

## ASSIGNMENT-11.1

Name:-P. Likitha

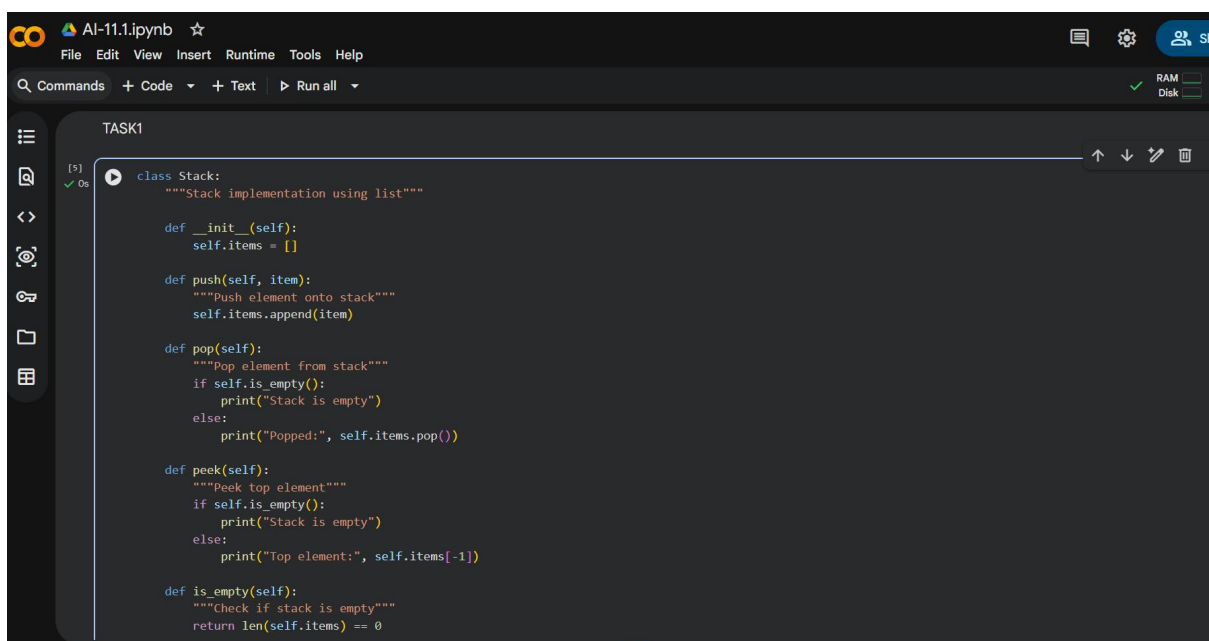
HT.NO:- 2303A52393

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.



```
class Stack:
    """Stack implementation using list"""

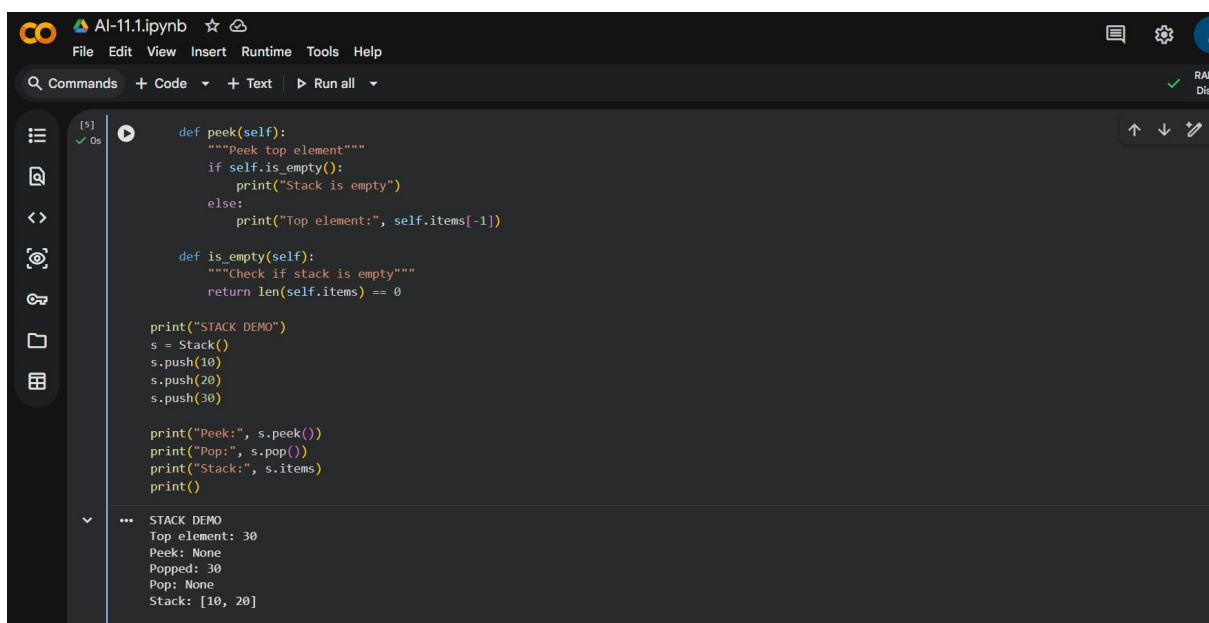
    def __init__(self):
        self.items = []

    def push(self, item):
        """Push element onto stack"""
        self.items.append(item)

    def pop(self):
        """Pop element from stack"""
        if self.is_empty():
            print("Stack is empty")
        else:
            print("Popped:", self.items.pop())

    def peek(self):
        """Peek top element"""
        if self.is_empty():
            print("Stack is empty")
        else:
            print("Top element:", self.items[-1])

    def is_empty(self):
        """Check if stack is empty"""
        return len(self.items) == 0
```



```
def peek(self):
    """Peek top element"""
    if self.is_empty():
        print("Stack is empty")
    else:
        print("Top element:", self.items[-1])

def is_empty(self):
    """Check if stack is empty"""
    return len(self.items) == 0

print("STACK DEMO")
s = Stack()
s.push(10)
s.push(20)
s.push(30)

print("Peek:", s.peek())
print("Pop:", s.pop())
print("Stack:", s.items)
print()
```

... STACK DEMO  
Top element: 30  
Peek: None  
Popped: 30  
Pop: None  
Stack: [10, 20]

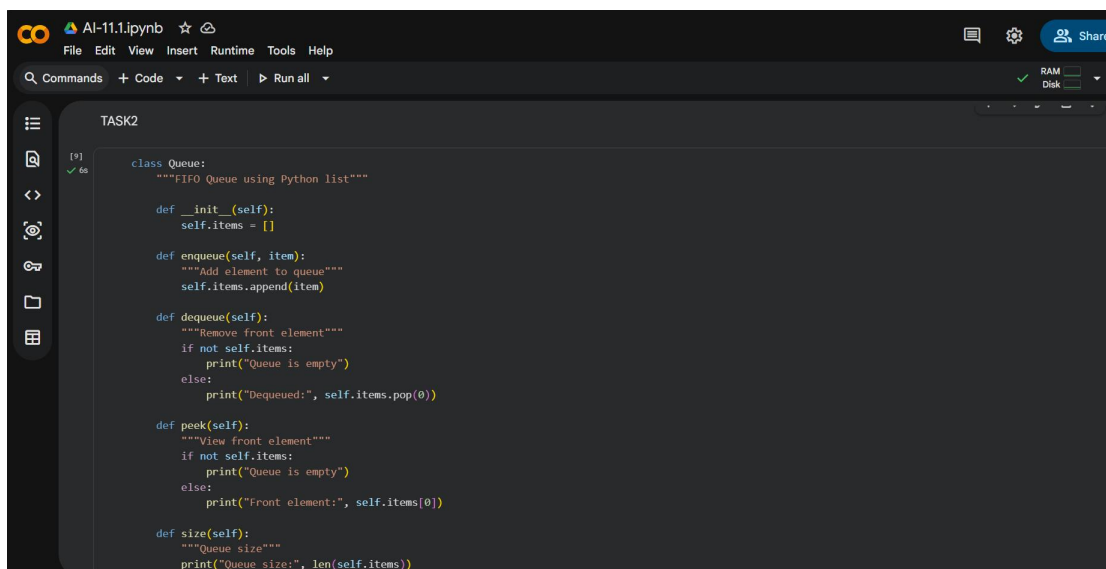
## Observation :-

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.



```
class Queue:
    """FIFO Queue using Python list"""

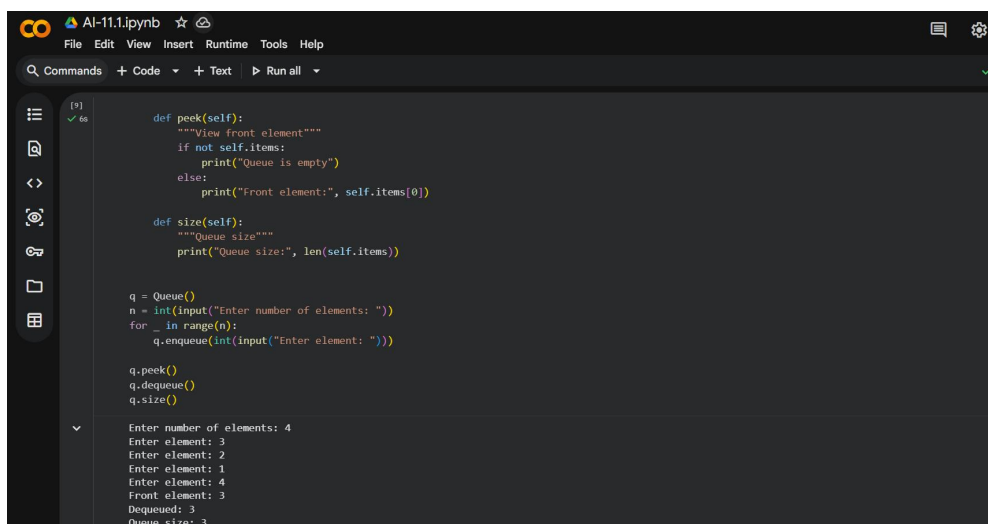
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        """Add element to queue"""
        self.items.append(item)

    def dequeue(self):
        """Remove front element"""
        if not self.items:
            print("Queue is empty")
        else:
            print("Dequeued:", self.items.pop(0))

    def peek(self):
        """View front element"""
        if not self.items:
            print("Queue is empty")
        else:
            print("Front element:", self.items[0])

    def size(self):
        """Queue size"""
        print("Queue size:", len(self.items))
```



```
def peek(self):
    """View front element"""
    if not self.items:
        print("Queue is empty")
    else:
        print("Front element:", self.items[0])

def size(self):
    """Queue size"""
    print("Queue size:", len(self.items))

q = Queue()
n = int(input("Enter number of elements: "))
for _ in range(n):
    q.enqueue(int(input("Enter element: ")))

q.peek()
q.dequeue()
q.size()

Enter number of elements: 4
Enter element: 3
Enter element: 2
Enter element: 1
Enter element: 4
Front element: 3
Dequeued: 3
Queue size: 3
```

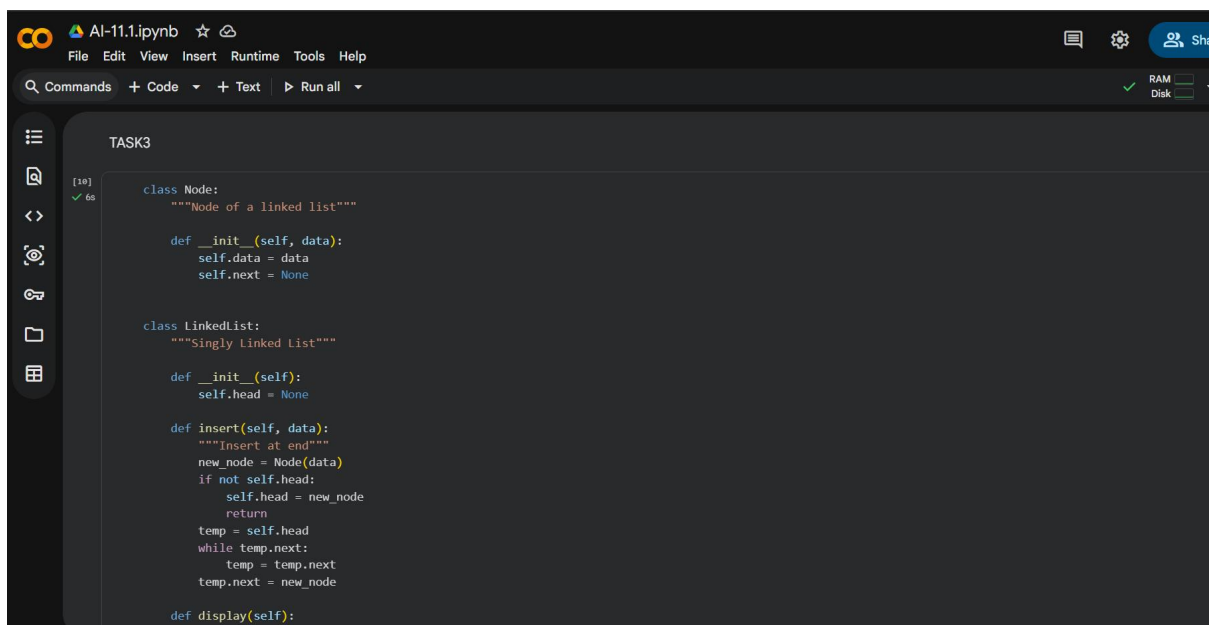
### Observation :

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.



```
[10] ✓ 6s
class Node:
    """Node of a linked list"""

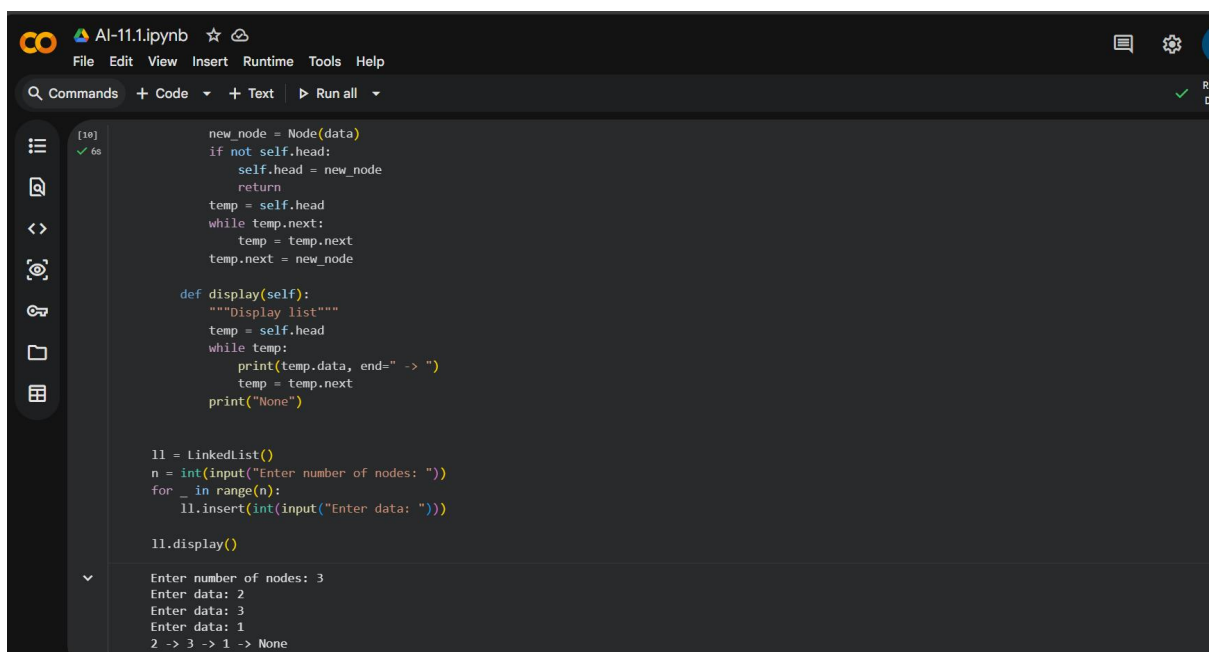
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    """Singly Linked List"""

    def __init__(self):
        self.head = None

    def insert(self, data):
        """Insert at end"""
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node

    def display(self):
        """Display list"""
```



```
[10] ✓ 6s
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node

    def display(self):
        """Display list"""
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")

l1 = LinkedList()
n = int(input("Enter number of nodes: "))
for _ in range(n):
    l1.insert(int(input("Enter data: ")))

l1.display()

Enter number of nodes: 3
Enter data: 2
Enter data: 3
Enter data: 1
2 -> 3 -> 1 -> None
```

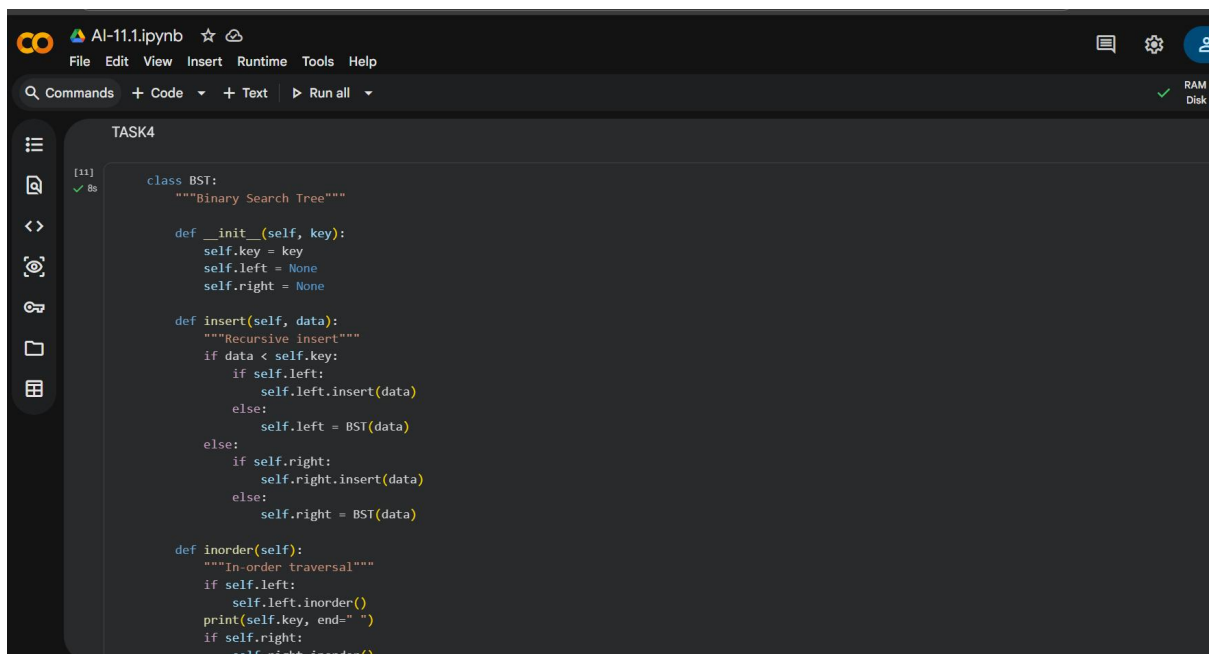
### Observation:

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.

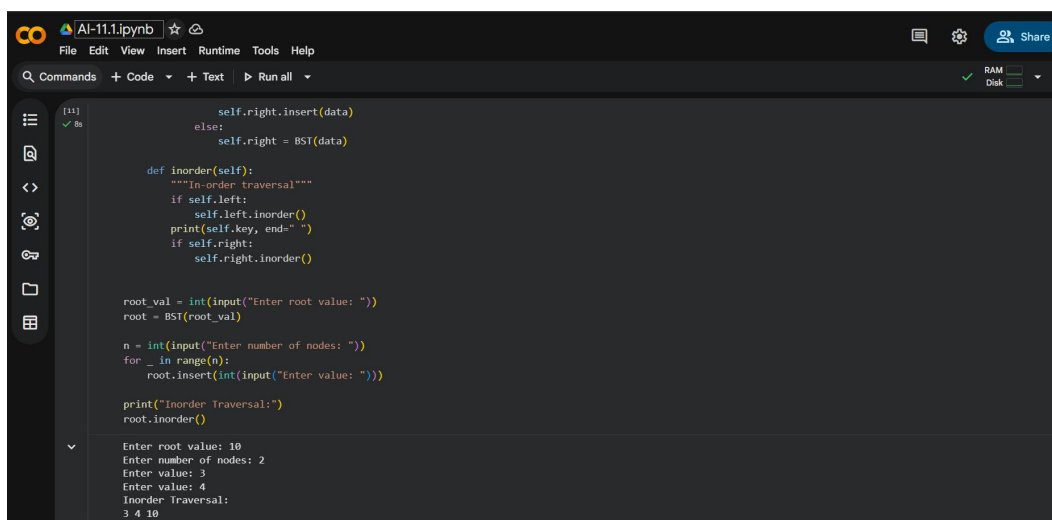


```
[11] ✓ Bs
class BST:
    """Binary Search Tree"""

    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

    def insert(self, data):
        """Recursive insert"""
        if data < self.key:
            if self.left:
                self.left.insert(data)
            else:
                self.left = BST(data)
        else:
            if self.right:
                self.right.insert(data)
            else:
                self.right = BST(data)

    def inorder(self):
        """In-order traversal"""
        if self.left:
            self.left.inorder()
        print(self.key, end=" ")
        if self.right:
            self.right.inorder()
```



```
[11] ✓ Bs
        self.right.insert(data)
    else:
        self.right = BST(data)

    def inorder(self):
        """In-order traversal"""
        if self.left:
            self.left.inorder()
        print(self.key, end=" ")
        if self.right:
            self.right.inorder()

root_val = int(input("Enter root value: "))
root = BST(root_val)

n = int(input("Enter number of nodes: "))
for _ in range(n):
    root.insert(int(input("Enter value: ")))

print("Inorder Traversal:")
root.inorder()

Enter root value: 10
Enter number of nodes: 2
Enter value: 3
Enter value: 4
Inorder Traversal:
3 4 10
```

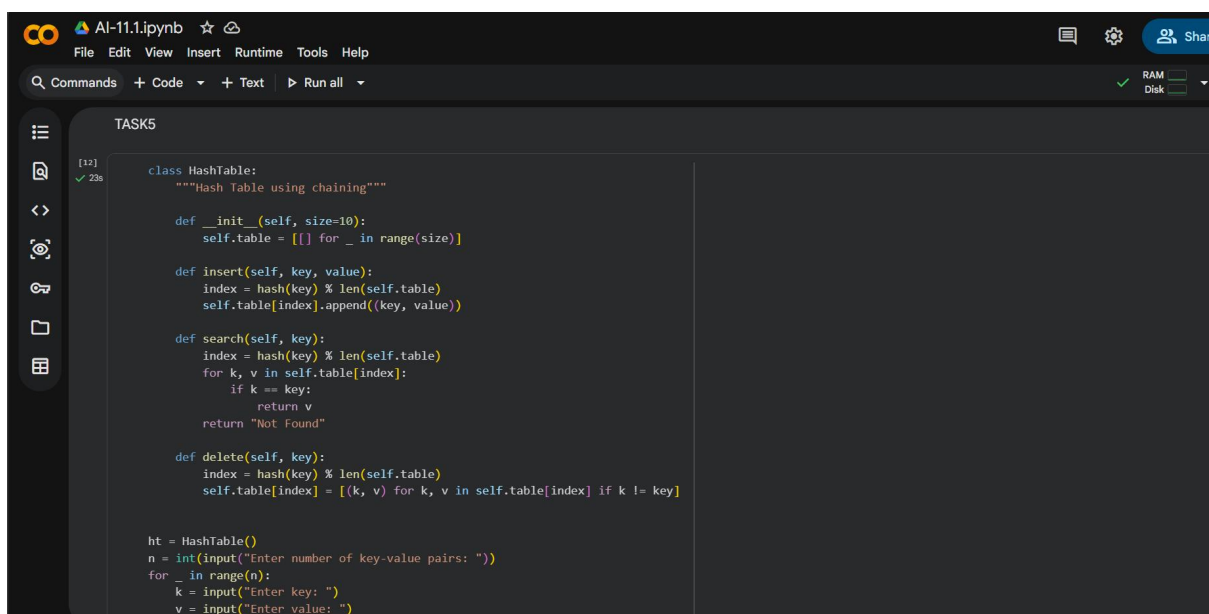
## Observation:

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.



```
[12]: class HashTable:
      """Hash Table using chaining"""

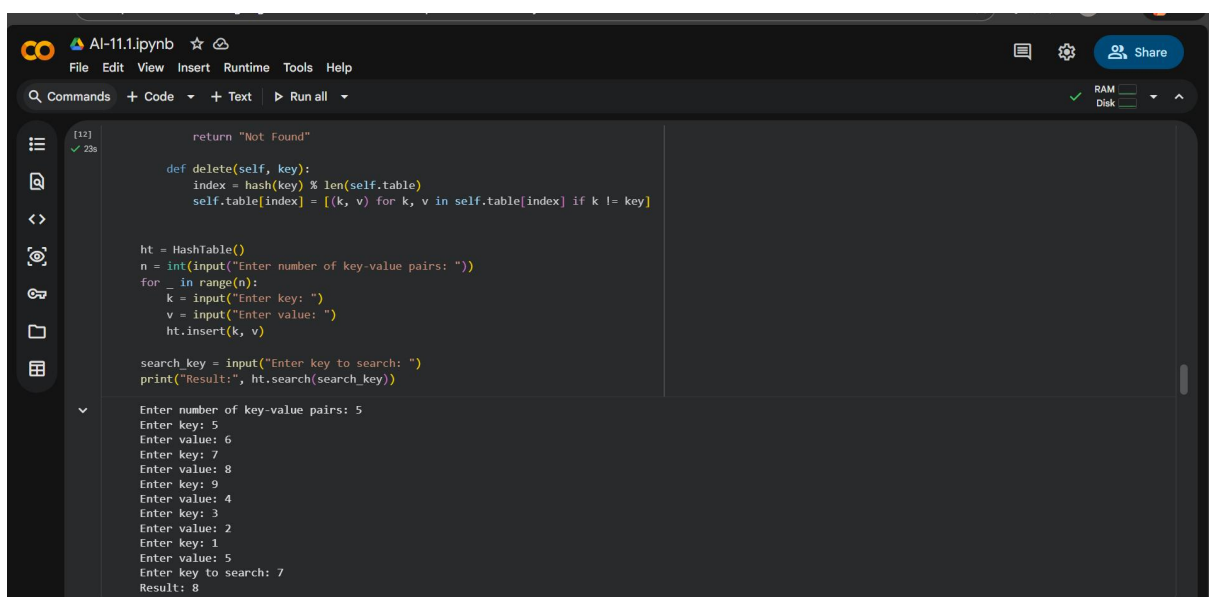
      def __init__(self, size=10):
          self.table = [[] for _ in range(size)]

      def insert(self, key, value):
          index = hash(key) % len(self.table)
          self.table[index].append((key, value))

      def search(self, key):
          index = hash(key) % len(self.table)
          for k, v in self.table[index]:
              if k == key:
                  return v
          return "Not Found"

      def delete(self, key):
          index = hash(key) % len(self.table)
          self.table[index] = [(k, v) for k, v in self.table[index] if k != key]

      ht = HashTable()
      n = int(input("Enter number of key-value pairs: "))
      for _ in range(n):
          k = input("Enter key: ")
          v = input("Enter value: ")
```



```
      return "Not Found"

      def delete(self, key):
          index = hash(key) % len(self.table)
          self.table[index] = [(k, v) for k, v in self.table[index] if k != key]

      ht = HashTable()
      n = int(input("Enter number of key-value pairs: "))
      for _ in range(n):
          k = input("Enter key: ")
          v = input("Enter value: ")
          ht.insert(k, v)

      search_key = input("Enter key to search: ")
      print("Result:", ht.search(search_key))

Enter number of key-value pairs: 5
Enter key: 5
Enter value: 6
Enter key: 7
Enter value: 8
Enter key: 9
Enter value: 4
Enter key: 3
Enter value: 2
Enter key: 1
Enter value: 5
Enter key to search: 7
Result: 8
```

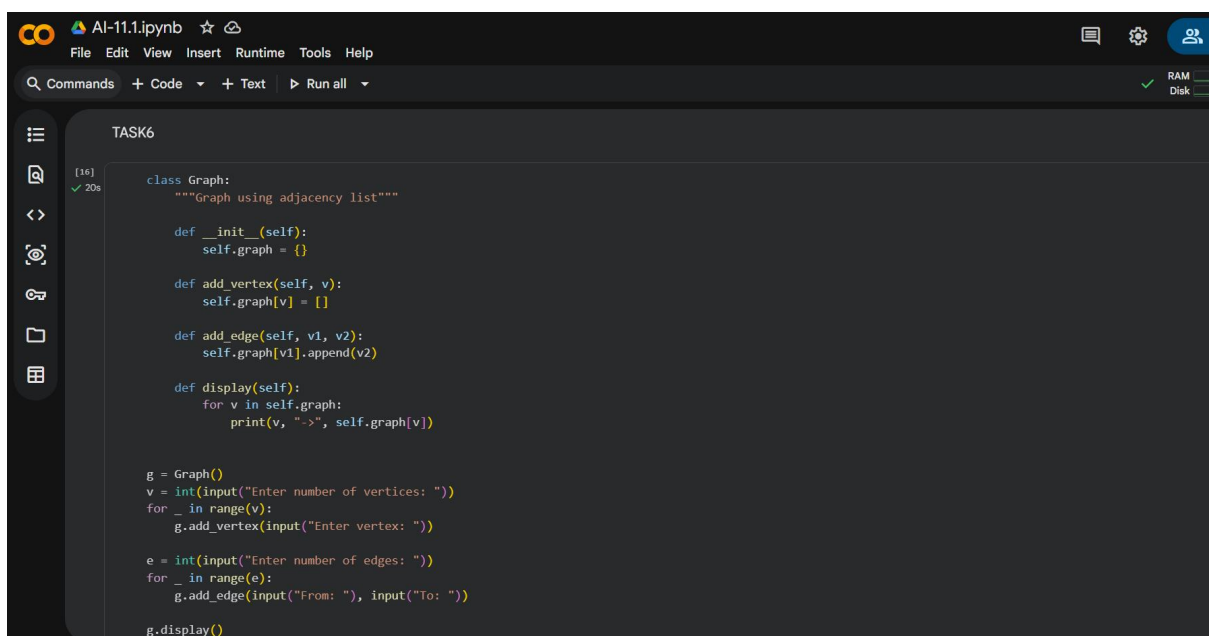
## Observation:

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.



```
[16] ✓ 20s
class Graph:
    """Graph using adjacency list"""

    def __init__(self):
        self.graph = {}

    def add_vertex(self, v):
        self.graph[v] = []

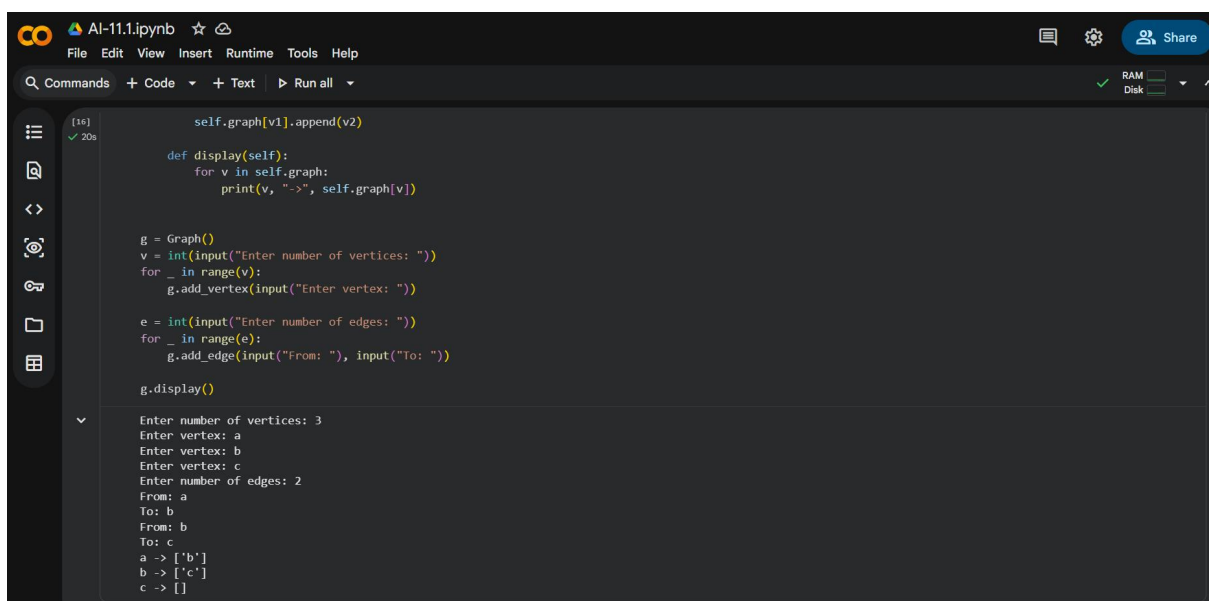
    def add_edge(self, v1, v2):
        self.graph[v1].append(v2)

    def display(self):
        for v in self.graph:
            print(v, "->", self.graph[v])

g = Graph()
v = int(input("Enter number of vertices: "))
for _ in range(v):
    g.add_vertex(input("Enter vertex: "))

e = int(input("Enter number of edges: "))
for _ in range(e):
    g.add_edge(input("From: "), input("To: "))

g.display()
```



```
[16] ✓ 20s
    self.graph[v1].append(v2)

    def display(self):
        for v in self.graph:
            print(v, "->", self.graph[v])

g = Graph()
v = int(input("Enter number of vertices: "))
for _ in range(v):
    g.add_vertex(input("Enter vertex: "))

e = int(input("Enter number of edges: "))
for _ in range(e):
    g.add_edge(input("From: "), input("To: "))

g.display()

Enter number of vertices: 3
Enter vertex: a
Enter vertex: b
Enter vertex: c
Enter number of edges: 2
From: a
To: b
From: b
To: c
a -> ['b']
b -> ['c']
c -> []
```

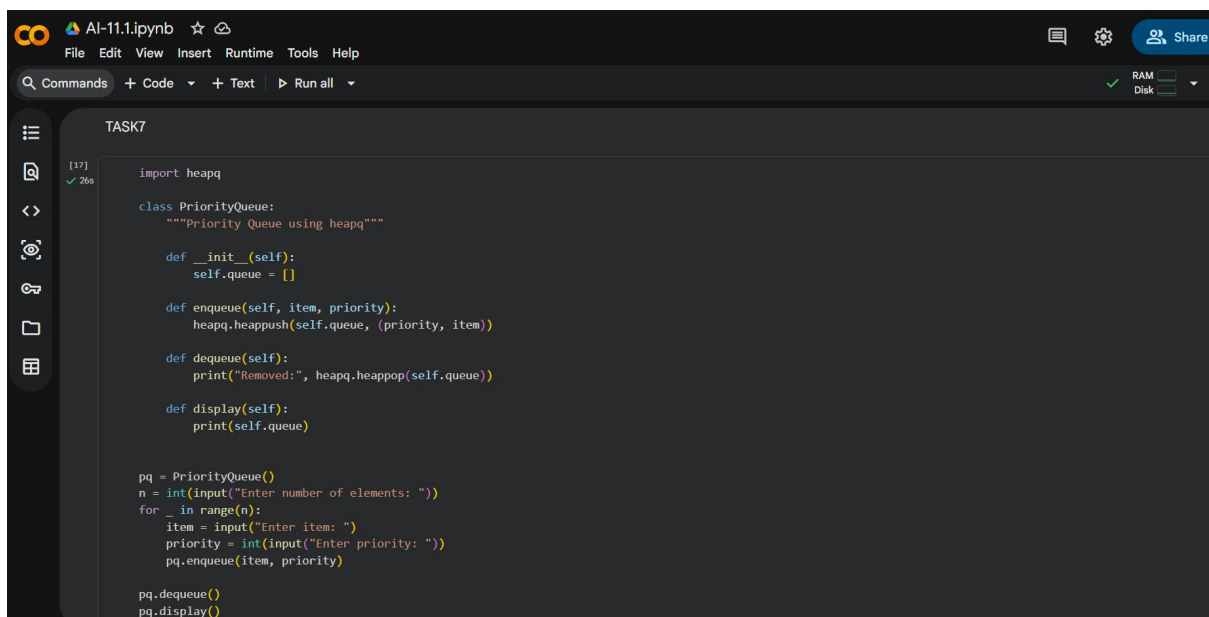
## Observation:

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.



```
import heapq

class PriorityQueue:
    """Priority Queue using heapq"""

    def __init__(self):
        self.queue = []

    def enqueue(self, item, priority):
        heapq.heappush(self.queue, (priority, item))

    def dequeue(self):
        print("Removed:", heapq.heappop(self.queue))

    def display(self):
        print(self.queue)

pq = PriorityQueue()
n = int(input("Enter number of elements: "))
for _ in range(n):
    item = input("Enter item: ")
    priority = int(input("Enter priority: "))
    pq.enqueue(item, priority)

pq.dequeue()
pq.display()
```



```
def enqueue(self, item, priority):
    heapq.heappush(self.queue, (priority, item))

def dequeue(self):
    print("Removed:", heapq.heappop(self.queue))

def display(self):
    print(self.queue)

pq = PriorityQueue()
n = int(input("Enter number of elements: "))
for _ in range(n):
    item = input("Enter item: ")
    priority = int(input("Enter priority: "))
    pq.enqueue(item, priority)

pq.dequeue()
pq.display()

Enter number of elements: 3
Enter item: task1
Enter priority: 2
Enter item: task2
Enter priority: 1
Enter item: task3
Enter priority: 3
Removed: (1, 'task2')
[(2, 'task1'), (3, 'task3')]
```

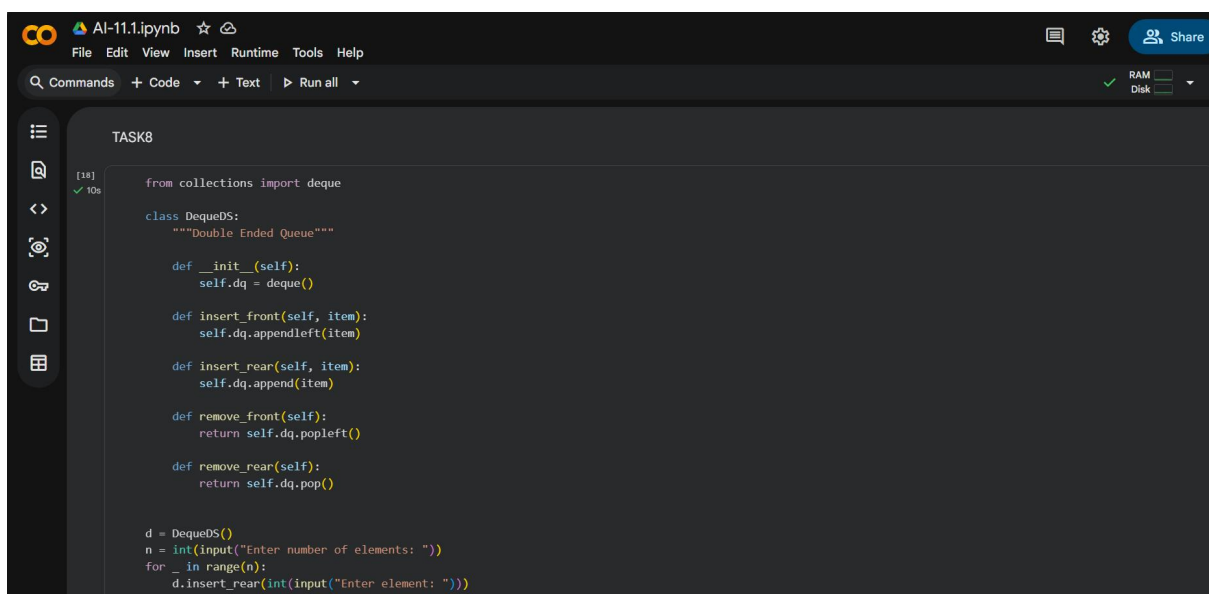
## Observation :

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.



```
[18] ✓ 10s
TASK8

from collections import deque

class DequeDS:
    """Double Ended Queue"""

    def __init__(self):
        self.dq = deque()

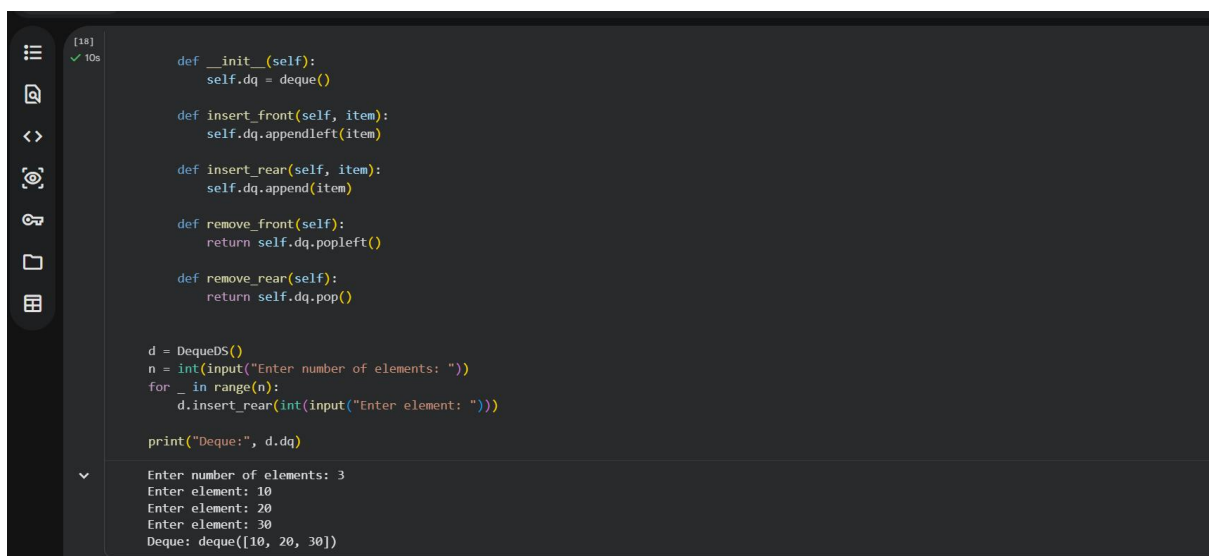
    def insert_front(self, item):
        self.dq.appendleft(item)

    def insert_rear(self, item):
        self.dq.append(item)

    def remove_front(self):
        return self.dq.popleft()

    def remove_rear(self):
        return self.dq.pop()

d = DequeDS()
n = int(input("Enter number of elements: "))
for _ in range(n):
    d.insert_rear(int(input("Enter element: ")))
```



```
[18] ✓ 10s

    def __init__(self):
        self.dq = deque()

    def insert_front(self, item):
        self.dq.appendleft(item)

    def insert_rear(self, item):
        self.dq.append(item)

    def remove_front(self):
        return self.dq.popleft()

    def remove_rear(self):
        return self.dq.pop()

d = DequeDS()
n = int(input("Enter number of elements: "))
for _ in range(n):
    d.insert_rear(int(input("Enter element: ")))

print("Deque:", d.dq)

Enter number of elements: 3
Enter element: 10
Enter element: 20
Enter element: 30
Deque: deque([10, 20, 30])
```



### Observation:

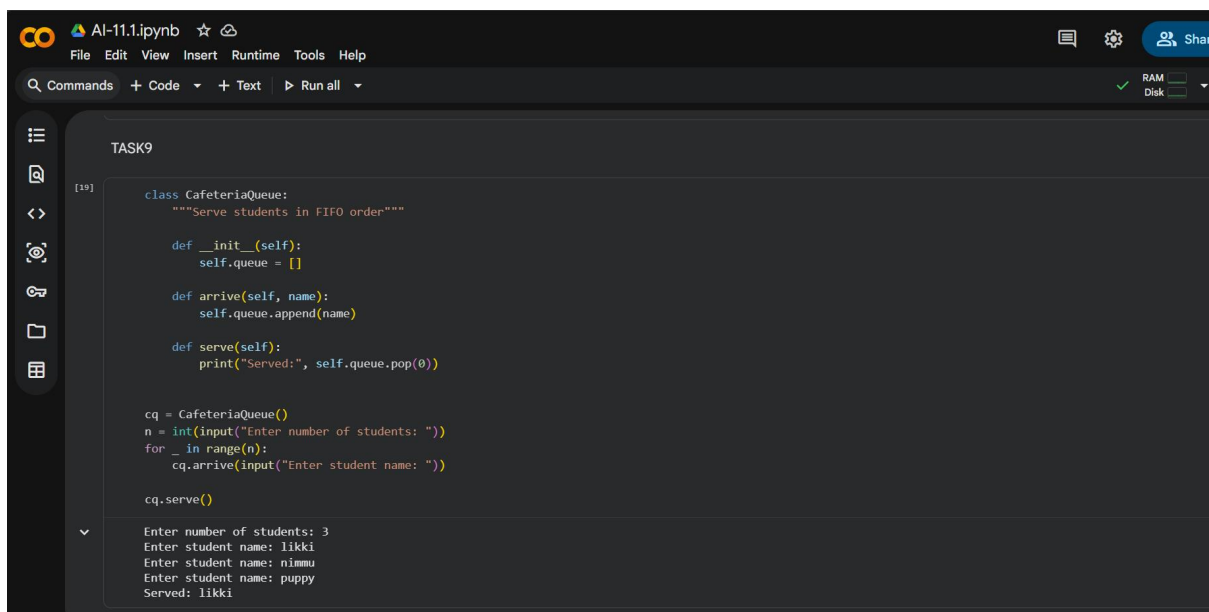
The deque implementation allows flexible insertion and deletion from both ends. Operations are performed efficiently in constant time. This structure is more efficient than list-based implementations for such operations. The deque is suitable for applications like sliding window problems. The code is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



```
class CafeteriaQueue:
    """Serve students in FIFO order"""

    def __init__(self):
        self.queue = []

    def arrive(self, name):
        self.queue.append(name)

    def serve(self):
        print("Served:", self.queue.pop(0))

cq = CafeteriaQueue()
n = int(input("Enter number of students: "))
for _ in range(n):
    cq.arrive(input("Enter student name: "))

cq.serve()

Enter number of students: 3
Enter student name: likki
Enter student name: nimmu
Enter student name: puppy
Served: likki
```

### Observation:

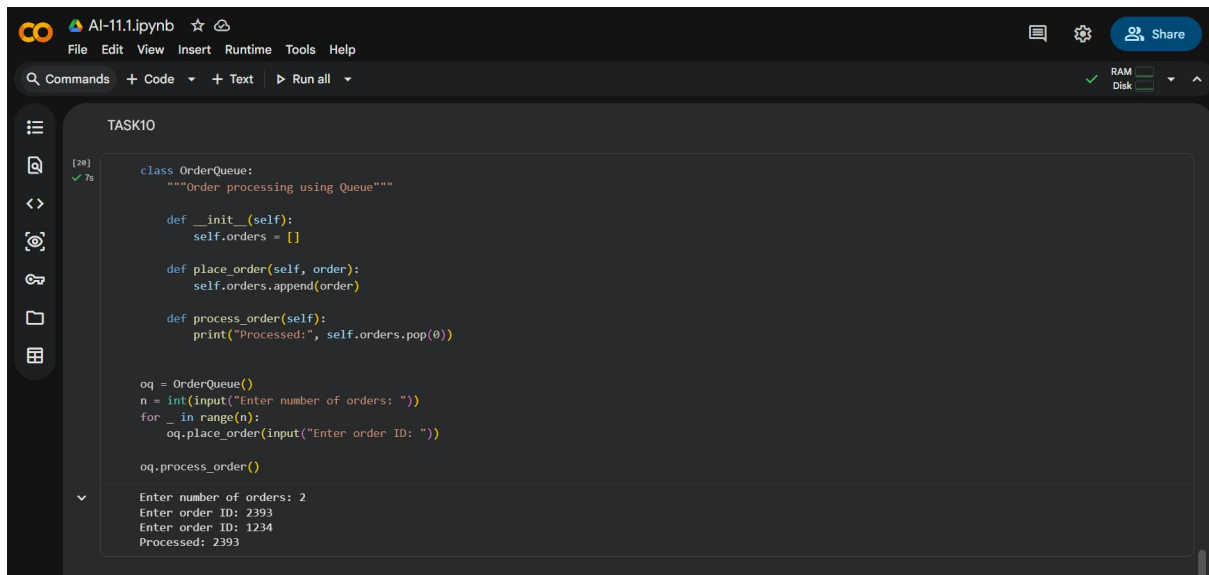
Each campus feature is mapped to an optimal data structure. Queues efficiently handle cafeteria orders, while graphs represent bus routes effectively. Hash tables enable fast searching for event registrations. The selected data structures improve performance and scalability. The implemented feature demonstrates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



The screenshot shows a Jupyter Notebook titled "AI-11.1.ipynb" with a dark theme. The notebook is in "TASK10" mode. The code defines a class `OrderQueue` with methods `__init__`, `place_order`, and `process_order`. The `__init__` method initializes `self.orders` as an empty list. The `place_order` method appends an order to the list. The `process_order` method prints the order ID and removes it from the list. The code then creates an instance of `OrderQueue`, prompts the user for the number of orders (2), and loops to prompt for order IDs (2393 and 1234). Finally, it calls `process_order` on the instance.

```
[28] ✓ 7s
class OrderQueue:
    """Order processing using Queue"""

    def __init__(self):
        self.orders = []

    def place_order(self, order):
        self.orders.append(order)

    def process_order(self):
        print("Processed:", self.orders.pop(0))

oq = OrderQueue()
n = int(input("Enter number of orders: "))
for _ in range(n):
    oq.place_order(input("Enter order ID: "))

oq.process_order()
```

Enter number of orders: 2  
Enter order ID: 2393  
Enter order ID: 1234  
Processed: 2393

### 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.