Dequeued: Task 1
Dequeued: Task 3
Priority queue is empty.
PS C:\Users\neera\OneDrive\Desktop\assignments> 
```

#JUSTIFICATION:

The PriorityQueue class uses Python's heapq module to store items so that the highest priority item can be removed quickly. The enqueue method adds an item with its priority, and the dequeue method removes the item with the highest priority, which means the one with the smallest priority value. The display method shows all the items currently in the queue along with their priorities. This makes the queue efficient and useful for handling tasks based on importance.

Task Description #8 – Deque

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

Sample Input Code:

class DequeDS:

pass

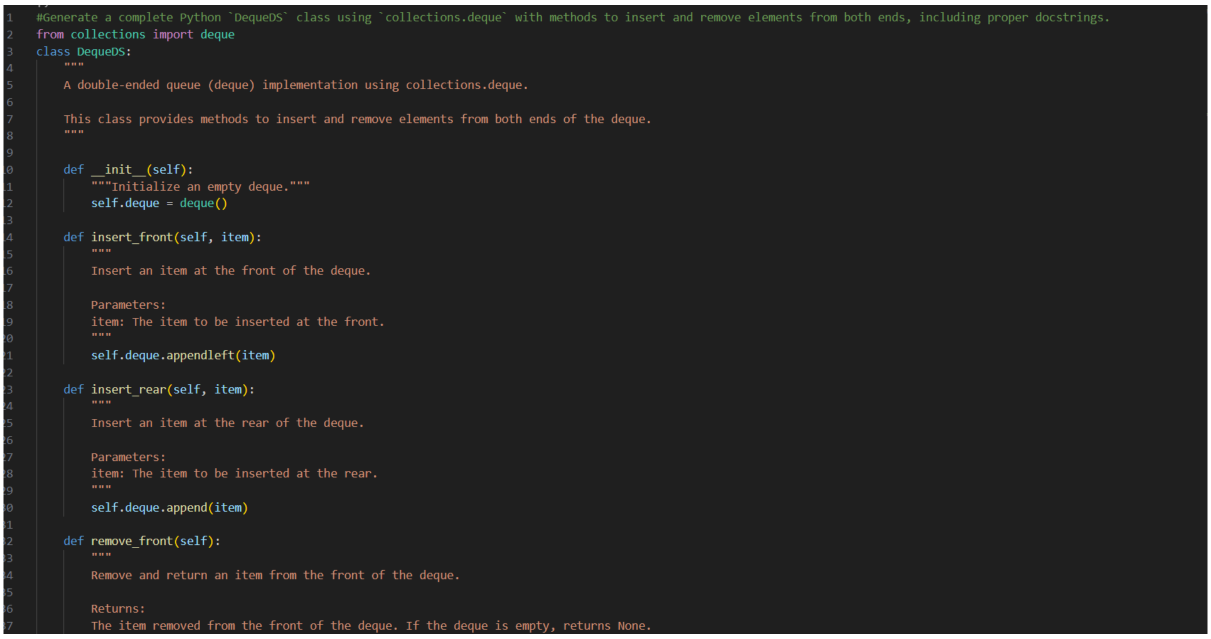
Expected Output:

• Insert and remove from both ends with docstrings.

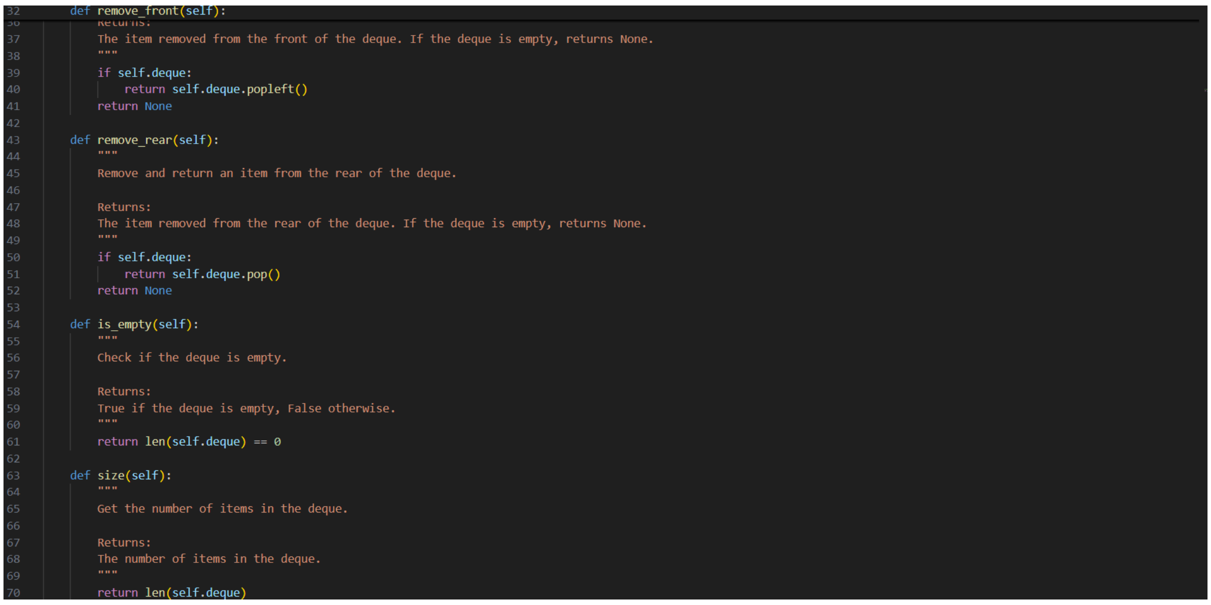
PROMPT:

Generate a complete Python DequeDS class using collections.deque with methods to insert and remove elements from both ends, including proper docstrings.

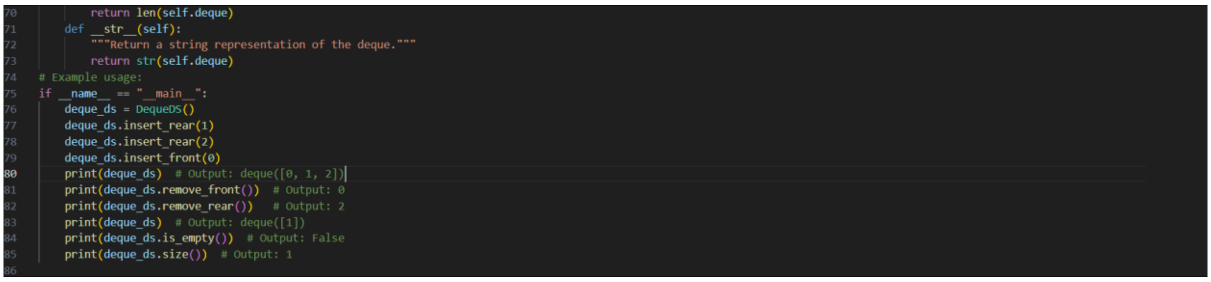
CODE:



```python
#Generate a complete Python `DequeDS` class using `collections.deque` with methods to insert and remove elements from both ends, including proper docstrings.
from collections import deque
class DequeDS:
    """
    A double-ended queue (deque) implementation using collections.deque.

    This class provides methods to insert and remove elements from both ends of the deque.
    """

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

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

        Parameters:
        item: The item to be inserted at the front.
        """
        self.deque.appendleft(item)

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

        Parameters:
        item: The item to be inserted at the rear.
        """
        self.deque.append(item)

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

        Returns:
        The item removed from the front of the deque. If the deque is empty, returns None.
```

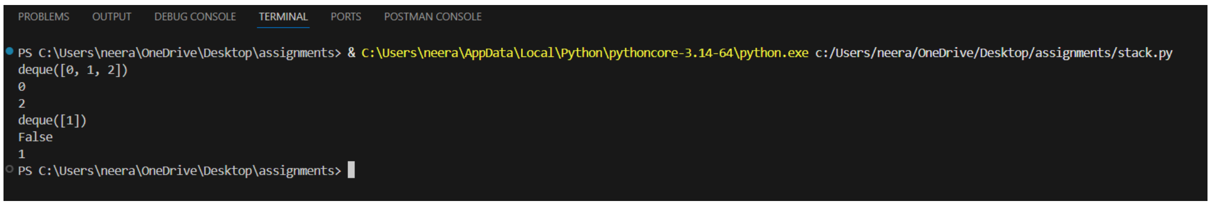


```python
    def remove_front(self):
        The item removed from the front of the deque. If the deque is empty, returns None.
        """
        if self.deque:
            return self.deque.popleft()
        return None

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

        Returns:
        The item removed from the rear of the deque. If the deque is empty, returns None.
        """
        if self.deque:
            return self.deque.pop()
        return None

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

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

    def size(self):
        """
        Get the number of items in the deque.

        Returns:
        The number of items in the deque.
        """
        return len(self.deque)
```



```python
            return len(self.deque)
        def __str__(self):
            """Return a string representation of the deque."""
            return str(self.deque)
    # Example usage:
    if __name__ == "__main__":
        deque_ds = DequeDS()
        deque_ds.insert_rear(1)
        deque_ds.insert_rear(2)
        deque_ds.insert_front(0)
        print(deque_ds)  # Output: deque([0, 1, 2])
        print(deque_ds.remove_front())  # Output: 0
        print(deque_ds.remove_rear())   # Output: 2
        print(deque_ds)  # Output: deque([1])
        print(deque_ds.is_empty())  # Output: False
        print(deque_ds.size())  # Output: 1
```

OUTPUT:



```
PS C:\Users\neera\OneDrive\Desktop\assignments> & C:\Users\neera\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/neera/OneDrive/Desktop/assignments/stack.py
deque([0, 1, 2])
0
2
deque([1])
False
1
PS C:\Users\neera\OneDrive\Desktop\assignments>
```

#JUSTIFICATION

The `DequeDS` class is implemented using Python's built-in `collections.deque`, which provides an efficient way to handle double-ended queues. The methods `insert_front`, `insert_rear`, `remove_front`, and `remove_rear` allow for easy insertion and removal of elements from both ends of the deque. The `is_empty` and `size` methods provide additional functionality to check if the deque is empty.

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:

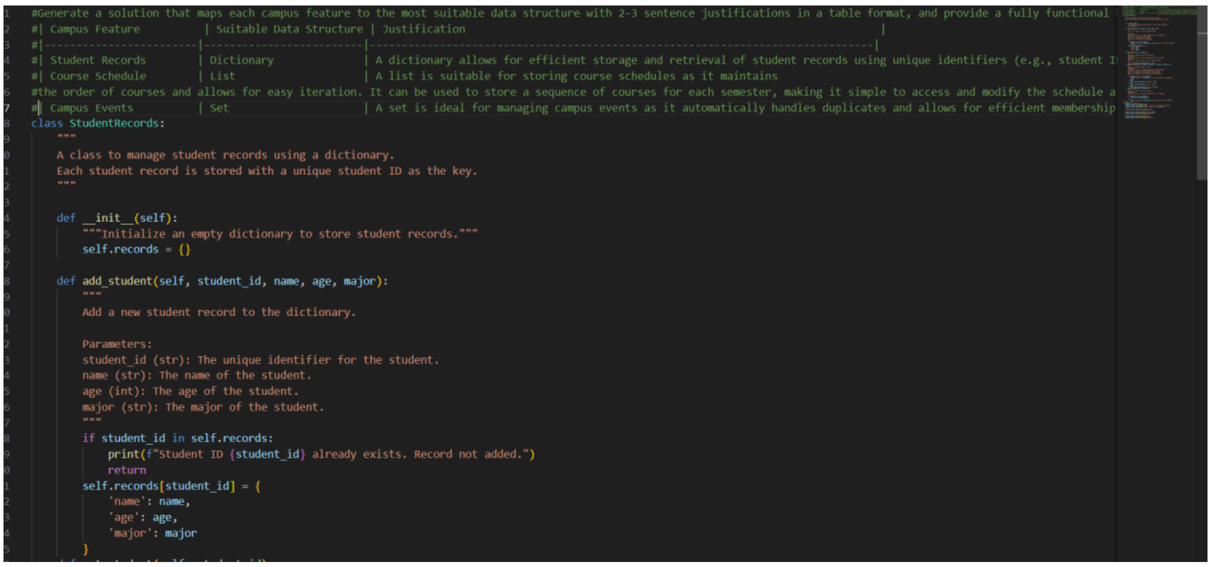
• A table mapping feature → chosen data structure → justification.

• A functional Python program implementing the chosen feature with comments and docstrings.

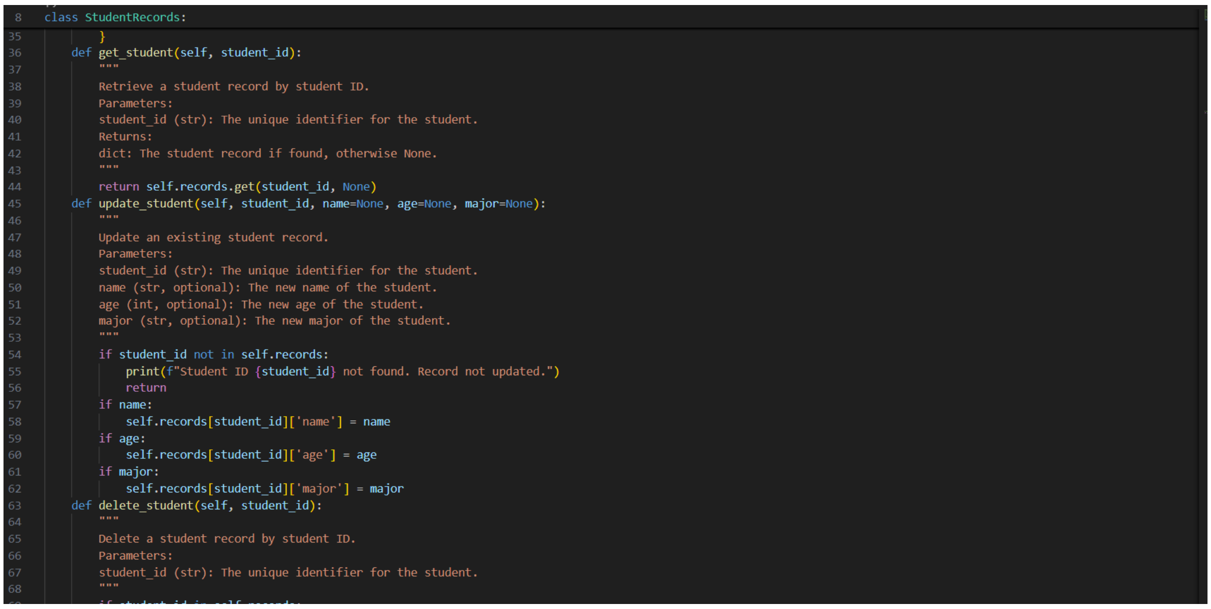
PROMPT:

Generate a solution that maps each campus feature to the most suitable data structure with 2–3 sentence justifications in a table format, and provide a fully functional Python program (with comments and docstrings) implementing one selected feature.

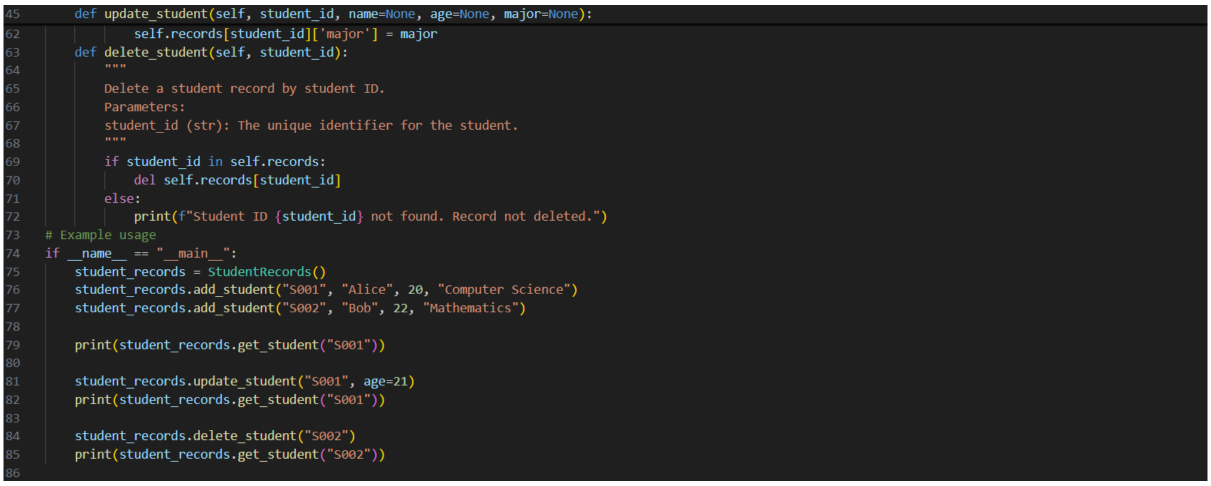
CODE:



```
#Generate a solution that maps each campus feature to the most suitable data structure with 2-3 sentence justifications in a table format, and provide a fully functional
#| Campus Feature          | Suitable Data Structure | Justification                                                          |
#|------------------------|------------------------|------------------------------------------------------------------------|
#| Student Records        | Dictionary             | A dictionary allows for efficient storage and retrieval of student records using unique identifiers (e.g., student I
#| Course Schedule        | List                   | A list is suitable for storing course schedules as it maintains
#the order of courses and allows for easy iteration. It can be used to store a sequence of courses for each semester, making it simple to access and modify the schedule a
#| Campus Events          | Set                    | A set is ideal for managing campus events as it automatically handles duplicates and allows for efficient membership
class StudentRecords:
    """
    A class to manage student records using a dictionary.
    Each student record is stored with a unique student ID as the key.
    """


    def __init__(self):
        """Initialize an empty dictionary to store student records."""
        self.records = {}

    def add_student(self, student_id, name, age, major):
        """
        Add a new student record to the dictionary.

        Parameters:
        student_id (str): The unique identifier for the student.
        name (str): The name of the student.
        age (int): The age of the student.
        major (str): The major of the student.
        """
        if student_id in self.records:
            print(f"Student ID {student_id} already exists. Record not added.")
            return
        self.records[student_id] = {
            'name': name,
            'age': age,
            'major': major
        }
```

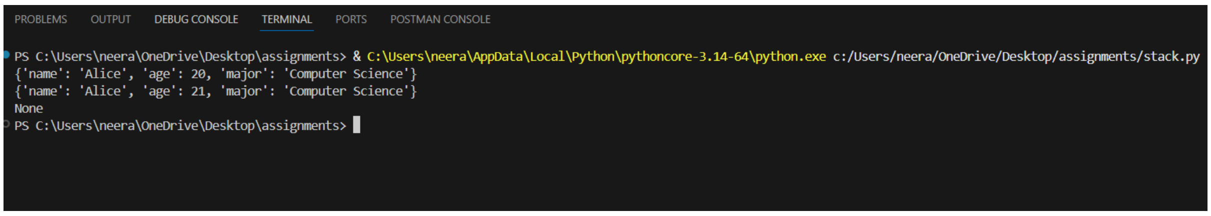


```python
class StudentRecords:
        }
    def get_student(self, student_id):
        """
        Retrieve a student record by student ID.
        Parameters:
        student_id (str): The unique identifier for the student.
        Returns:
        dict: The student record if found, otherwise None.
        """
        return self.records.get(student_id, None)
    def update_student(self, student_id, name=None, age=None, major=None):
        """
        Update an existing student record.
        Parameters:
        student_id (str): The unique identifier for the student.
        name (str, optional): The new name of the student.
        age (int, optional): The new age of the student.
        major (str, optional): The new major of the student.
        """
        if student_id not in self.records:
            print(f"Student ID {student_id} not found. Record not updated.")
            return
        if name:
            self.records[student_id]['name'] = name
        if age:
            self.records[student_id]['age'] = age
        if major:
            self.records[student_id]['major'] = major
    def delete_student(self, student_id):
        """
        Delete a student record by student ID.
        Parameters:
        student_id (str): The unique identifier for the student.
        """
```



```python
    def update_student(self, student_id, name=None, age=None, major=None):
            self.records[student_id]['major'] = major
    def delete_student(self, student_id):
        """
        Delete a student record by student ID.
        Parameters:
        student_id (str): The unique identifier for the student.
        """
        if student_id in self.records:
            del self.records[student_id]
        else:
            print(f"Student ID {student_id} not found. Record not deleted.")
# Example usage
if __name__ == "__main__":
    student_records = StudentRecords()
    student_records.add_student("S001", "Alice", 20, "Computer Science")
    student_records.add_student("S002", "Bob", 22, "Mathematics")

    print(student_records.get_student("S001"))

    student_records.update_student("S001", age=21)
    print(student_records.get_student("S001"))

    student_records.delete_student("S002")
    print(student_records.get_student("S002"))
```

OUTPUT:



```
PS C:\Users\neera\OneDrive\Desktop\assignments> & C:\Users\neera\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/neera/OneDrive/Desktop/assignments/stack.py
{'name': 'Alice', 'age': 20, 'major': 'Computer Science'}
{'name': 'Alice', 'age': 21, 'major': 'Computer Science'}
None
PS C:\Users\neera\OneDrive\Desktop\assignments> 
```

#JUSTIFICATION:

The StudentRecords class uses a dictionary to manage student records efficiently. Each student is identified by a unique student ID, which serves as the key in the dictionary, allowing for quick access and updates to student information. This structure is ideal for handling a large number of records while ensuring that operations such as adding, retrieving, updating, and deleting records are performed in constant time on average.

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:

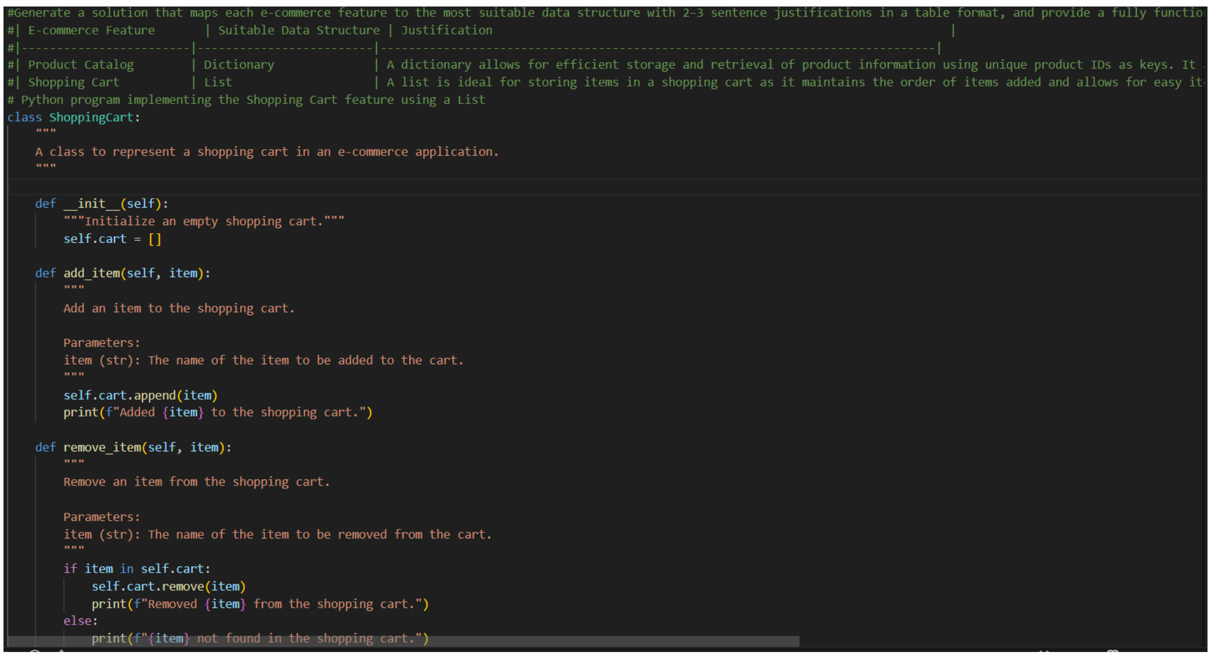
• A table mapping feature → chosen data structure → justification.

• A functional Python program implementing the chosen feature with comments and docstrings.

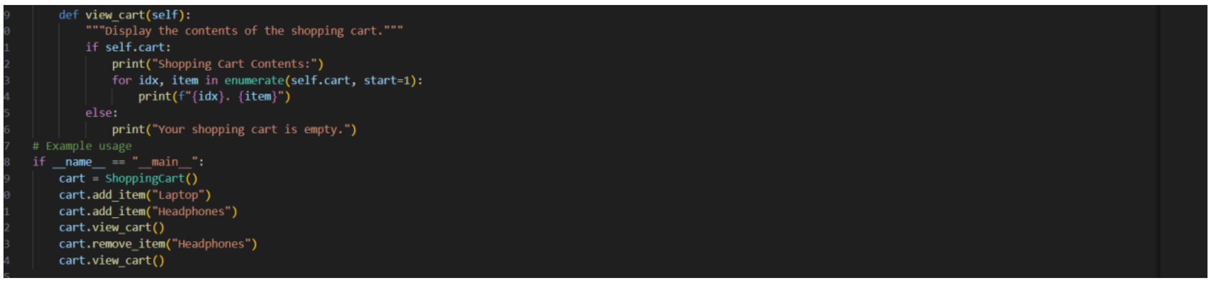
PROMPT:

Generate a solution that maps each e-commerce feature to the most suitable data structure with 2–3 sentence justifications in a table format, and provide a fully functional Python program (with comments and docstrings) implementing one selected feature.

CODE:

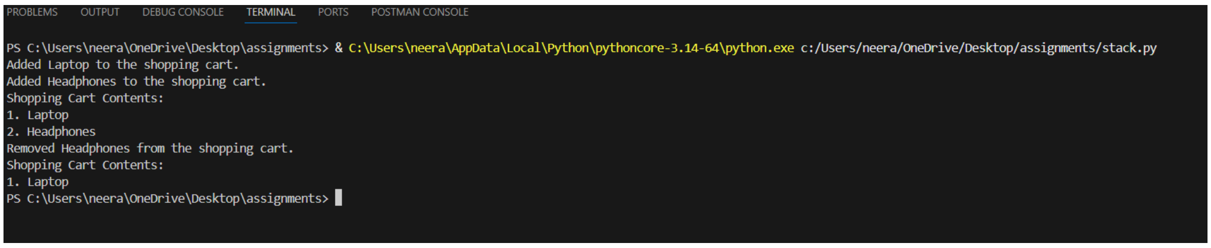


```python
#Generate a solution that maps each e-commerce feature to the most suitable data structure with 2-3 sentence justifications in a table format, and provide a fully functio
#| E-commerce Feature       | Suitable Data Structure | Justification                                                                       |
#|------------------------|------------------------|-----------------------------------------------------------------------------------|
#| Product Catalog        | Dictionary              | A dictionary allows for efficient storage and retrieval of product information using unique product IDs as keys. It
#| Shopping Cart          | List                    | A list is ideal for storing items in a shopping cart as it maintains the order of items added and allows for easy it
# Python program implementing the Shopping Cart feature using a List
class ShoppingCart:
    """
    A class to represent a shopping cart in an e-commerce application.
    """


    def __init__(self):
        """Initialize an empty shopping cart."""
        self.cart = []

    def add_item(self, item):
        """
        Add an item to the shopping cart.

        Parameters:
        item (str): The name of the item to be added to the cart.
        """
        self.cart.append(item)
        print(f"Added {item} to the shopping cart.")

    def remove_item(self, item):
        """
        Remove an item from the shopping cart.

        Parameters:
        item (str): The name of the item to be removed from the cart.
        """
        if item in self.cart:
            self.cart.remove(item)
            print(f"Removed {item} from the shopping cart.")
        else:
            print(f"{item} not found in the shopping cart.")
```



```python
    def view_cart(self):
        """Display the contents of the shopping cart."""
        if self.cart:
            print("Shopping Cart Contents:")
            for idx, item in enumerate(self.cart, start=1):
                print(f"{idx}. {item}")
        else:
            print("Your shopping cart is empty.")
# Example usage
if __name__ == "__main__":
    cart = ShoppingCart()
    cart.add_item("Laptop")
    cart.add_item("Headphones")
    cart.view_cart()
    cart.remove_item("Headphones")
    cart.view_cart()
```

OUTPUT:



```
PS C:\Users\neera\OneDrive\Desktop\assignments> & C:\Users\neera\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/neera/OneDrive/Desktop/assignments/stack.py
Added Laptop to the shopping cart.
Added Headphones to the shopping cart.
Shopping Cart Contents:
1. Laptop
2. Headphones
Removed Headphones from the shopping cart.
Shopping Cart Contents:
1. Laptop
PS C:\Users\neera\OneDrive\Desktop\assignments>
```

#JUSTIFICATION:

The ShoppingCart class uses a list to store items because it allows for easy addition and removal of items while maintaining the order in which they were added. This is important for a shopping cart as customers often expect to see their items in the order they were selected. Additionally, lists provide built-in methods for appending and removing items, making the implementation straightforward and efficient for this use case