Assignment-11.1

Name: Arjun Manoj

H.No:2303A52134

Batch:44

Task Description #1– Stack Implementation

Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods

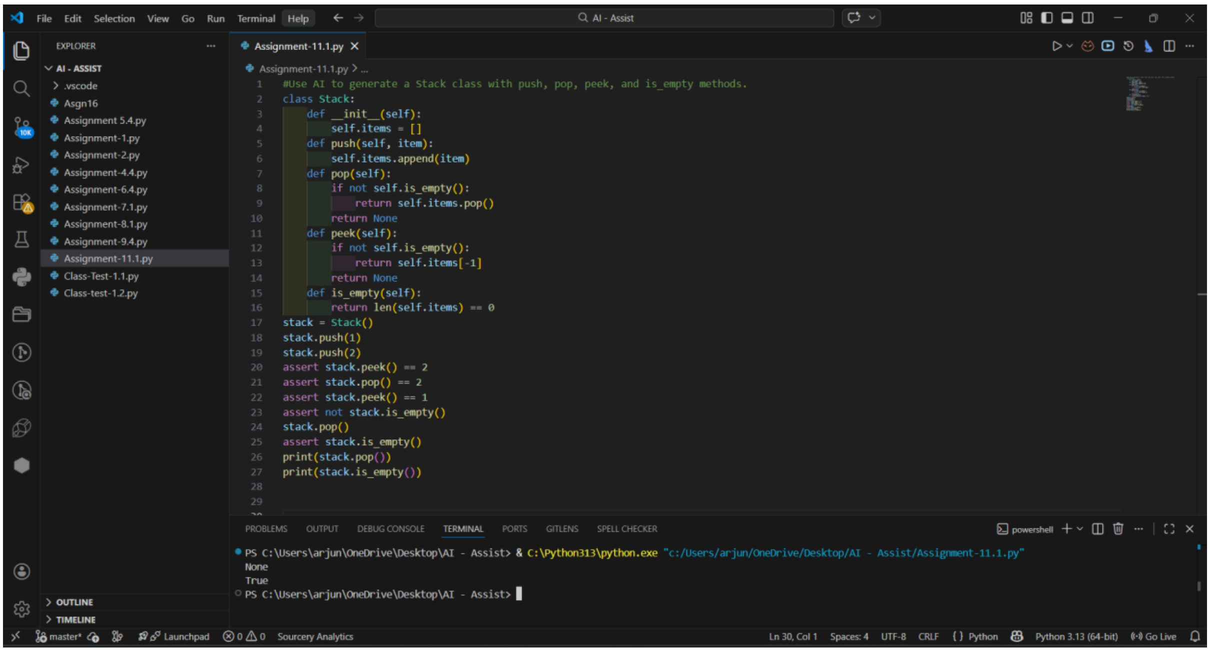
Prompt:

Use AI to generate a Python class that implements a Stack data structure with the following methods:

- push(item) – Add an element to the stack
- pop() – Remove and return the top element
- peek() – Return the top element without removing it
- is_empty() – Check whether the stack is empty

The stack should follow the LIFO (Last In First Out) principle and handle empty stack conditions safely.

Code:



Observation:

The implemented Stack class correctly follows the LIFO principle, where the last element pushed is the first to be removed. The push() method adds elements, while pop() and peek() safely handle empty stack conditions by returning None when the stack is empty. The is_empty() method efficiently checks stack status using length comparison. Overall, the implementation demonstrates proper stack behavior using Python lists

Task Description #2 – Queue Implementation

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

Prompt

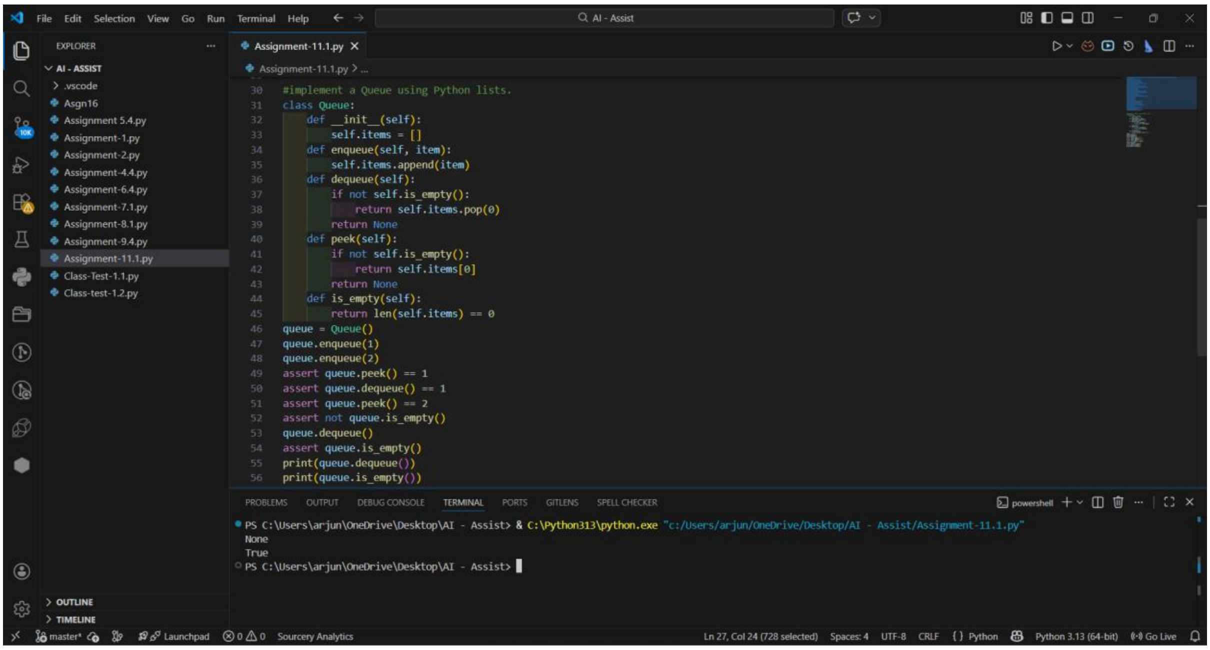
Use AI to generate a Python class that implements a Queue data structure with the following methods:

- enqueue(item) – Add an element to the rear of the queue
- dequeue() – Remove and return the front element

- peek() – Return the front element without removing it
- is_empty() – Check whether the queue is empty

The queue should follow the FIFO (First In First Out) principle and handle empty queue conditions safely.

Code:



Observation:

The implemented Queue class correctly follows the FIFO principle, where the first element inserted is the first one removed. The enqueue() method adds elements at the end, and dequeue() removes elements from the front using pop(0). The peek() method returns the front element without removal, and is_empty() checks whether the queue contains elements. The implementation works correctly, though using pop(0) has O(n) time complexity because elements are shifted after removal.

Task Description #3 – Linked List

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

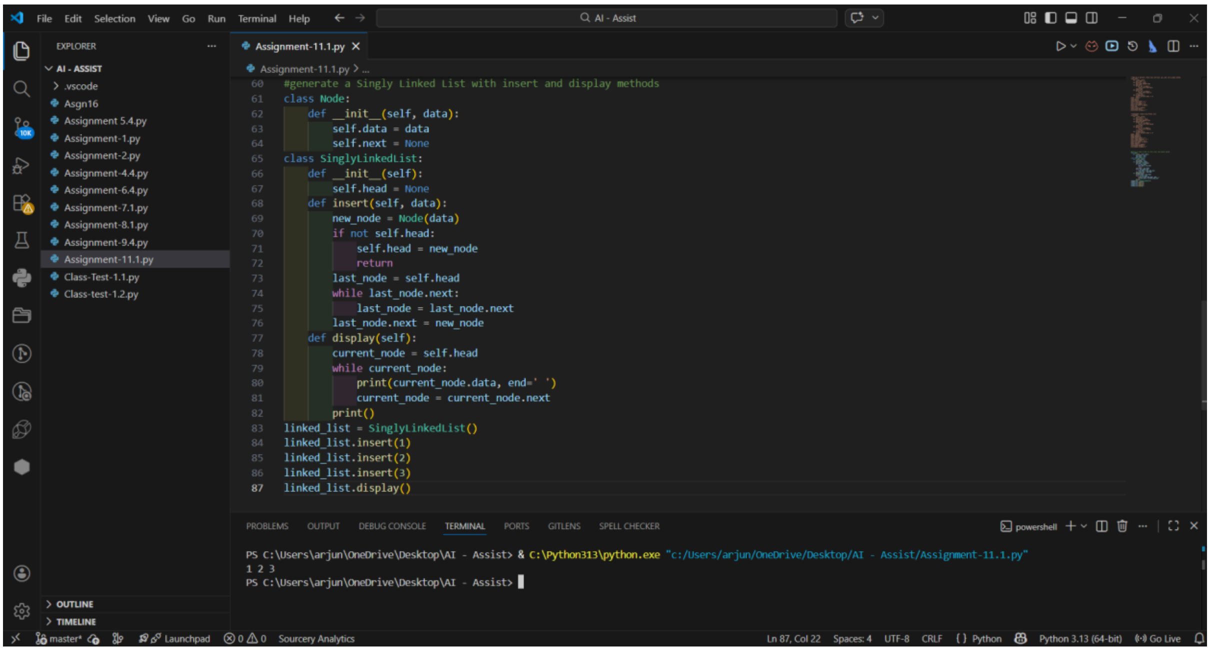
Prompt

Use AI to generate a Python program that implements a Singly Linked List with the following:

- A Node class containing data and next pointer
- An insert(data) method to add elements at the end of the list
- A display() method to print all elements in the list

The linked list should dynamically store elements and maintain proper node connections.

Code:



Observation:

The implemented Singly Linked List correctly maintains dynamic connections between nodes using pointers. The insert() method adds new elements at the end of the list by traversing to the last node and updating its next reference. The display() method traverses the list from head to tail and prints each element. This implementation demonstrates efficient dynamic memory usage without requiring contiguous storage like arrays.

Task Description #4 – Binary Search Tree (BST)

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

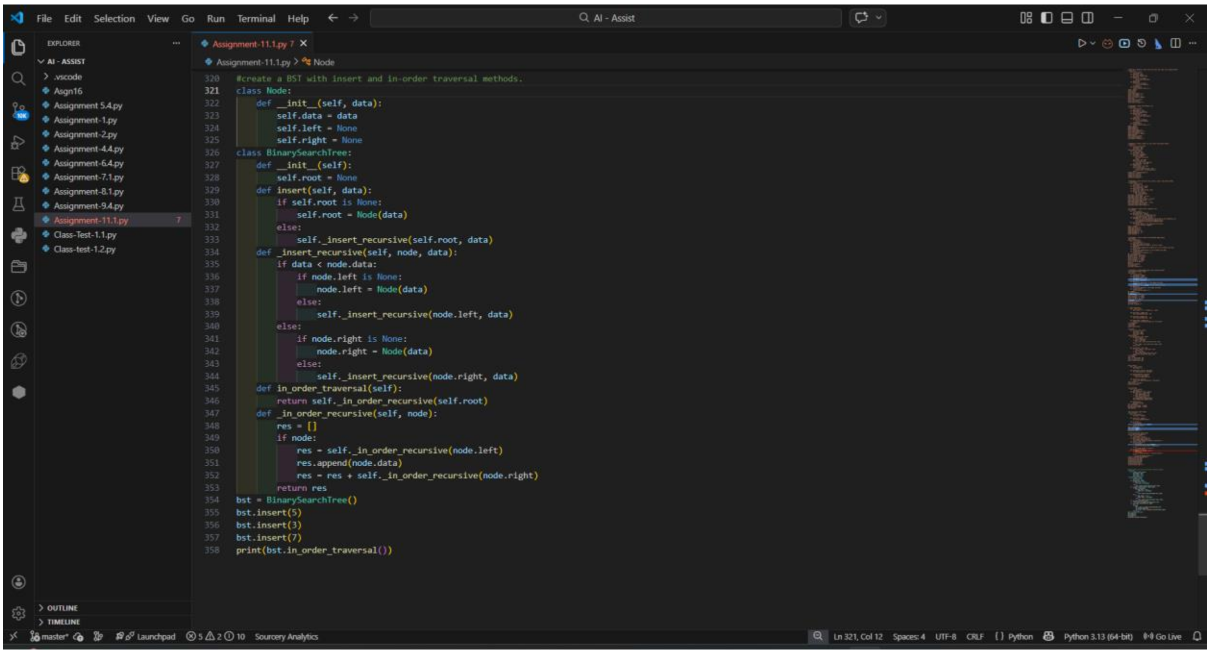
Prompt:

Use AI to generate a Python program that implements a Binary Search Tree (BST) with the following features:

- A Node class containing data, left, and right pointers
- An insert(data) method to add elements while maintaining BST properties
- An in_order_traversal() method to display elements in sorted order

The BST should follow the rule:
Left subtree < Root < Right subtree.

Code:



Observation:

The implemented Binary Search Tree correctly maintains the BST property during insertion by placing smaller values in the left subtree and larger values in the right subtree. The recursive insertion method ensures proper node placement. The in_order_traversal() method visits nodes in Left → Root → Right order, producing a sorted list of elements. This demonstrates how BST enables efficient searching and sorted data retrieval.

Task Description #5 – Hash Table

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

Prompt:

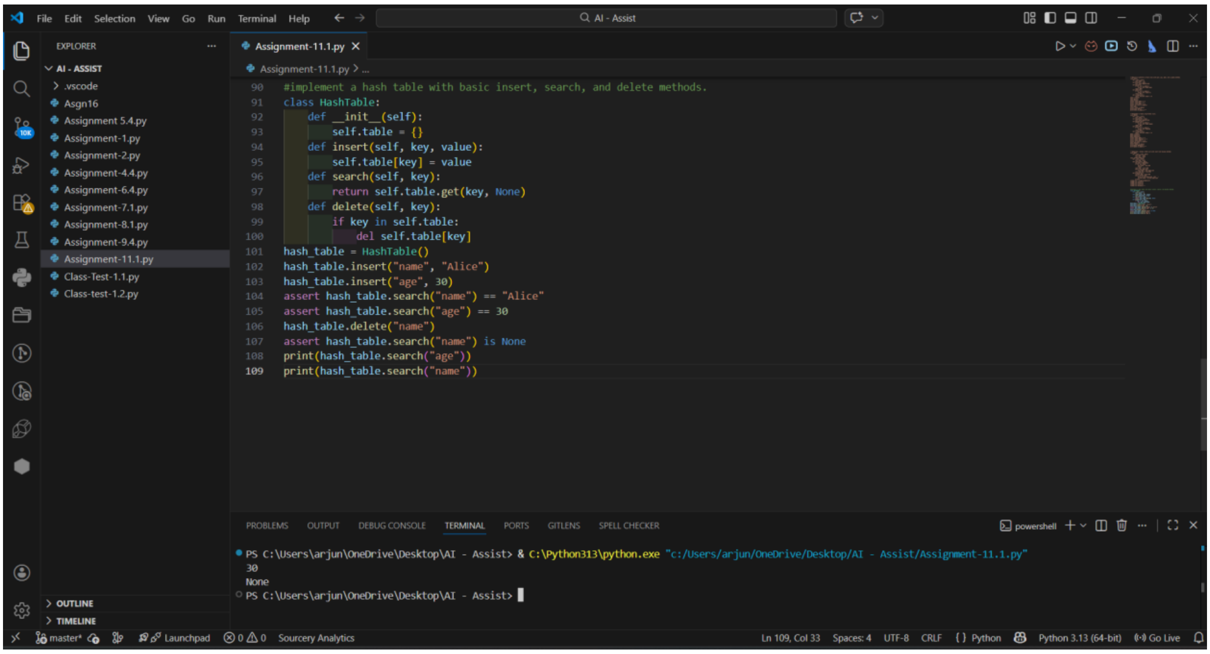
Use AI to generate a Python program that implements a Hash Table with the following methods:

- insert(key, value) – Store a key-value pair

- search(key) – Retrieve value using key
- delete(key) – Remove a key-value pair

The hash table should allow fast lookup and handle cases where the key does not exist.

Code:



Observation:

The implemented Hash Table uses Python's built-in dictionary to store key-value pairs efficiently. The insert() method adds or updates entries, search() retrieves values in average O(1) time, and delete() removes entries if they exist. The implementation demonstrates how hashing enables fast data access and efficient key-based storage.

Task Description #6 – Graph Representation

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

Prompt:

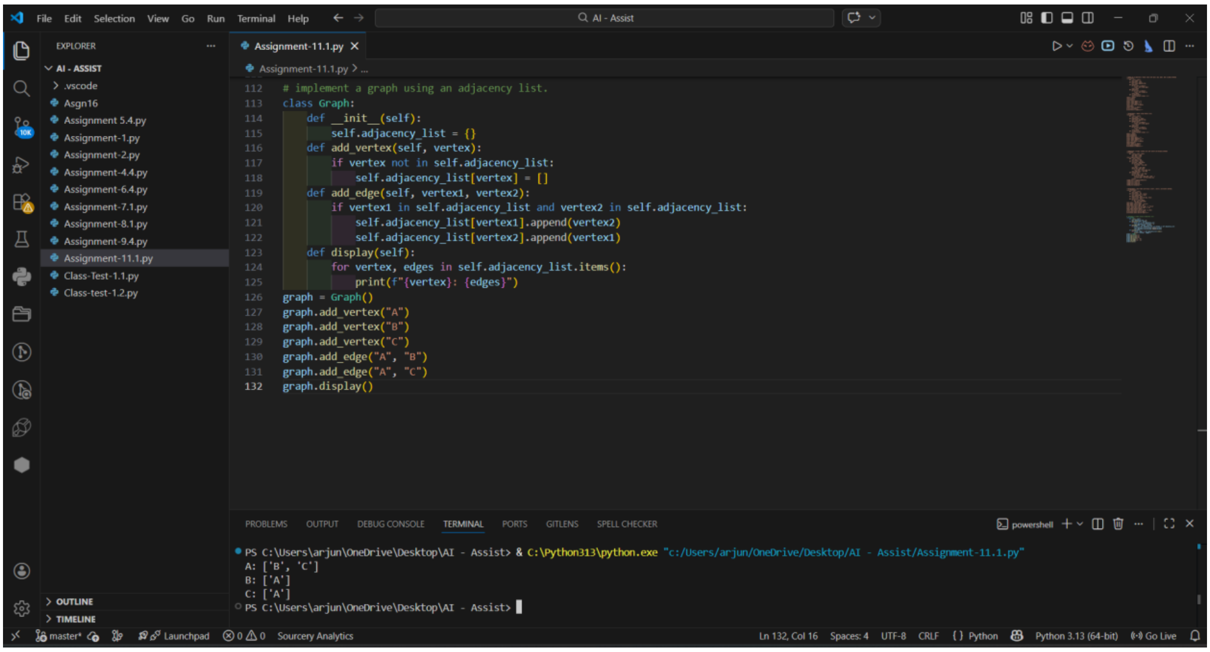
Use AI to generate a Python program that implements a **Graph** using an **Adjacency List** representation.

The graph should include:

- add_vertex(vertex) – Add a new vertex
- add_edge(vertex1, vertex2) – Add an undirected edge between two vertices
- display() – Print all vertices and their connected edges

The graph should dynamically store connections between vertices.

Code:



Observation:

The implemented Graph uses a dictionary to represent the adjacency list, where each vertex maps to a list of connected vertices. The add_vertex() method ensures vertices are created before edges are added, and add_edge() connects two vertices in both directions, forming an undirected graph. This structure efficiently represents relationships between nodes and is space-efficient for sparse graphs.

Task Description #7 – Priority Queue

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

Prompt:

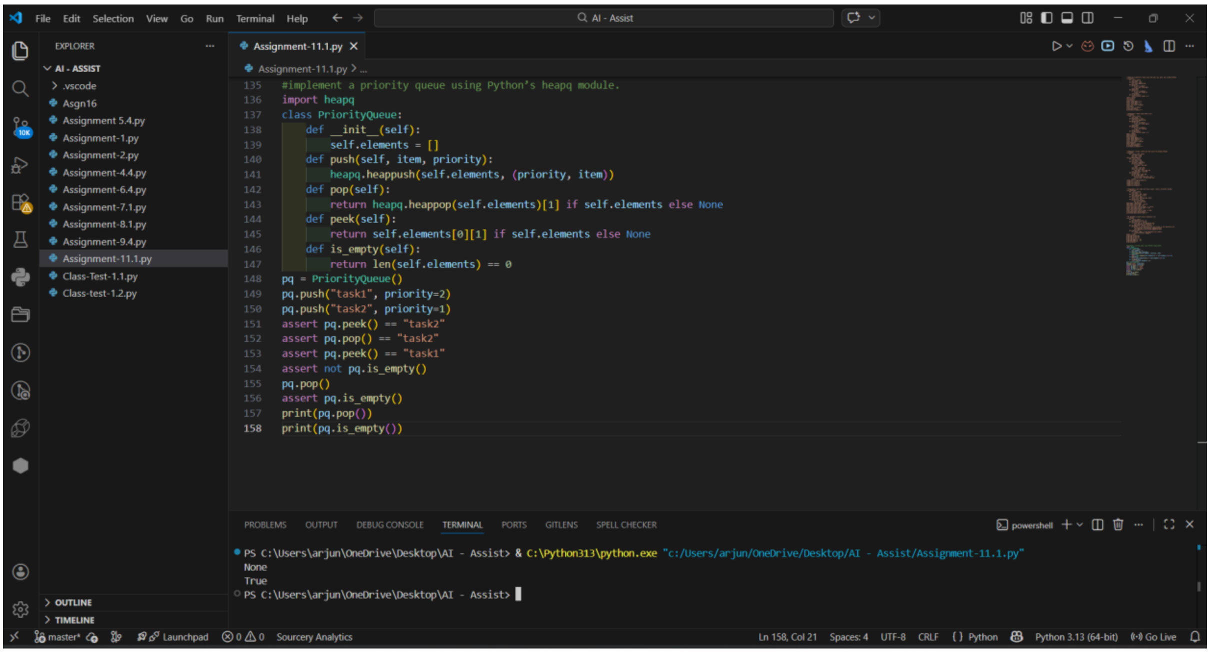
Use AI to generate a Python program that implements a Priority Queue using Python's built-in heapq module.

The class should include:

- push(item, priority) – Insert an element with a given priority
- pop() – Remove and return the element with the highest priority (lowest priority number in min-heap)
- peek() – View the highest priority element without removing it
- is_empty() – Check whether the priority queue is empty

The implementation should follow the heap property for efficient priority-based retrieval.

Code:



Observation:

The implemented Priority Queue uses Python's heapq, which internally maintains a min-heap structure. Elements are stored as (priority, item) tuples, ensuring that the item with the smallest priority value is removed first. The push() and pop() operations run in O(log n) time, while peek() operates in O(1) time. This implementation efficiently manages tasks based on priority rather than insertion order.

Task Description #8 – Deque

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

Prompt:

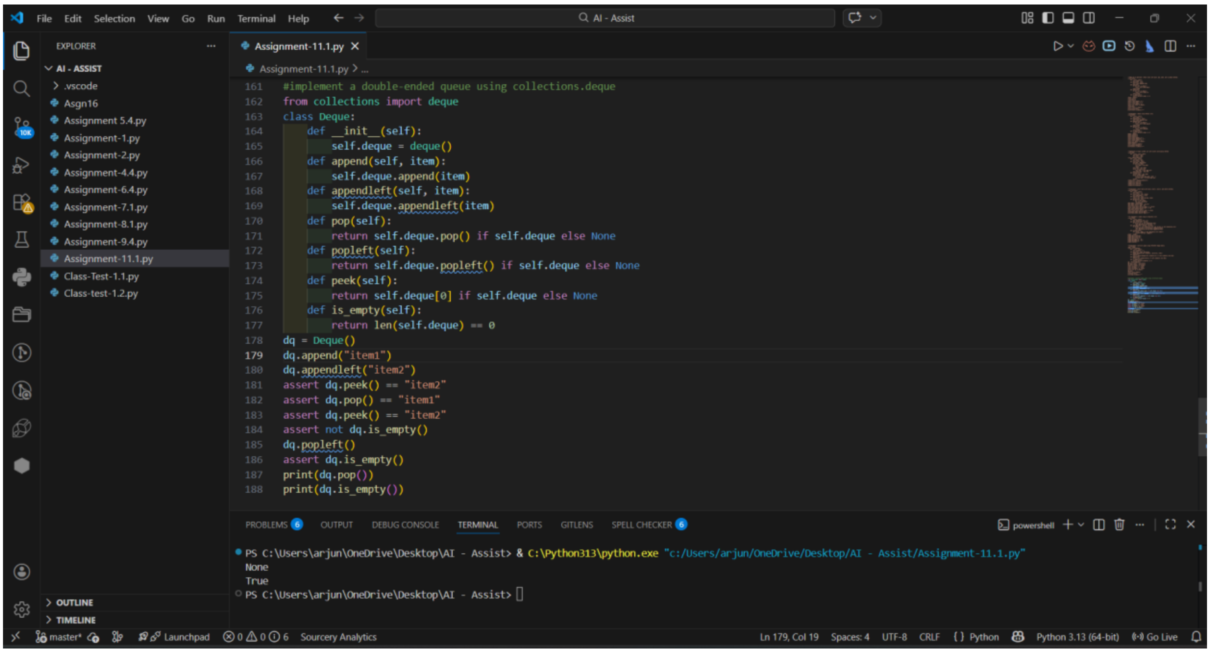
Use AI to generate a Python program that implements a Double-Ended Queue (Deque) using Python's built-in collections.deque module.

The class should include:

- append(item) – Add element to the right end
- appendleft(item) – Add element to the left end
- pop() – Remove and return element from the right end
- popleft() – Remove and return element from the left end
- peek() – View the front element
- is_empty() – Check whether the deque is empty

The implementation should allow insertion and deletion from both ends efficiently.

Code:



Observation:

The implemented Deque uses Python’s collections.deque, which allows efficient insertion and deletion from both ends in O(1) time. The append() and appendleft() methods add elements to the rear and front respectively, while pop() and popleft() remove elements from both ends. The peek() method retrieves the front element without removing it. This structure is flexible and combines features of both Stack and Queue.

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

Scenario:

Your college wants to develop a Campus Resource Management System

that handles:

1. Student Attendance Tracking – Daily log of students

entering/exiting the campus.

2. Event Registration System – Manage participants in events with quick search and removal.

3. Library Book Borrowing – Keep track of available books and their due dates.

4. Bus Scheduling System – Maintain bus routes and stop

connections.

5. Cafeteria Order Queue – Serve students in the order they arrive.

Prompt:

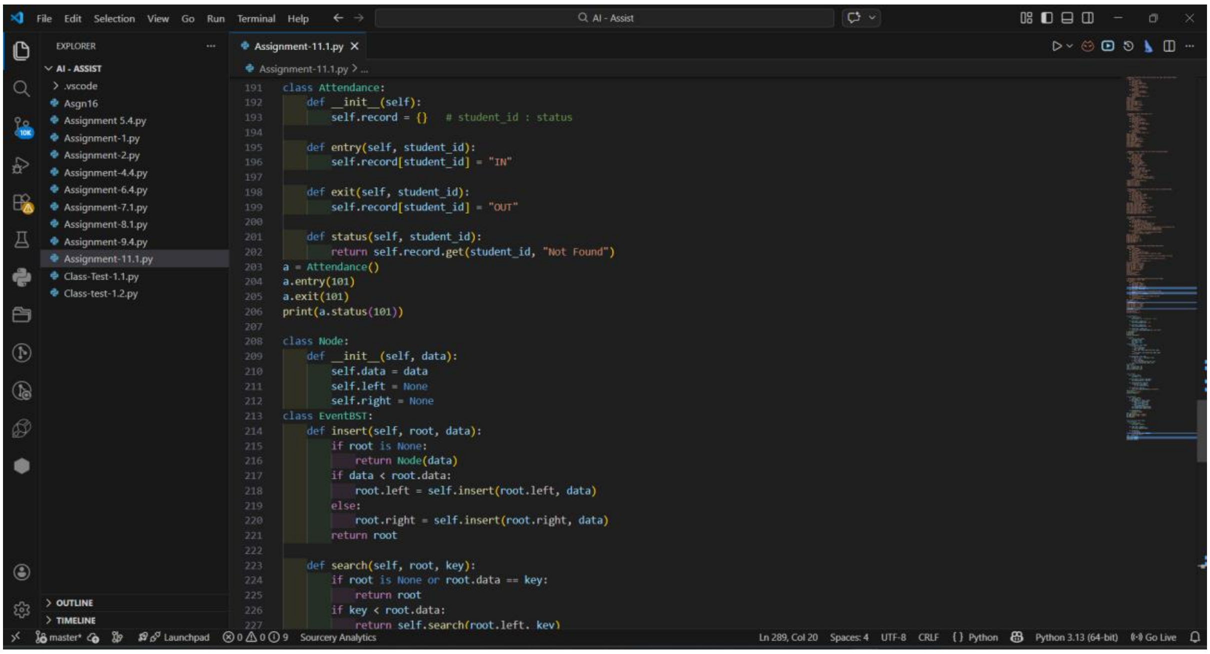
Design and implement a Campus Resource Management System that efficiently manages multiple campus operations using appropriate data structures.
Each feature must use a suitable data structure based on its operational requirements such as fast search, insertion, deletion, ordering, or connectivity handling.

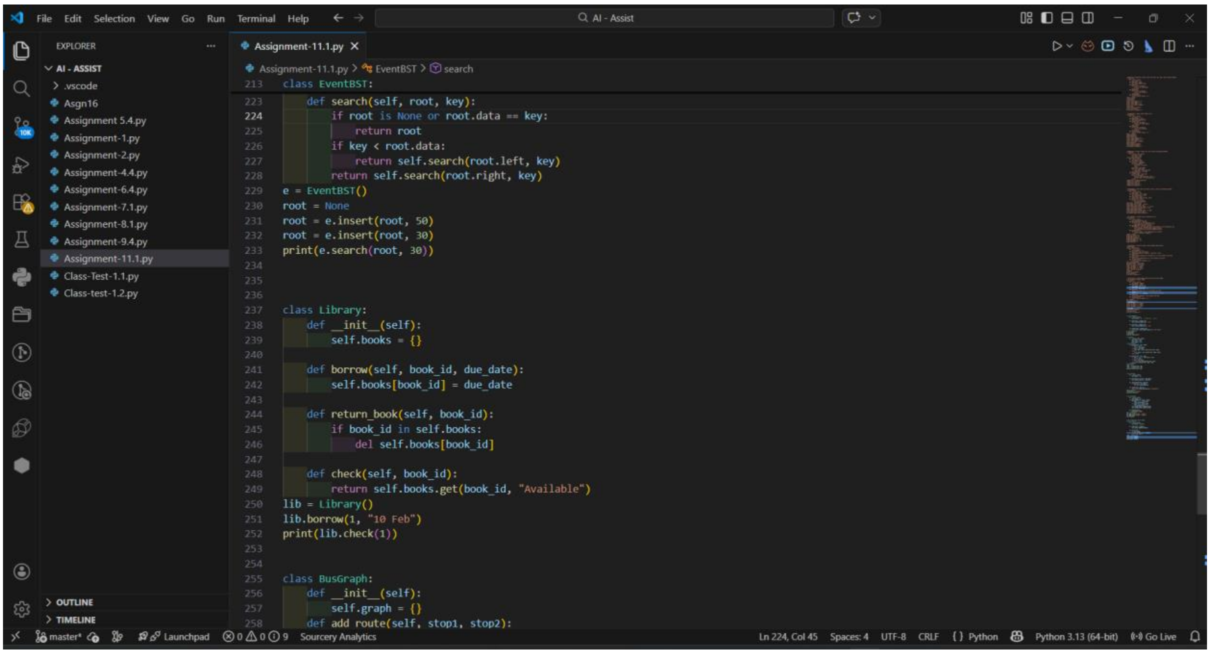
The system includes:

1. Student Attendance Tracking
2. Event Registration System
3. Library Book Borrowing
4. Bus Scheduling System
5. Cafeteria Order Queue

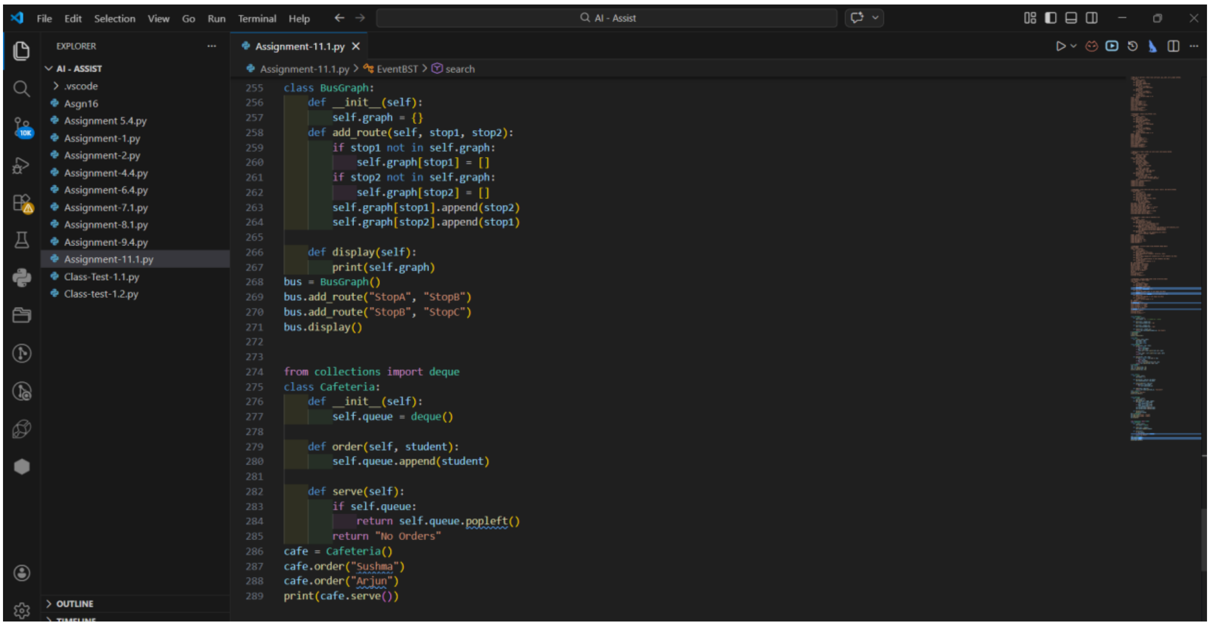
Each module is implemented independently using the most efficient data structure for that task.

Code:

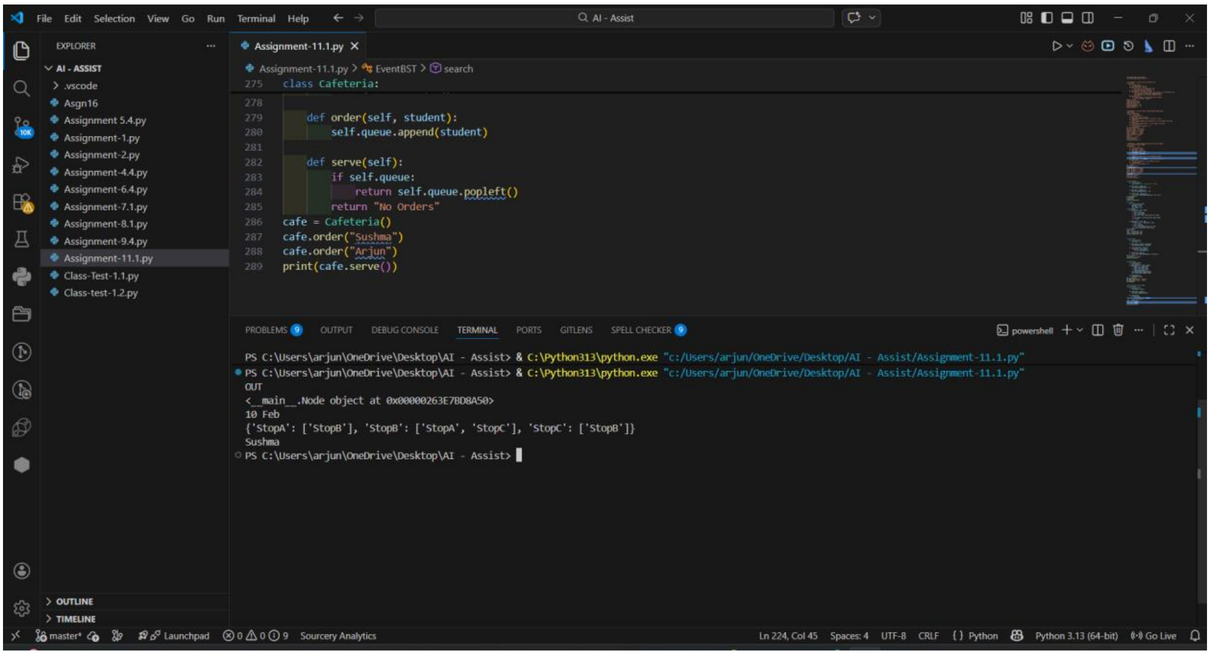




```python
class EventBST:
    def search(self, root, key):
        if root is None or root.data == key:
            return root
        if key < root.data:
            return self.search(root.left, key)
        return self.search(root.right, key)
e = EventBST()
root = None
root = e.insert(root, 50)
root = e.insert(root, 30)
print(e.search(root, 30))


class Library:
    def __init__(self):
        self.books = {}

    def borrow(self, book_id, due_date):
        self.books[book_id] = due_date

    def return_book(self, book_id):
        if book_id in self.books:
            del self.books[book_id]

    def check(self, book_id):
        return self.books.get(book_id, "Available")
lib = Library()
lib.borrow(1, "10 Feb")
print(lib.check(1))


class BusGraph:
    def __init__(self):
        self.graph = {}
    def add_route(self, stop1, stop2):
```



```python
class BusGraph:
    def __init__(self):
        self.graph = {}
    def add_route(self, stop1, stop2):
        if stop1 not in self.graph:
            self.graph[stop1] = []
        if stop2 not in self.graph:
            self.graph[stop2] = []
        self.graph[stop1].append(stop2)
        self.graph[stop2].append(stop1)

    def display(self):
        print(self.graph)
bus = BusGraph()
bus.add_route("StopA", "StopB")
bus.add_route("StopB", "StopC")
bus.display()


from collections import deque
class Cafeteria:
    def __init__(self):
        self.queue = deque()

    def order(self, student):
        self.queue.append(student)

    def serve(self):
        if self.queue:
            return self.queue.popleft()
        return "No Orders"
cafe = Cafeteria()
cafe.order("Sushma")
cafe.order("Arjun")
print(cafe.serve())
```



Observation:

The system demonstrates how different real-world problems require different data structures for optimal performance.

- Hash Tables provide constant-time access for tracking attendance and books.
- Binary Search Tree enables structured storage with efficient search and removal.
- Graph effectively models interconnected bus routes.
- Queue ensures fair servicing in cafeteria using FIFO principle.

This implementation shows that choosing the correct data structure improves efficiency, maintainability, and scalability of a system.

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.

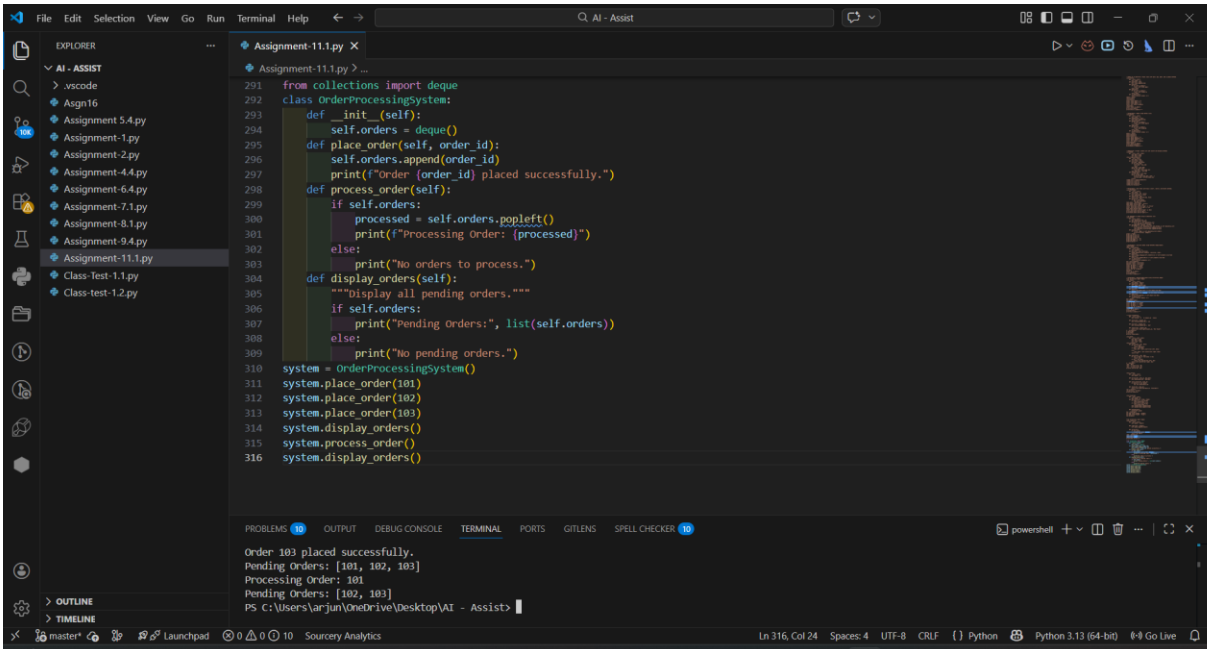
5. Delivery Route Planning – Connect warehouses and delivery

locations.

Prompt:

Design a Smart E-Commerce Platform that efficiently manages shopping operations using suitable data structures. Each feature must use the most appropriate data structure based on operational requirements such as dynamic insertion, ranking, fast searching, order processing, and network connectivity.

Code:



Observation:

Different modules in an e-commerce system require different data structures to achieve efficiency. Queue ensures fair order processing, Linked List allows flexible cart updates, Priority Queue helps in ranking top-selling products, Hash Table enables instant product lookup, and Graph models delivery routes effectively. Selecting the correct data structure improves system performance and scalability.