ASSIGNMENT-11.1

Name:-P. Likitha

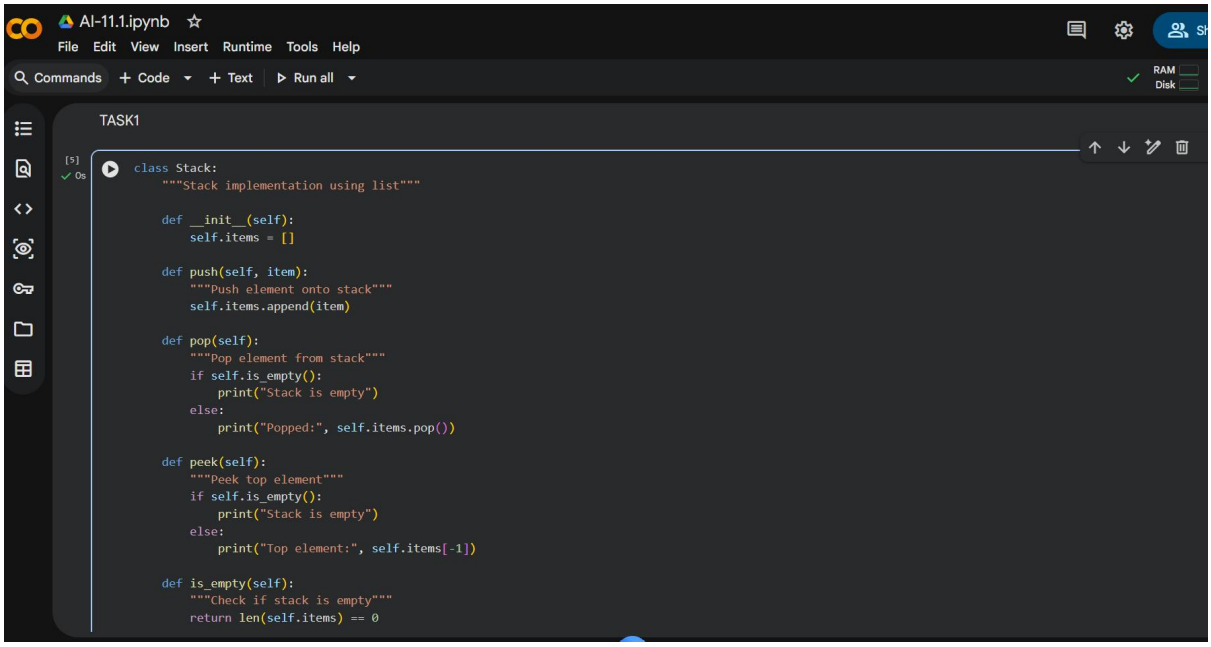
HT.NO:- 2303A52393

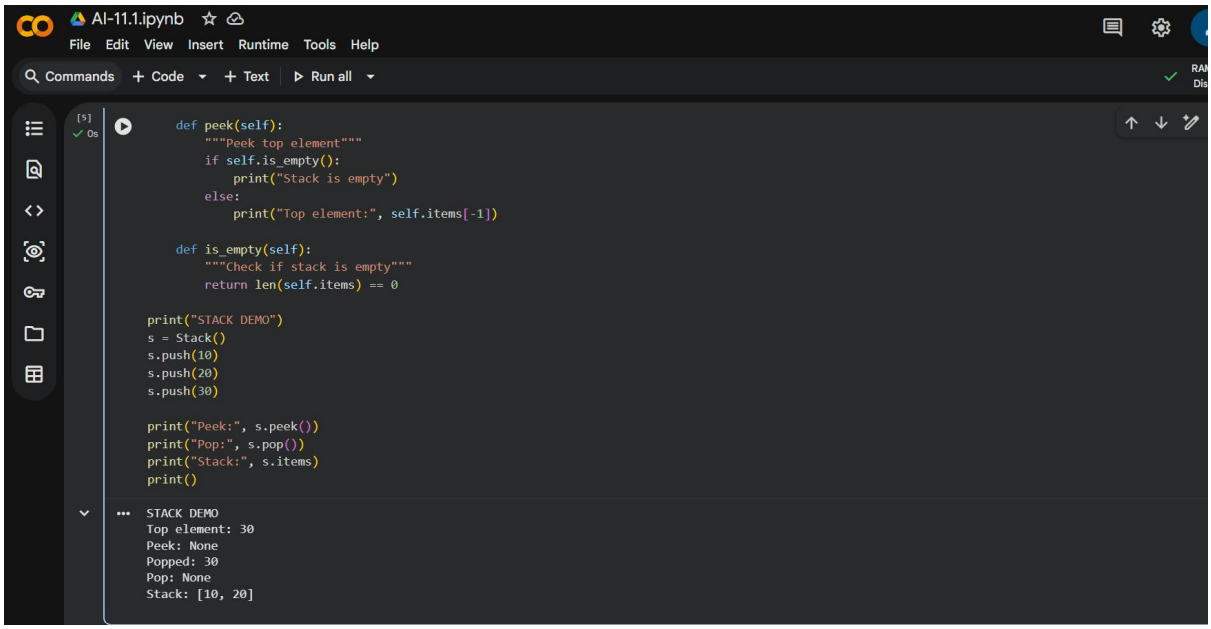
Batch:- 43

Task 1 – Stack Implementation

Prompt:-

The task is to use AI to generate a Stack class in Python. The stack should be implemented using a Python list and must support push, pop, peek, and is_empty operations. The implementation should strictly follow the LIFO (Last In First Out) principle. Each method should be clearly defined and include proper docstrings explaining its purpose and functionality. The generated code should be simple, readable, and suitable for academic use.





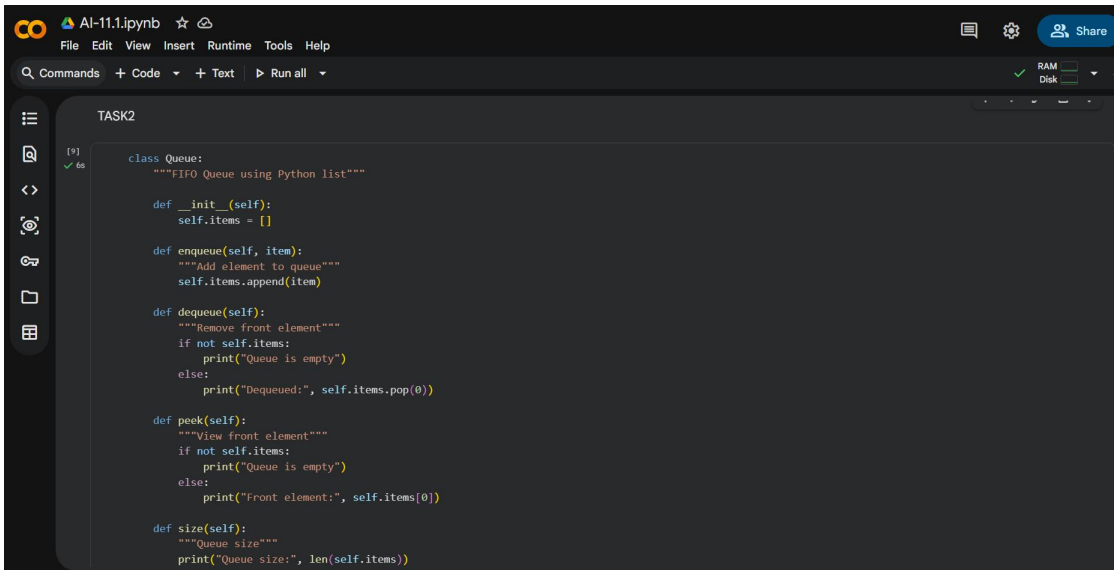
Observation :-

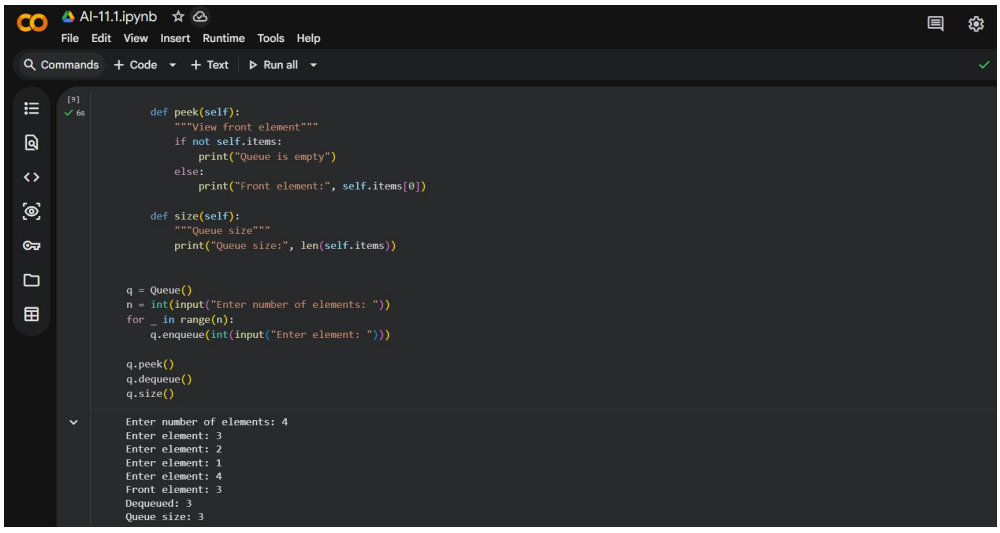
The generated Stack implementation works correctly and follows the LIFO principle. The push method adds elements to the top of the stack, while the pop method removes the most recently added element. The peek method returns the top element without deleting it. The is_empty method accurately checks whether the stack contains elements. Overall, the stack behaves as expected and is well-documented.

Task 2 – Queue Implementation

Prompt:

The objective of this task is to use AI to implement a Queue using Python lists. The queue should operate based on the FIFO (First In First Out) principle. It must include enqueue, dequeue, peek, and size methods. The implementation should handle empty queue conditions properly and include docstrings for better understanding. The code should demonstrate clear queue behavior.

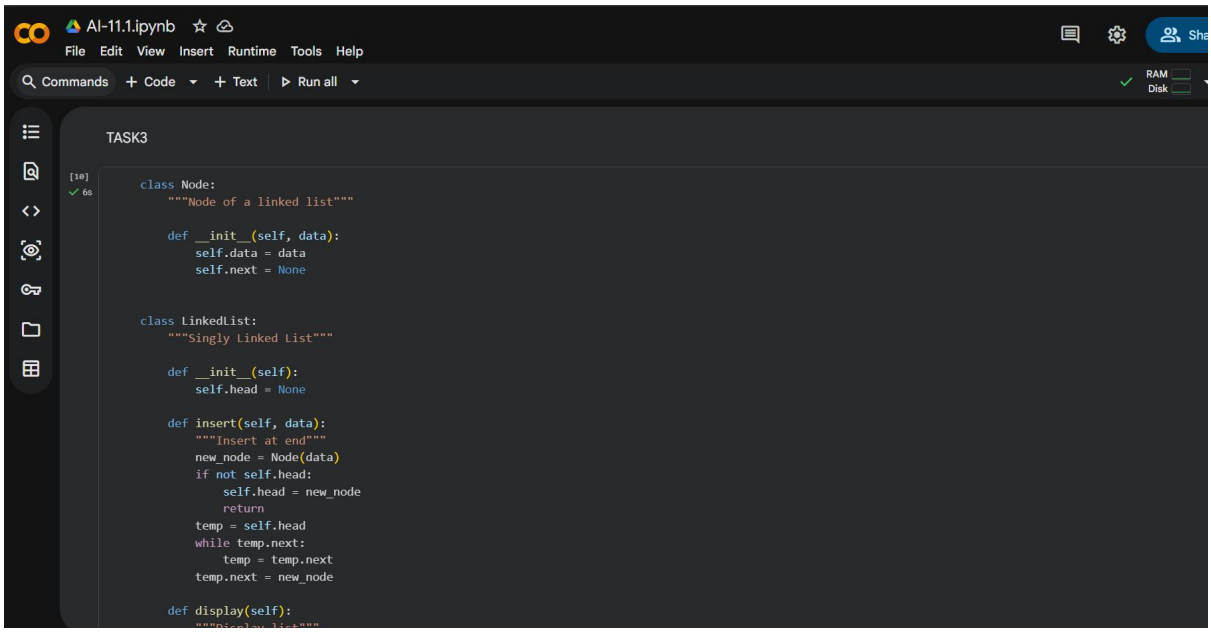


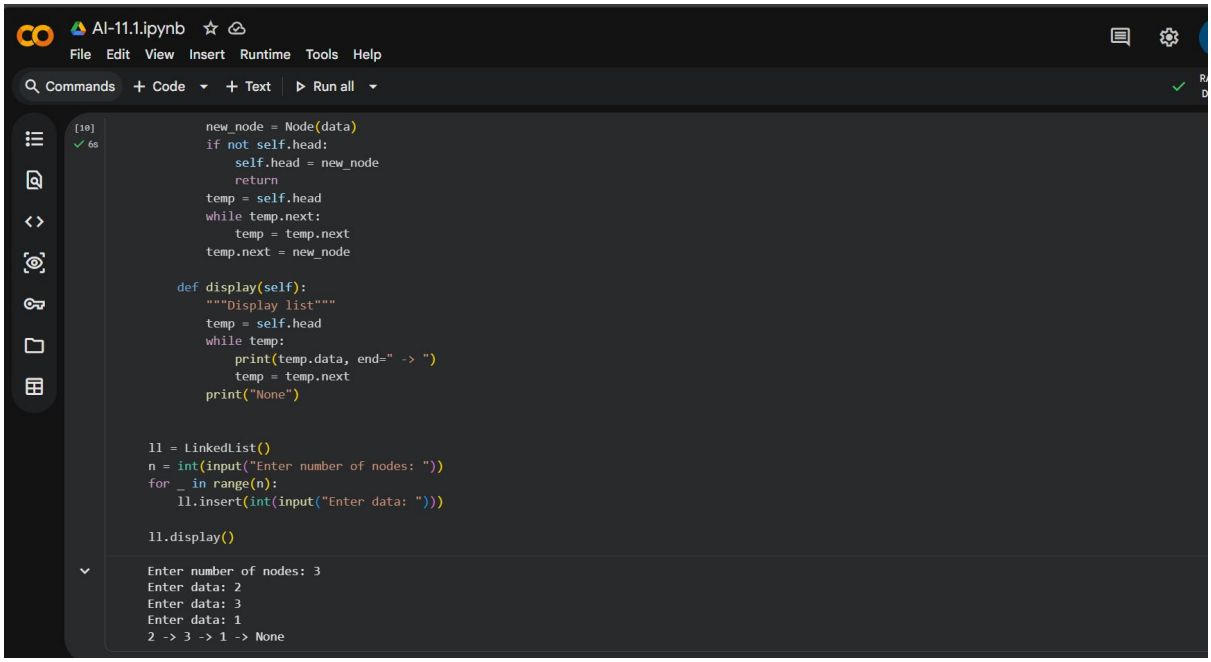


Observation :
The Queue implementation follows the FIFO principle correctly. Elements are added to the rear of the queue using the enqueue method and removed from the front using the dequeue method. The peek method displays the front element without removing it. The size method correctly returns the number of elements present in the queue. The implementation is efficient and easy to understand.

Task 3 – Singly Linked List

Prompt:
This task requires using AI to generate a Singly Linked List in Python. A Node class should be created to store data and the reference to the next node. A LinkedList class should be implemented with insert and display methods. The insert operation should add new nodes at the end of the list. Proper documentation should be included for clarity.

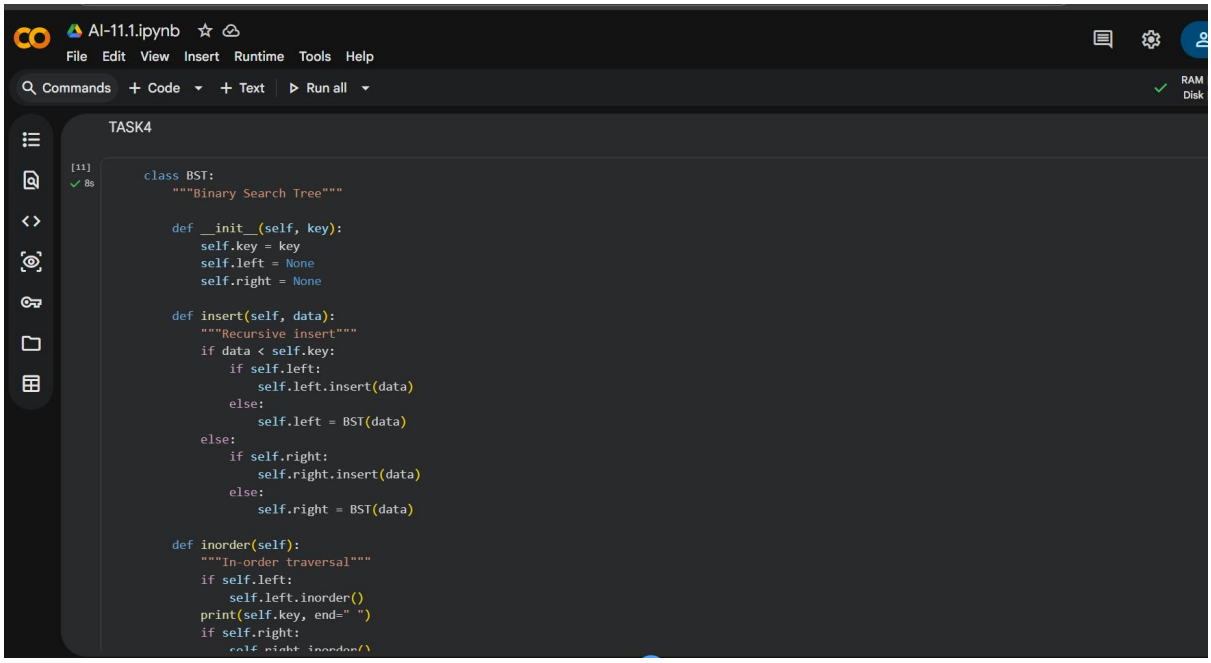


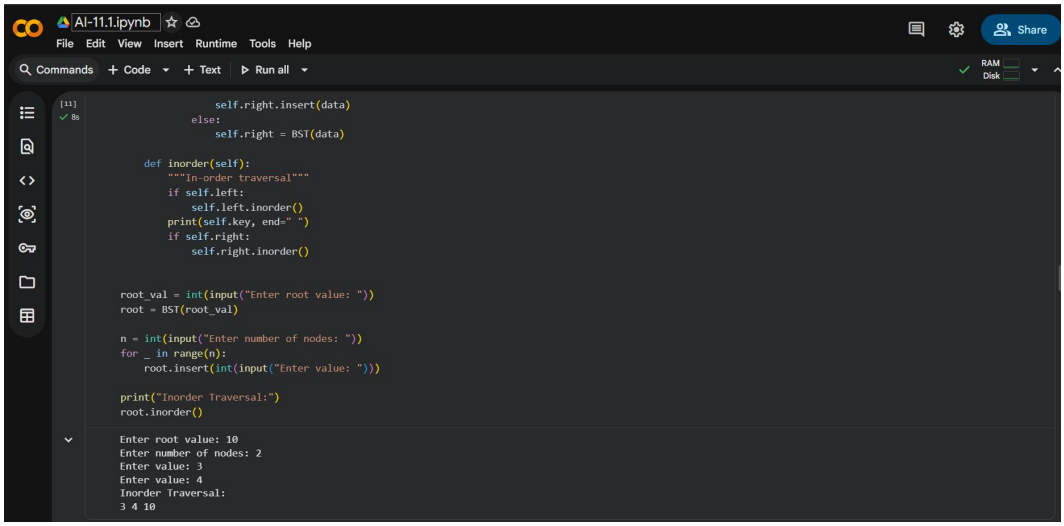


Observation:
The generated linked list implementation works as expected. Nodes are connected sequentially using references, allowing dynamic memory usage. The insert method successfully adds elements to the end of the list. The display method prints all elements in order. This implementation clearly demonstrates how linked lists manage linear data efficiently.

Task 4 – Binary Search Tree (BST)

Prompt :
The aim of this task is to use AI to implement a Binary Search Tree in Python. The BST should include recursive insert functionality and an in-order traversal method. The tree must follow BST properties, where smaller values are placed in the left subtree and larger values in the right subtree. The implementation should include method-level documentation.

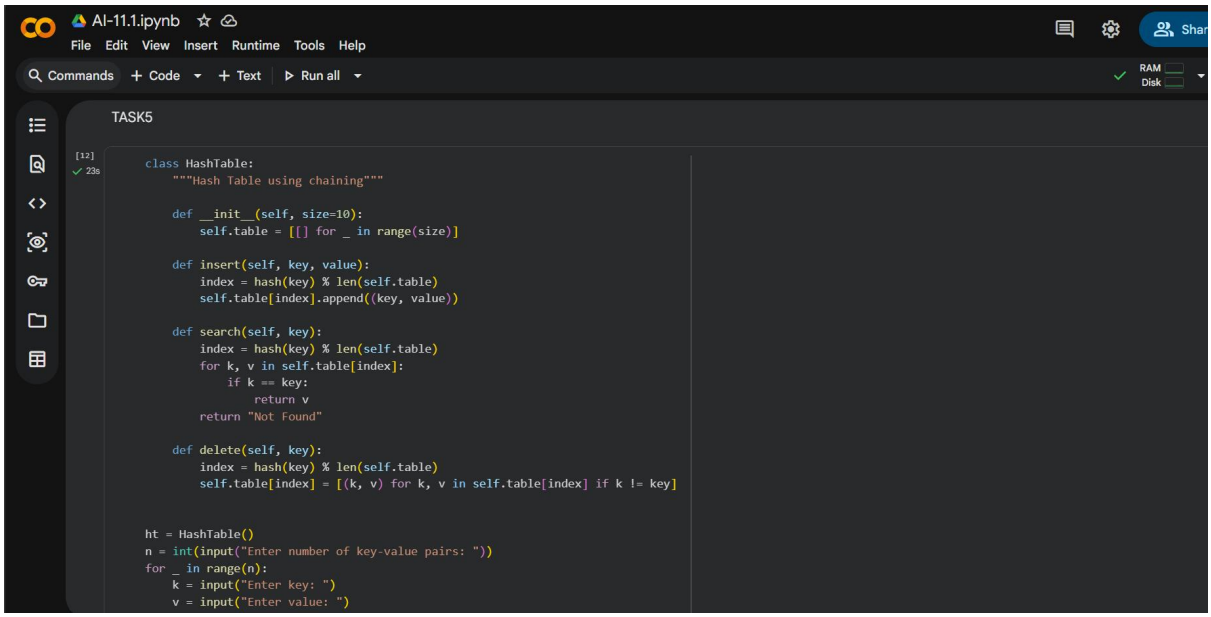


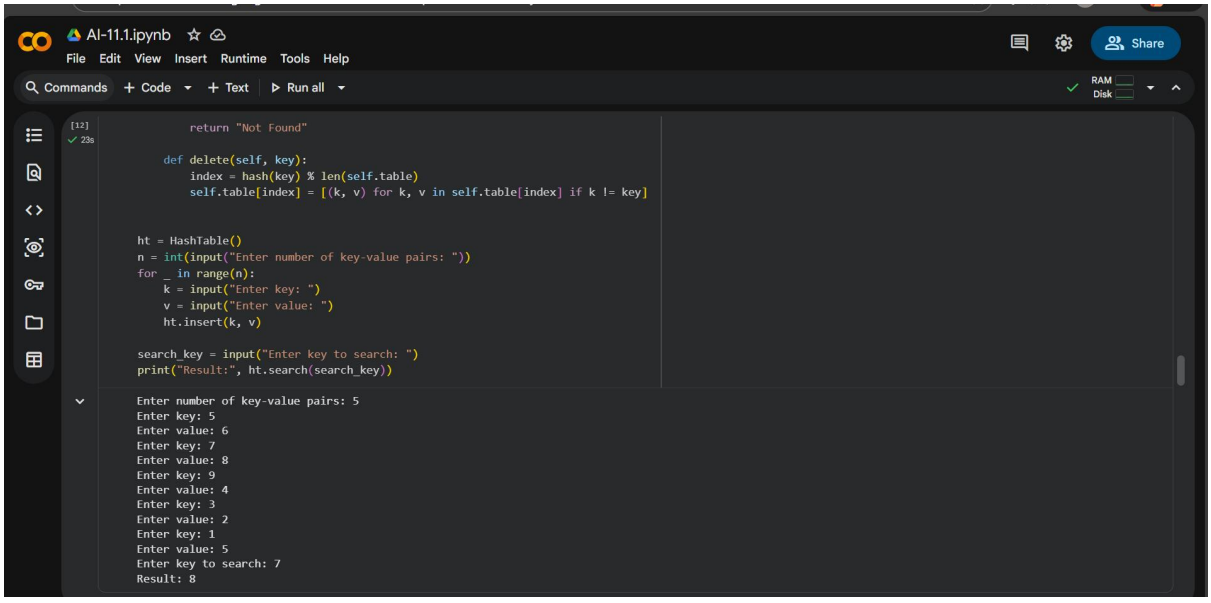


Observation:
The Binary Search Tree implementation maintains elements in a sorted structure. Recursive insertion correctly places each value in its appropriate position. The in-order traversal prints the elements in ascending order. This confirms that the BST properties are preserved. The recursive approach simplifies traversal and insertion logic.

Task 5 – Hash Table

Prompt :
This task involves using AI to implement a Hash Table in Python. The hash table should support basic operations such as insert, search, and delete. Collision handling must be implemented using the chaining technique. Python's built-in hash function should be used to compute indices. Proper comments should explain how collisions are handled.

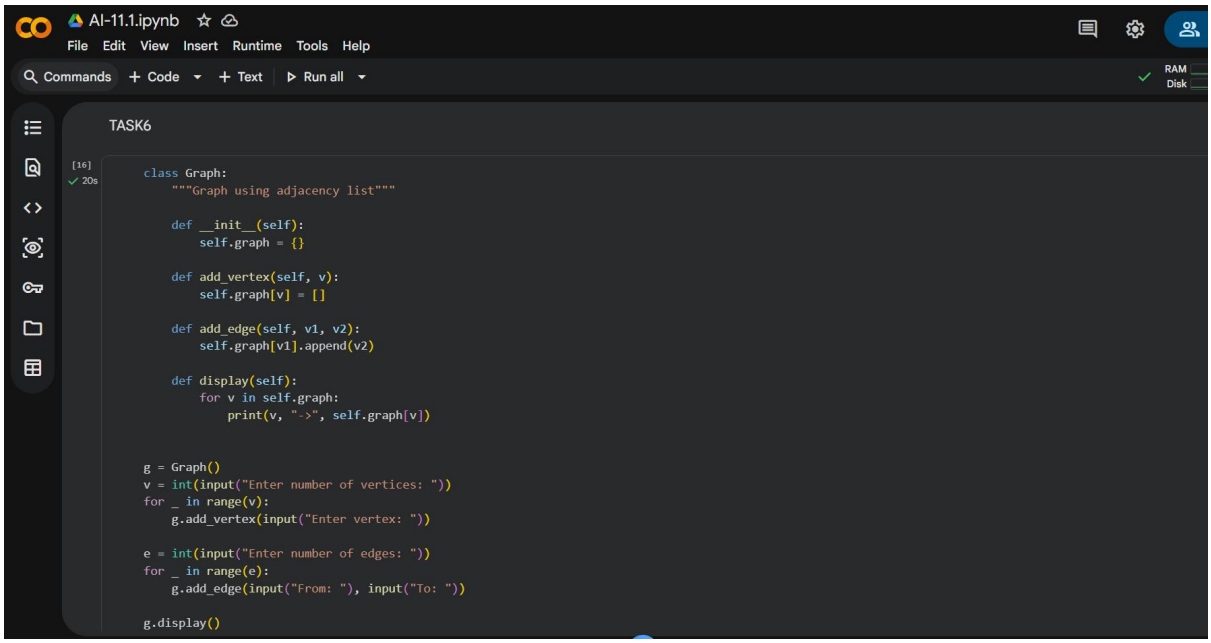


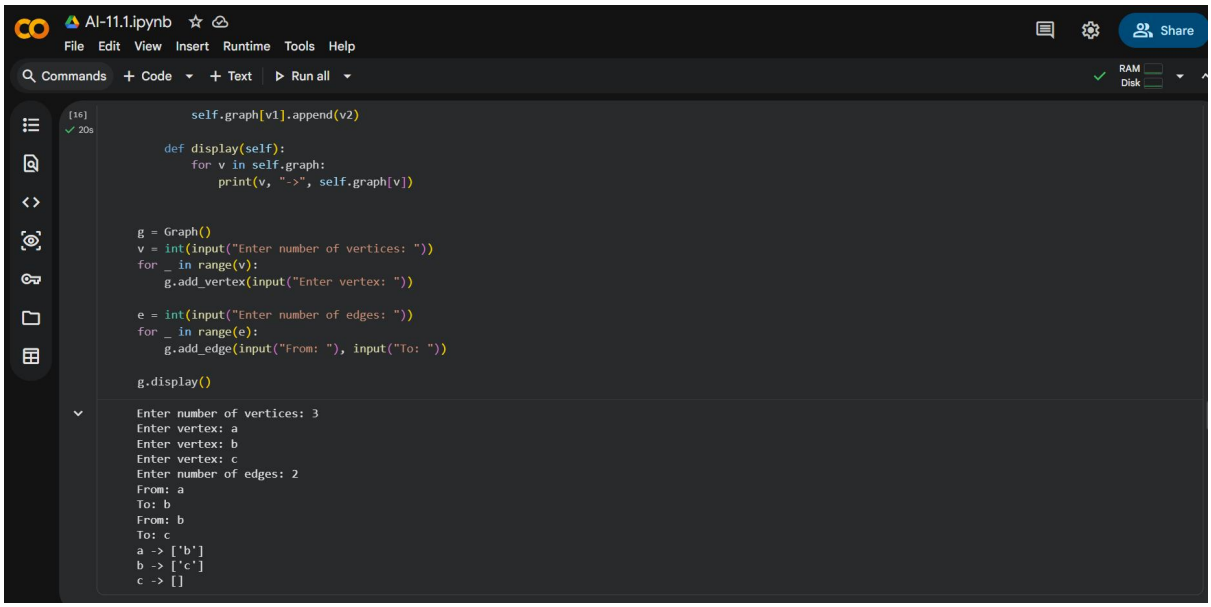


Observation:
The hash table implementation efficiently stores key-value pairs. Collisions are handled using chaining, where multiple elements are stored in the same bucket. The insert method places elements correctly, and the search operation retrieves values efficiently. The delete method removes keys without affecting other entries. The implementation demonstrates effective hashing concepts.

Task 6 – Graph Representation

Prompt:
The task is to use AI to implement a graph using an adjacency list representation. The graph should allow adding vertices and edges dynamically. A dictionary-based structure should be used to store connections. A display method should show all vertices and their connected edges. Proper documentation should be included.

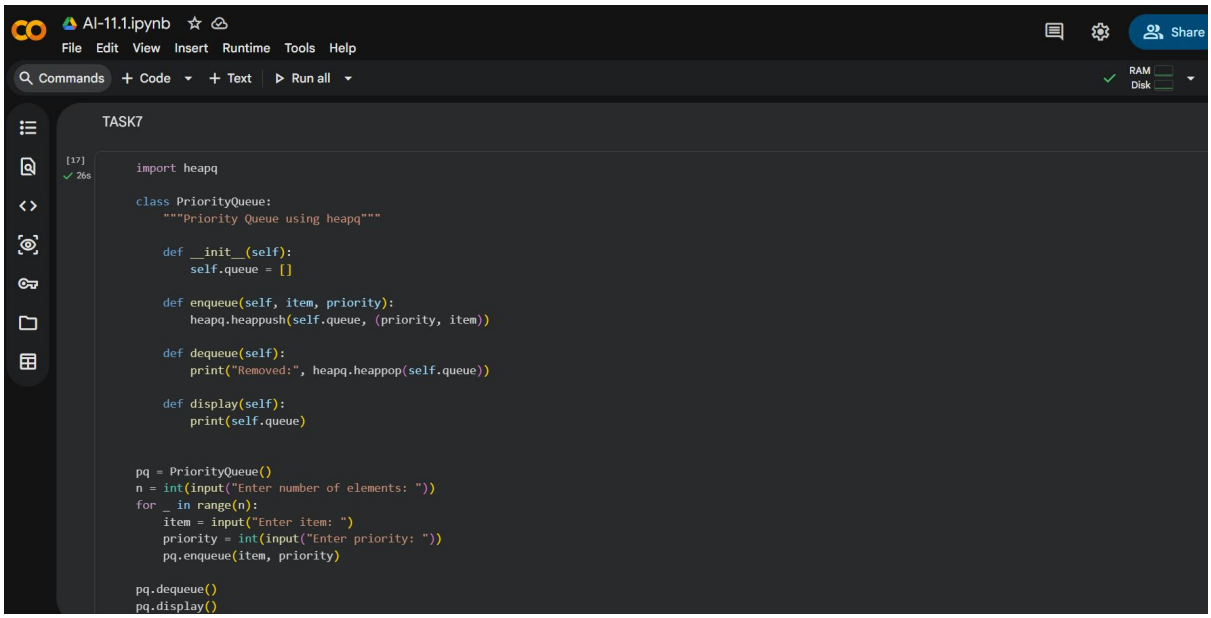




Observation:
The graph implementation correctly represents relationships using adjacency lists. Each vertex maintains a list of connected vertices. The add vertex and add edge methods work correctly. The display method clearly shows the structure of the graph. This representation is efficient for modeling real-world networks.

Task 7 – Priority Queue

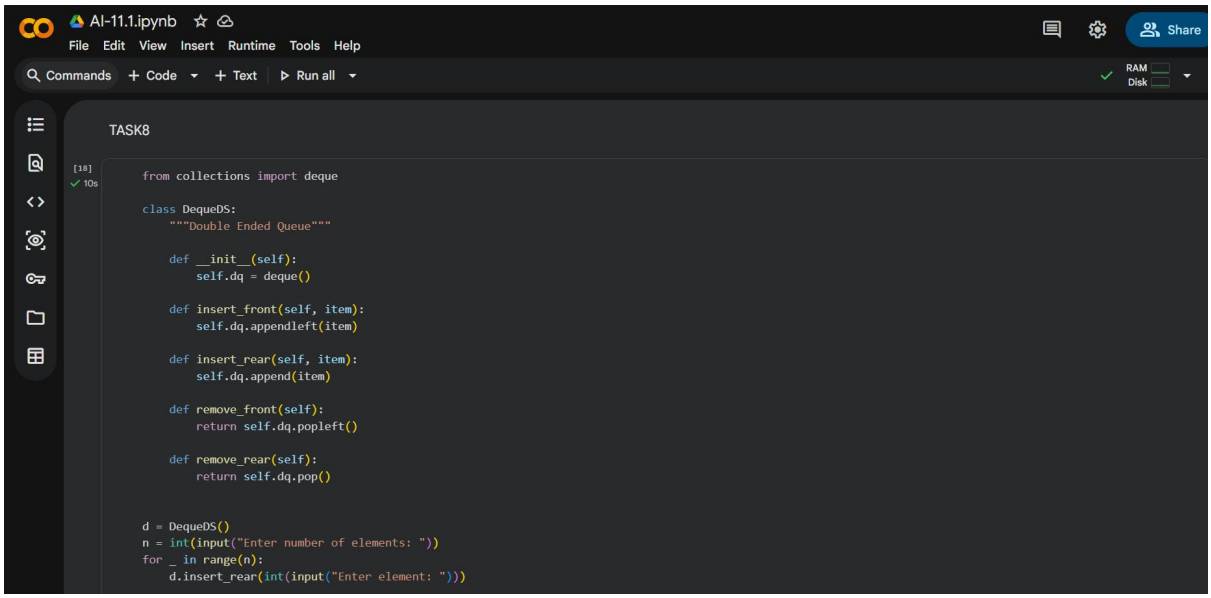
Prompt:
This task requires using AI to implement a Priority Queue in Python with the help of the heapq module. The queue should allow insertion of elements along with priority values. The dequeue operation should always remove the highest priority element. A display method should show the current state of the queue. Documentation should be included.



Observation :
The priority queue implementation correctly orders elements based on priority. The heapq module ensures efficient insertion and removal operations. Elements with the highest priority are dequeued first. The structure is suitable for scheduling and task management applications. The implementation is efficient and reliable.

Task 8 – Deque

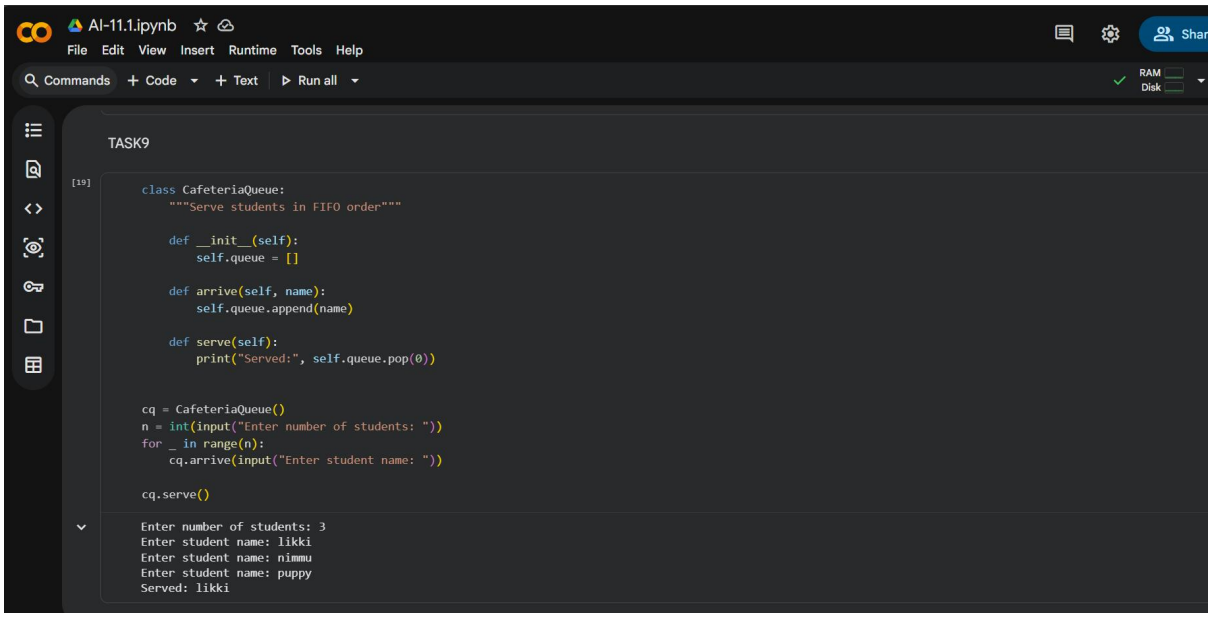
Prompt:
The objective of this task is to use AI to implement a double-ended queue using collections.deque. The deque should support insertion and removal of elements from both front and rear ends. The implementation should demonstrate efficient operations. Clear docstrings should explain each method.



Observation:
The deque implementation allows flexible insertion and deletion from both ends. Operations are performed efficiently in constant time. This structure is more efficient than list-based implementations for such operations. The deque is suitable for applications like sliding window problems. The code is well-structured and documented.

Task 9 – Campus Resource Management System

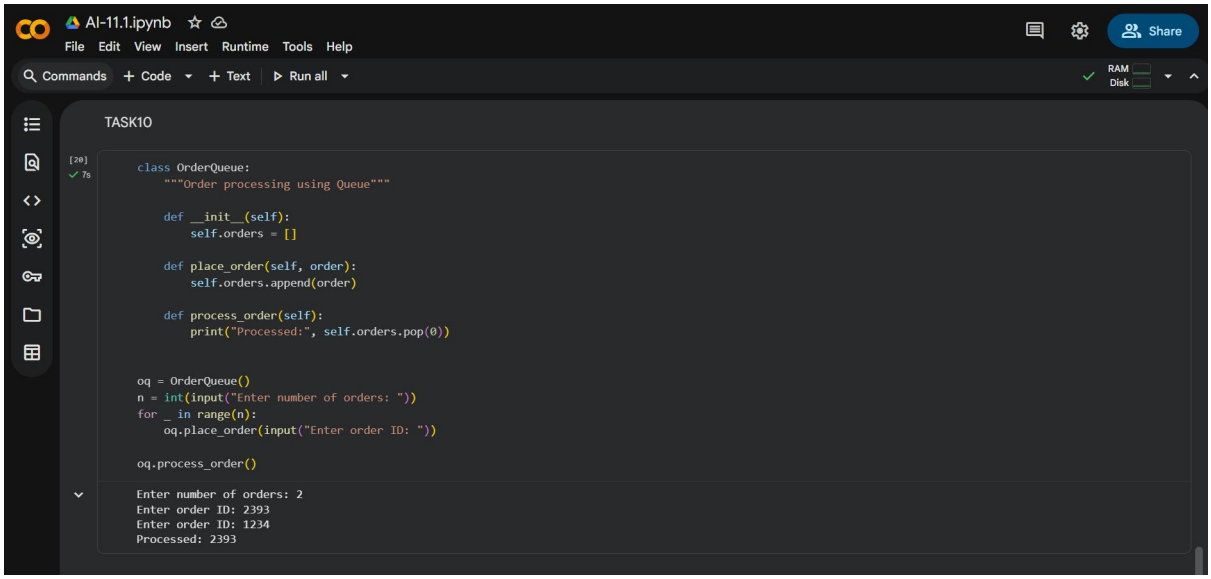
Prompt:
This task involves using AI to choose appropriate data structures for different campus management features. Each feature should be analyzed based on its requirements such as ordering, searching, and connectivity. The most suitable data structure should be selected and justified. One feature should be implemented using Python with proper documentation.



Observation:
Each campus feature is mapped to an optimal data structure. Queues efficiently handle cafeteria orders, while graphs represent bus routes effectively. Hash tables enable fast searching for event registrations. The selected data structures improve performance and scalability. The implemented feature demonstrates practical real-world usage.

Task 10 – Smart E-Commerce Platform

Prompt:
The task is to use AI to analyze data structure requirements for a smart e-commerce platform. Suitable data structures should be selected for cart management, order processing, search, and delivery planning. Each selection should be justified clearly. One feature should be implemented using Python with comments and docstrings.



Observation:

Different e-commerce features require different data structures for optimal performance. Queues ensure orders are processed in the correct sequence. Hash tables provide fast product lookup. Priority queues help track top-selling products efficiently. The overall design improves system speed and reliability.