ASSIGNMENT-11.1

HT.NO:2303A510F6

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:

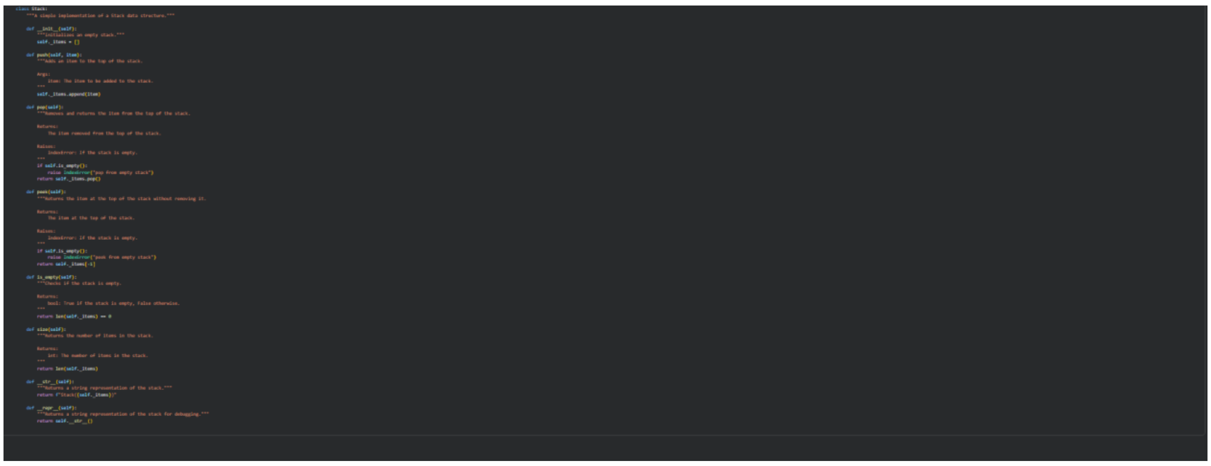
pass

Expected Output:

• A functional stack implementation with all required methods and

docstrings.

CODE:



```python
class Stack:
    """A simple implementation of a Stack data structure."""

    def __init__(self):
        """Initializes an empty stack."""
        self._items = []

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

        Args:
            item: The item to be added to the stack.
        """
        self._items.append(item)

    def pop(self):
        """Removes and returns the item from the top of the stack.

        Returns:
            The item removed from the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """Returns 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 empty stack")
        return self._items[-1]

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

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

    def size(self):
        """Returns the number of items in the stack.

        Returns:
            int: The number of items in the stack.
        """
        return len(self._items)

    def __str__(self):
        """Returns a string representation of the stack."""
        return f"Stack({self._items})"

    def __repr__(self):
        """Returns a string representation of the stack for debugging."""
        return self.__str__()
```

Task Description #2 – Queue Implementation

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

Sample Input Code:

class Queue:

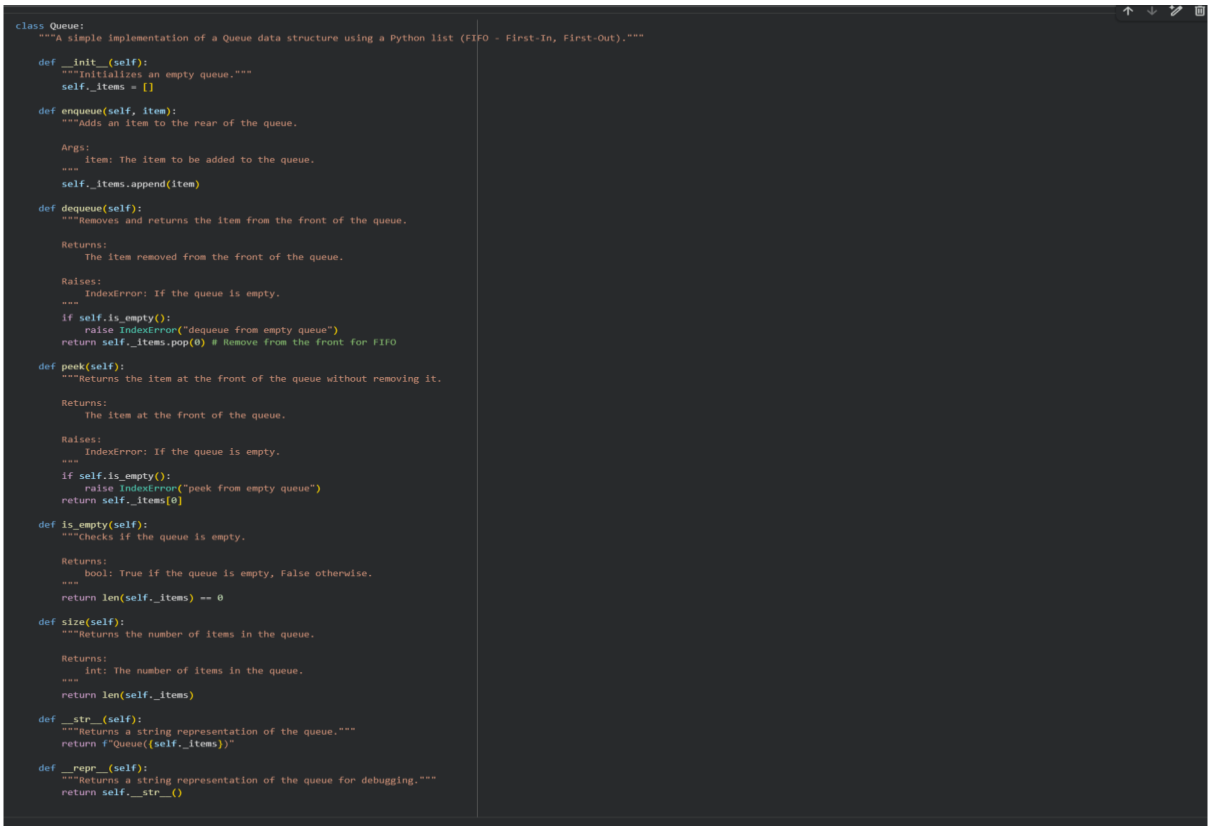
pass

Expected Output:

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

Methods

CODE:



```python
class Queue:
    """A simple implementation of a Queue data structure using a Python list (FIFO - First-In, First-Out)."""

    def __init__(self):
        """Initializes an empty queue."""
        self._items = []

    def enqueue(self, item):
        """Adds an item to the rear of the queue.

        Args:
            item: The item to be added to the queue.
        """
        self._items.append(item)

    def dequeue(self):
        """Removes and returns the item from the front of the queue.

        Returns:
            The item removed from the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.pop(0) # Remove from the front for FIFO

    def peek(self):
        """Returns the item at the front of the queue without removing it.

        Returns:
            The item at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self._items[0]

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

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

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

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

    def __str__(self):
        """Returns a string representation of the queue."""
        return f"Queue({self._items})"

    def __repr__(self):
        """Returns a string representation of the queue for debugging."""
        return self.__str__()
```

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:

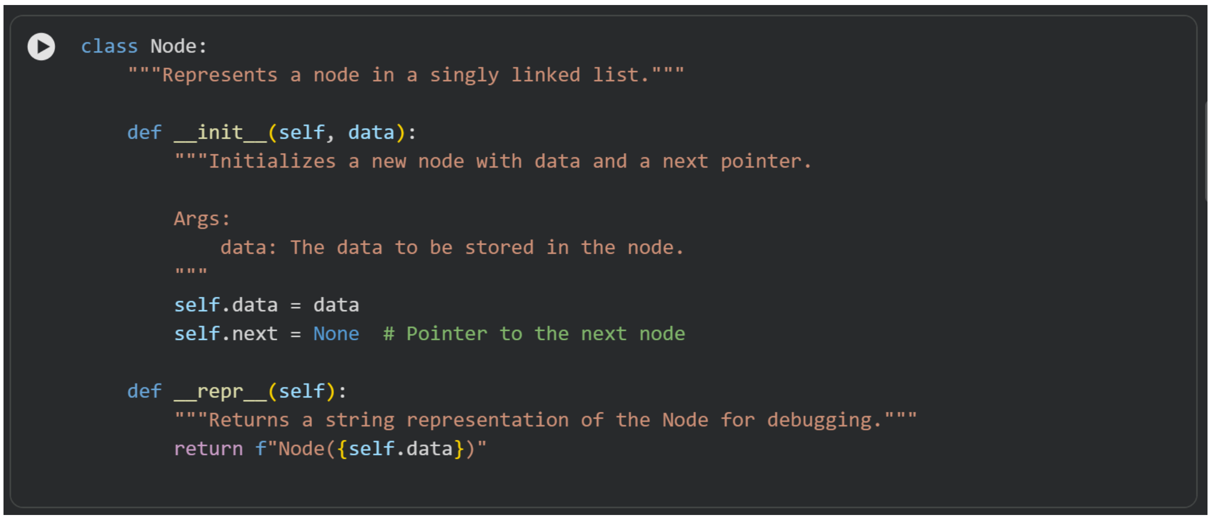
pass

Expected Output:

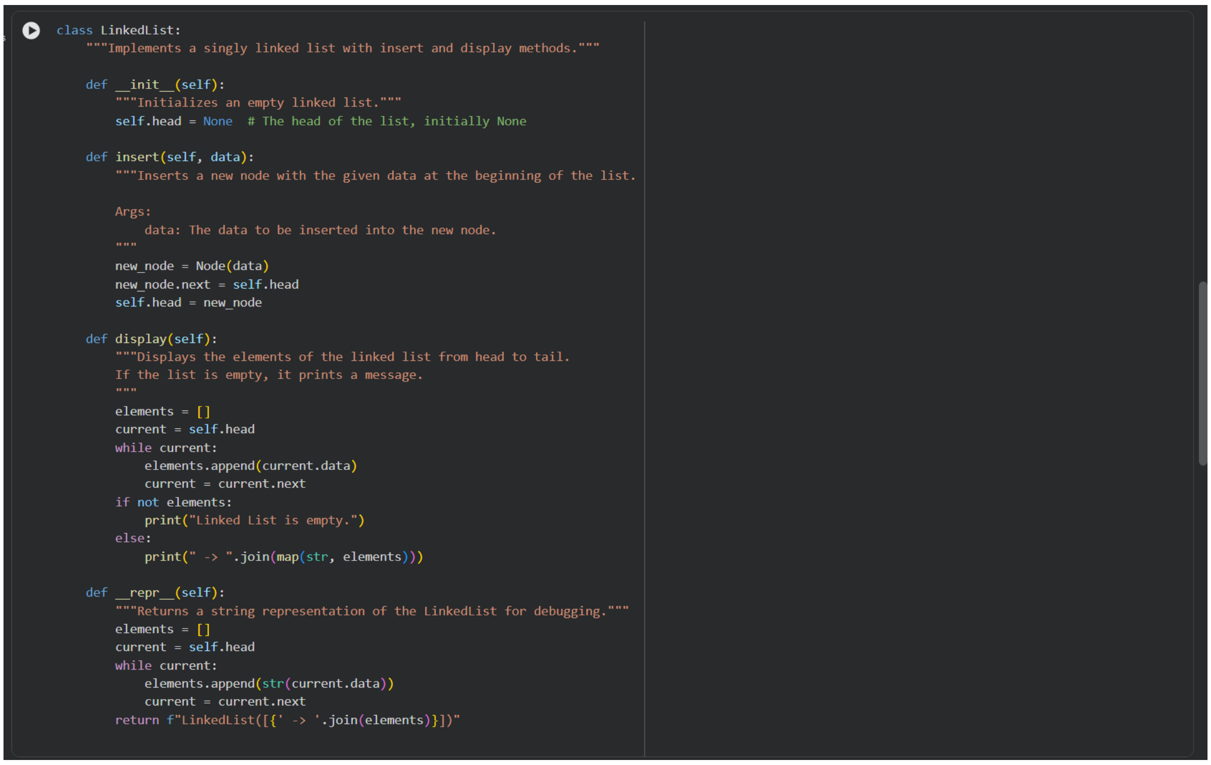
• A working linked list implementation with clear method

Documentation

CODE:

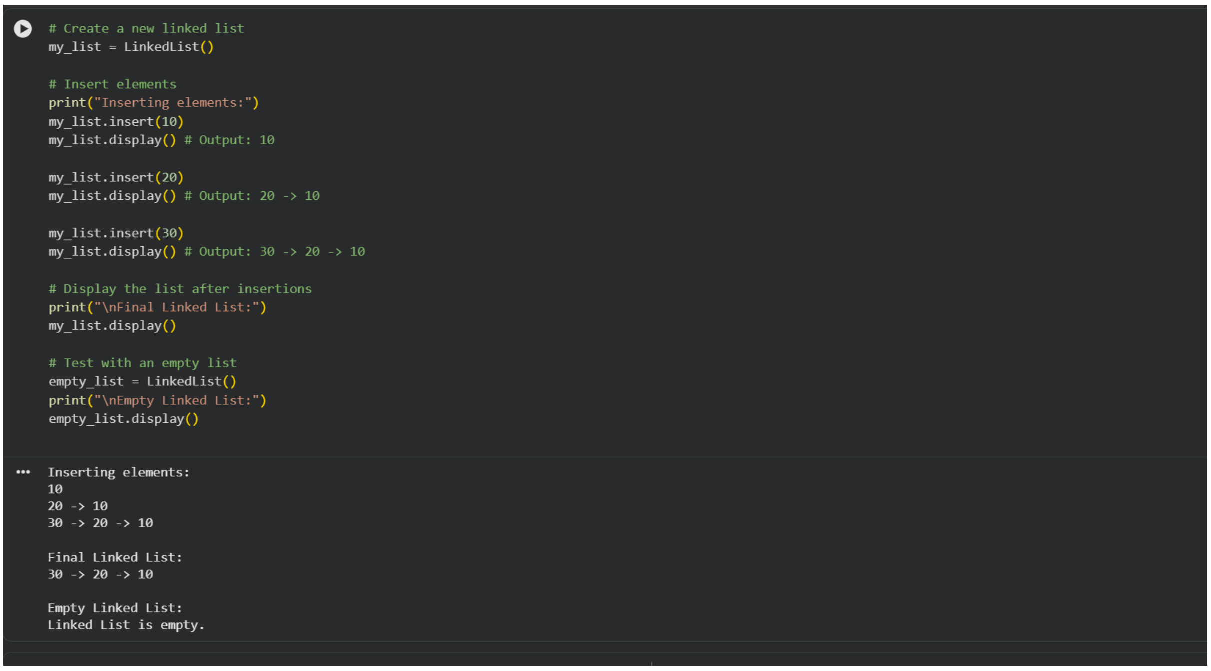


```python
class Node:
    """Represents a node in a singly linked list."""

    def __init__(self, data):
        """Initializes a new node with data and a next pointer.

        Args:
            data: The data to be stored in the node.
        """
        self.data = data
        self.next = None  # Pointer to the next node

    def __repr__(self):
        """Returns a string representation of the Node for debugging."""
        return f"Node({self.data})"
```



```python
class LinkedList:
    """Implements a singly linked list with insert and display methods."""

    def __init__(self):
        """Initializes an empty linked list."""
        self.head = None  # The head of the list, initially None

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

        Args:
            data: The data to be inserted into the new node.
        """
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def display(self):
        """Displays the elements of the linked list from head to tail.
        If the list is empty, it prints a message.
        """
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        if not elements:
            print("Linked List is empty.")
        else:
            print(" -> ".join(map(str, elements)))

    def __repr__(self):
        """Returns a string representation of the LinkedList for debugging."""
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        return f"LinkedList([{' -> '.join(elements)}])"
```

OUTPUT:



```
# Create a new linked list
my_list = LinkedList()

# Insert elements
print("Inserting elements:")
my_list.insert(10)
my_list.display() # Output: 10

my_list.insert(20)
my_list.display() # Output: 20 -> 10

my_list.insert(30)
my_list.display() # Output: 30 -> 20 -> 10

# Display the list after insertions
print("\nFinal Linked List:")
my_list.display()

# Test with an empty list
empty_list = LinkedList()
print("\nEmpty Linked List:")
empty_list.display()
```

```
Inserting elements:
10
20 -> 10
30 -> 20 -> 10

Final Linked List:
30 -> 20 -> 10

Empty Linked List:
Linked List is empty.
```

Task Description #4 – Binary Search Tree (BST)

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

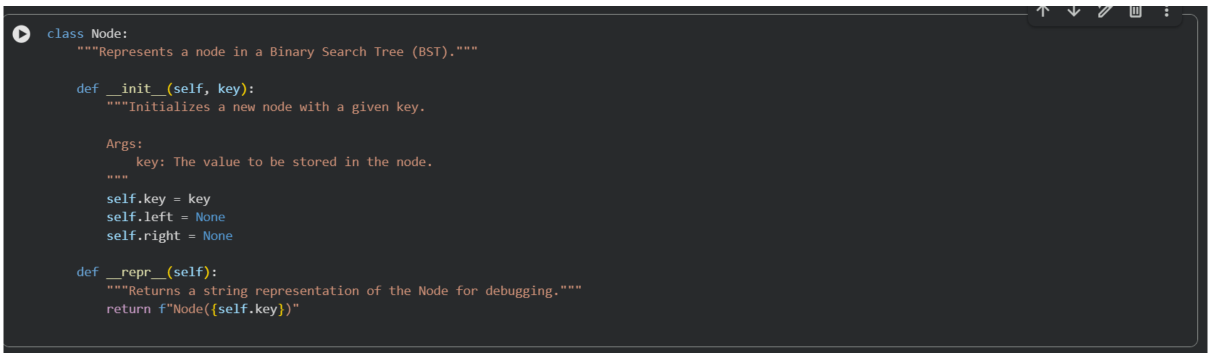
Sample Input Code:

class BST:
pass

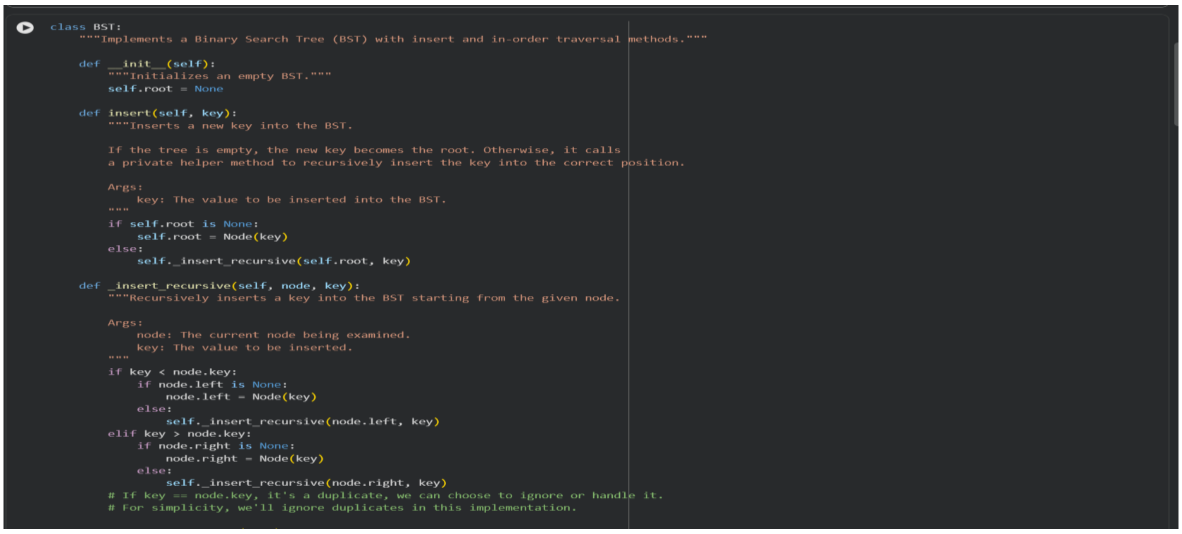
Expected Output:

• BST implementation with recursive insert and traversal methods.

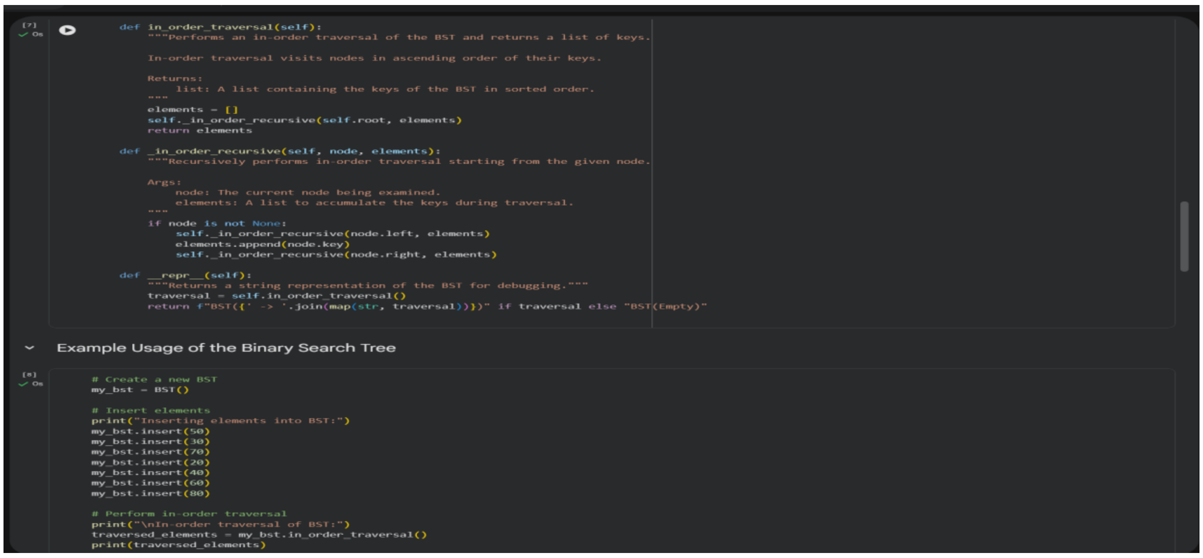
CODE:



```
class Node:
    """Represents a node in a Binary Search Tree (BST)."""

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

        Args:
            key: The value to be stored in the node.
        """
        self.key = key
        self.left = None
        self.right = None

    def __repr__(self):
        """Returns a string representation of the Node for debugging."""
        return f"Node({self.key})"
```



```python
class BST:
    """Implements a Binary Search Tree (BST) with insert and in-order traversal methods."""

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

    def insert(self, key):
        """Inserts a new key into the BST.

        If the tree is empty, the new key becomes the root. Otherwise, it calls
        a private helper method to recursively insert the key into the correct position.

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

    def _insert_recursive(self, node, key):
        """Recursively inserts a key into the BST starting from the given node.

        Args:
            node: The current node being examined.
            key: The value to be inserted.
        """
        if key < node.key:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert_recursive(node.left, key)
        elif key > node.key:
            if node.right is None:
                node.right = Node(key)
            else:
                self._insert_recursive(node.right, key)
        # If key == node.key, it's a duplicate, we can choose to ignore or handle it.
        # For simplicity, we'll ignore duplicates in this implementation.
```

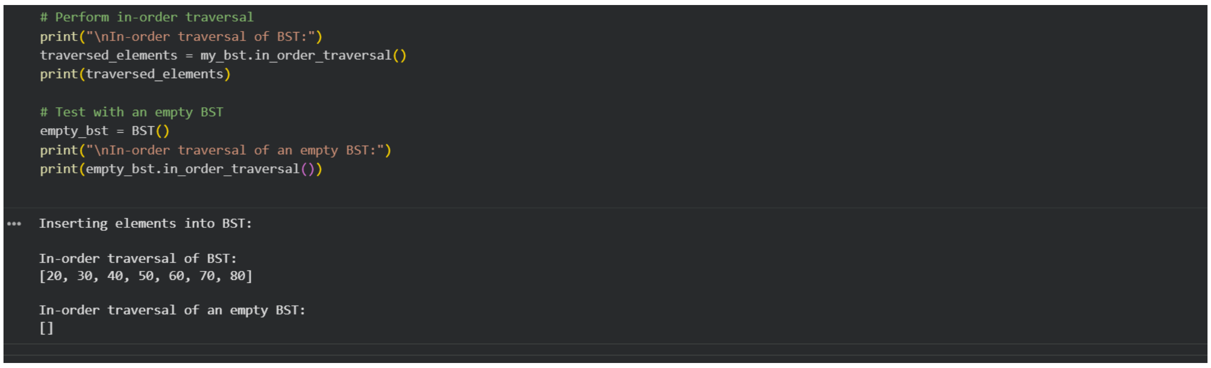


```python
[7]
    def in_order_traversal(self):
        """Performs an in-order traversal of the BST and returns a list of keys.

        In-order traversal visits nodes in ascending order of their keys.

        Returns:
            list: A list containing the keys of the BST in sorted order.
        """
        elements = []
        self._in_order_recursive(self.root, elements)
        return elements

    def _in_order_recursive(self, node, elements):
        """Recursively performs in-order traversal starting from the given node.

        Args:
            node: The current node being examined.
            elements: A list to accumulate the keys during traversal.
        """
        if node is not None:
            self._in_order_recursive(node.left, elements)
            elements.append(node.key)
            self._in_order_recursive(node.right, elements)

    def __repr__(self):
        """Returns a string representation of the BST for debugging."""
        traversal = self.in_order_traversal()
        return f"BST({' -> '.join(map(str, traversal))})" if traversal else "BST(Empty)"
```

Example Usage of the Binary Search Tree

```python
[8]
# Create a new BST
my_bst = BST()

# Insert elements
print("Inserting elements into BST:")
my_bst.insert(50)
my_bst.insert(30)
my_bst.insert(70)
my_bst.insert(20)
my_bst.insert(40)
my_bst.insert(60)
my_bst.insert(80)

# Perform in-order traversal
print("\nIn-order traversal of BST:")
traversed_elements = my_bst.in_order_traversal()
print(traversed_elements)
```

OUTPUT:



```python
# Perform in-order traversal
print("\nIn-order traversal of BST:")
traversed_elements = my_bst.in_order_traversal()
print(traversed_elements)

# Test with an empty BST
empty_bst = BST()
print("\nIn-order traversal of an empty BST:")
print(empty_bst.in_order_traversal())
```

```
Inserting elements into BST:

In-order traversal of BST:
[20, 30, 40, 50, 60, 70, 80]

In-order traversal of an empty BST:
[]
```

Task Description #5 – Hash Table

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

delete methods.

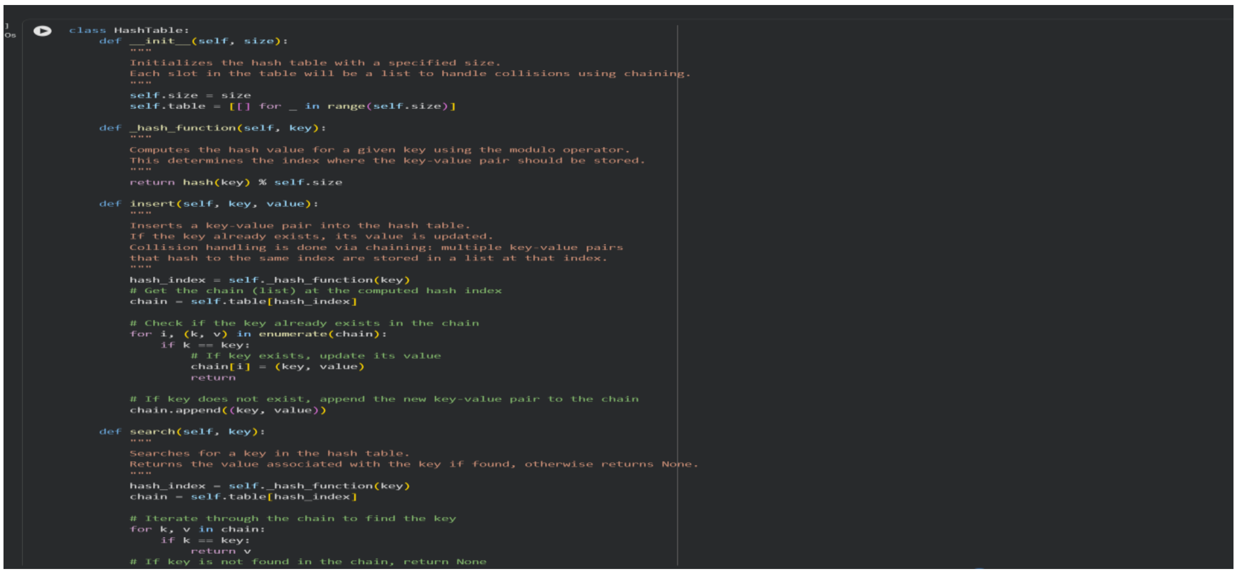
Sample Input Code:

class HashTable:

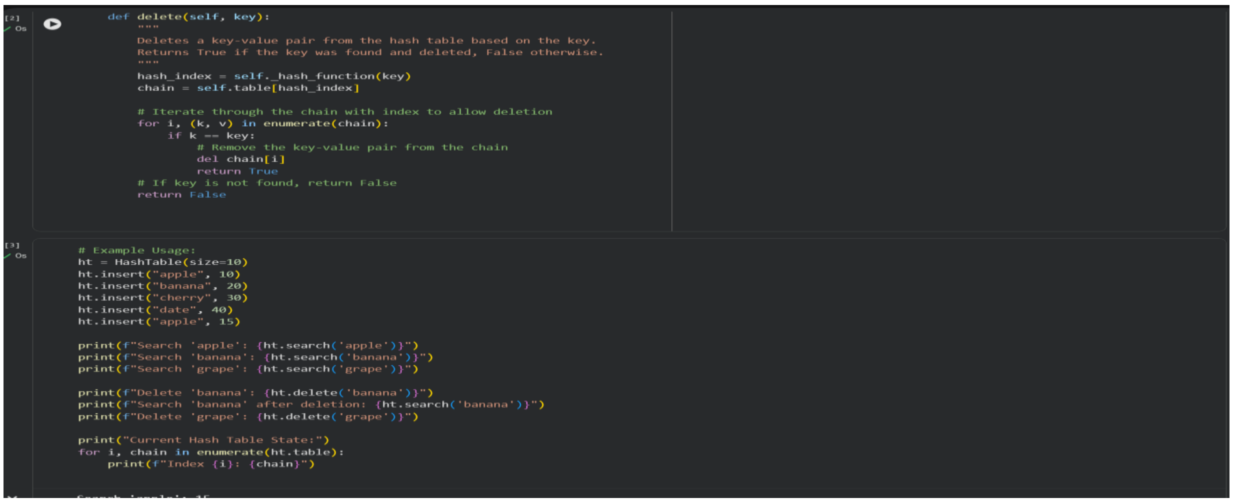
pass

Expected Output:

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



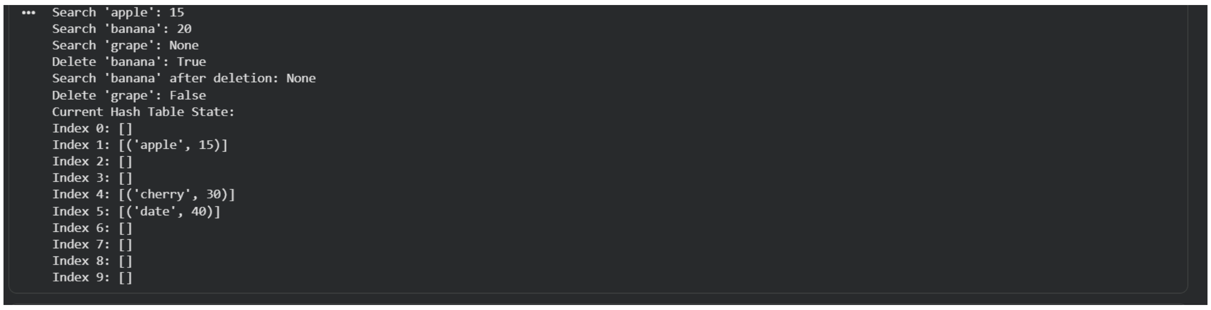
```python
class HashTable:
    def __init__(self, size):
        """
        Initializes the hash table with a specified size.
        Each slot in the table will be a list to handle collisions using chaining.
        """
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def _hash_function(self, key):
        """
        Computes the hash value for a given key using the modulo operator.
        This determines the index where the key-value pair should be stored.
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """
        Inserts a key-value pair into the hash table.
        If the key already exists, its value is updated.
        Collision handling is done via chaining: multiple key-value pairs
        that hash to the same index are stored in a list at that index.
        """
        hash_index = self._hash_function(key)
        # Get the chain (list) at the computed hash index
        chain = self.table[hash_index]

        # Check if the key already exists in the chain
        for i, (k, v) in enumerate(chain):
            if k == key:
                # If key exists, update its value
                chain[i] = (key, value)
                return

        # If key does not exist, append the new key-value pair to the chain
        chain.append((key, value))

    def search(self, key):
        """
        Searches for a key in the hash table.
        Returns the value associated with the key if found, otherwise returns None.
        """
        hash_index = self._hash_function(key)
        chain = self.table[hash_index]

        # Iterate through the chain to find the key
        for k, v in chain:
            if k == key:
                return v
        # If key is not found in the chain, return None
```



```python
    def delete(self, key):
        """
        Deletes a key-value pair from the hash table based on the key.
        Returns True if the key was found and deleted, False otherwise.
        """
        hash_index = self._hash_function(key)
        chain = self.table[hash_index]

        # Iterate through the chain with index to allow deletion
        for i, (k, v) in enumerate(chain):
            if k == key:
                # Remove the key-value pair from the chain
                del chain[i]
                return True
        # If key is not found, return False
        return False
```

```python
# Example Usage:
ht = HashTable(size=10)
ht.insert("apple", 10)
ht.insert("banana", 20)
ht.insert("cherry", 30)
ht.insert("date", 40)
ht.insert("apple", 15)

print(f"Search 'apple': {ht.search('apple')}")
print(f"Search 'banana': {ht.search('banana')}")
print(f"Search 'grape': {ht.search('grape')}")

print(f"Delete 'banana': {ht.delete('banana')}")
print(f"Search 'banana' after deletion: {ht.search('banana')}")
print(f"Delete 'grape': {ht.delete('grape')}")

print("Current Hash Table State:")
for i, chain in enumerate(ht.table):
    print(f"Index {i}: {chain}")
```

OUTPUT:



```
Search 'apple': 15
Search 'banana': 20
Search 'grape': None
Delete 'banana': True
Search 'banana' after deletion: None
Delete 'grape': False
Current Hash Table State:
Index 0: []
Index 1: [('apple', 15)]
Index 2: []
Index 3: []
Index 4: [('cherry', 30)]
Index 5: [('date', 40)]
Index 6: []
Index 7: []
Index 8: []
Index 9: []
```

Task Description #6 – Graph Representation

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

Sample Input Code:

class Graph:

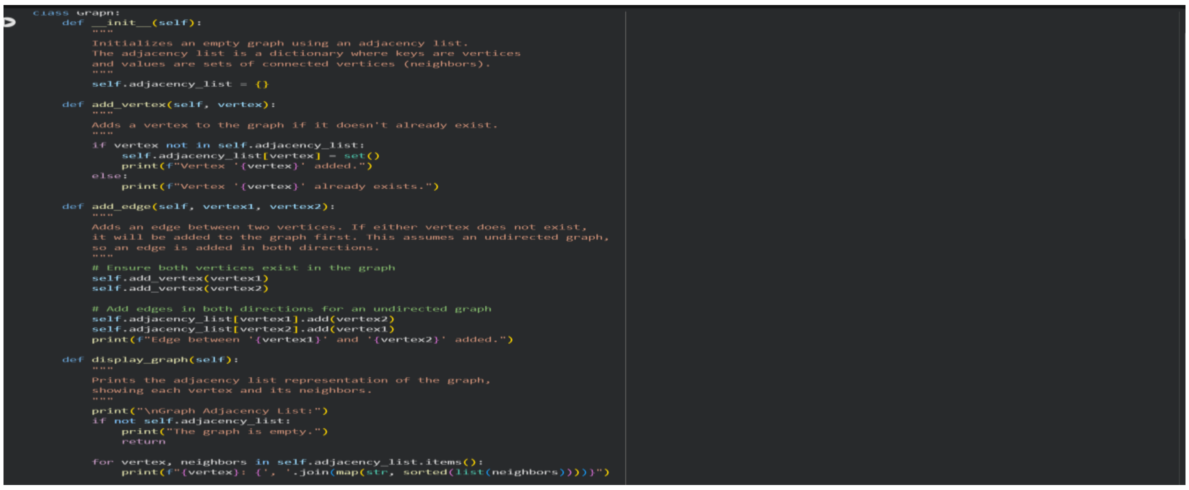
pass

Expected Output:

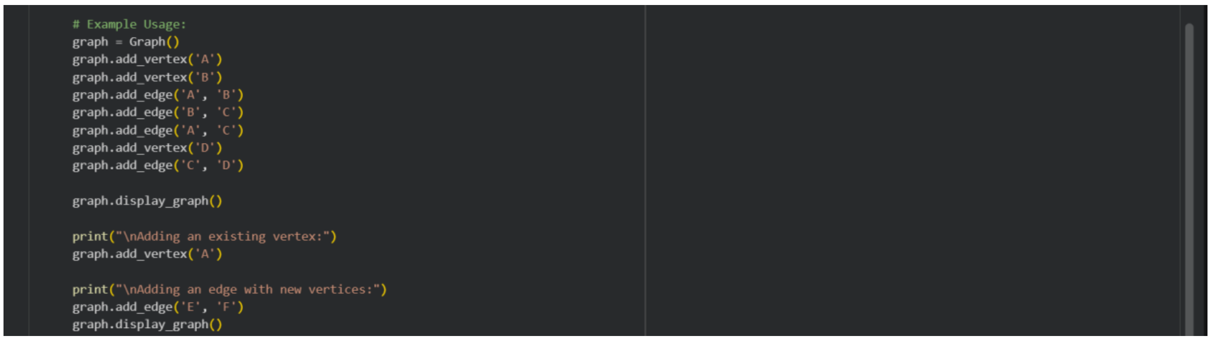
• Graph with methods to add vertices, add edges, and display

Connections

CODE:

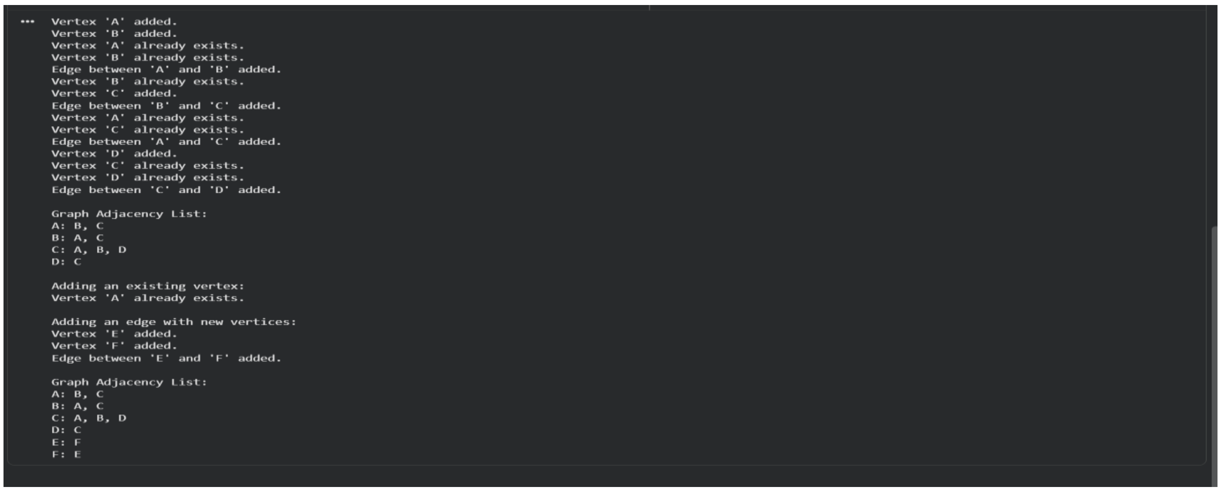


```
class Graph:
    def __init__(self):
        """
        Initializes an empty graph using an adjacency list.
        The adjacency list is a dictionary where keys are vertices
        and values are sets of connected vertices (neighbors).
        """
        self.adjacency_list = {}

    def add_vertex(self, vertex):
        """
        Adds a vertex to the graph if it doesn't already exist.
        """
        if vertex not in self.adjacency_list:
            self.adjacency_list[vertex] = set()
            print(f"Vertex '{vertex}' added.")
        else:
            print(f"Vertex '{vertex}' already exists.")

    def add_edge(self, vertex1, vertex2):
        """
        Adds an edge between two vertices. If either vertex does not exist,
        it will be added to the graph first. This assumes an undirected graph,
        so an edge is added in both directions.
        """
        # Ensure both vertices exist in the graph
        self.add_vertex(vertex1)
        self.add_vertex(vertex2)

        # Add edges in both directions for an undirected graph
        self.adjacency_list[vertex1].add(vertex2)
        self.adjacency_list[vertex2].add(vertex1)
        print(f"Edge between '{vertex1}' and '{vertex2}' added.")

    def display_graph(self):
        """
        Prints the adjacency list representation of the graph,
        showing each vertex and its neighbors.
        """
        print("\nGraph Adjacency List:")
        if not self.adjacency_list:
            print("The graph is empty.")
            return

        for vertex, neighbors in self.adjacency_list.items():
            print(f"{vertex}: {', '.join(map(str, sorted(list(neighbors))))}")
```



```
# Example Usage:
graph = Graph()
graph.add_vertex('A')
graph.add_vertex('B')
graph.add_edge('A', 'B')
graph.add_edge('B', 'C')
graph.add_edge('A', 'C')
graph.add_vertex('D')
graph.add_edge('C', 'D')

graph.display_graph()

print("\nAdding an existing vertex:")
graph.add_vertex('A')

print("\nAdding an edge with new vertices:")
graph.add_edge('E', 'F')
graph.display_graph()
```

OUTPUT:



```
Vertex 'A' added.
Vertex 'B' added.
Vertex 'A' already exists.
Vertex 'B' already exists.
Edge between 'A' and 'B' added.
Vertex 'B' already exists.
Vertex 'C' added.
Edge between 'B' and 'C' added.
Vertex 'A' already exists.
Vertex 'C' already exists.
Edge between 'A' and 'C' added.
Vertex 'D' added.
Vertex 'C' already exists.
Vertex 'D' already exists.
Edge between 'C' and 'D' added.

Graph Adjacency List:
A: B, C
B: A, C
C: A, B, D
D: C

Adding an existing vertex:
Vertex 'A' already exists.

Adding an edge with new vertices:
Vertex 'E' added.
Vertex 'F' added.
Edge between 'E' and 'F' added.

Graph Adjacency List:
A: B, C
B: A, C
C: A, B, D
D: C
E: F
F: E
```

Task Description #7 – Priority Queue

Task: Use AI to implement a priority queue using Python’s heapq

module.

Sample Input Code:

class PriorityQueue:

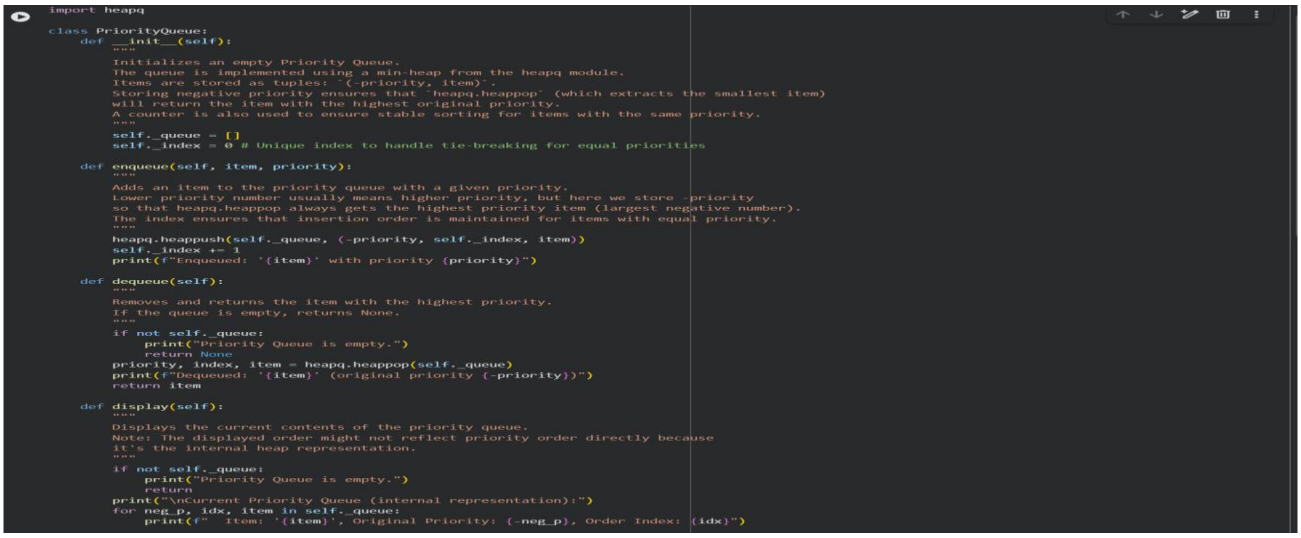
pass

Expected Output:

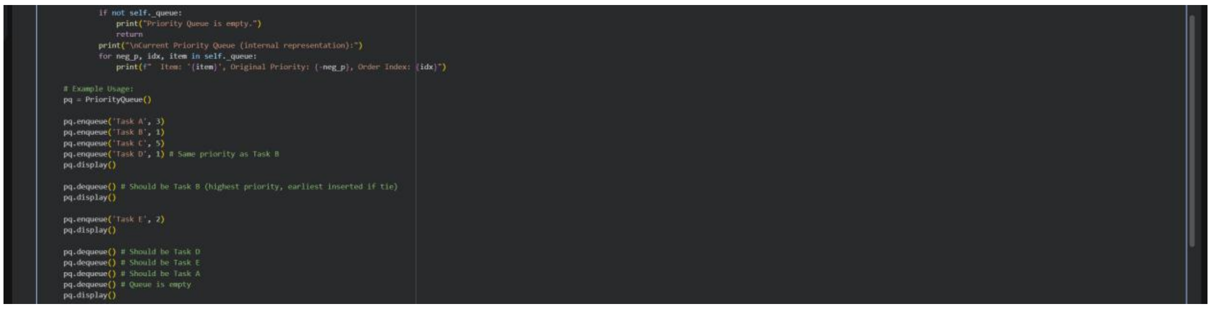
• Implementation with enqueue (priority), dequeue (highest priority),

and display methods.

CODE:

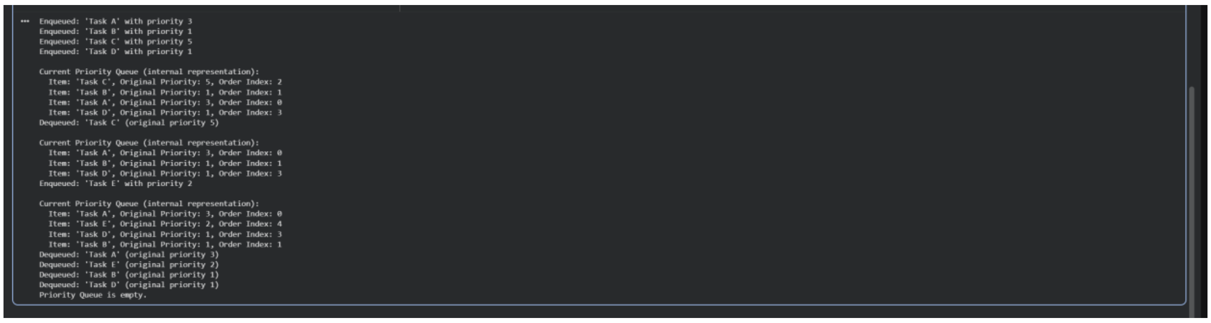


```python
import heapq

class PriorityQueue:
    def __init__(self):
        """
        Initializes an empty Priority Queue.
        The queue is implemented using a min-heap from the heapq module.
        Items are stored as tuples: `(-priority, item)`.
        Storing negative priority ensures that `heapq.heappop` (which extracts the smallest item)
        will return the item with the highest original priority.
        A counter is also used to ensure stable sorting for items with the same priority.
        """
        self._queue = []
        self._index = 0 # Unique index to handle tie-breaking for equal priorities

    def enqueue(self, item, priority):
        """
        Adds an item to the priority queue with a given priority.
        Lower priority number usually means higher priority, but here we store -priority
        so that heapq.heappop always gets the highest priority item (largest negative number).
        The index ensures that insertion order is maintained for items with equal priority.
        """
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1
        print(f"Enqueued: '{item}' with priority {priority}")

    def dequeue(self):
        """
        Removes and returns the item with the highest priority.
        If the queue is empty, returns None.
        """
        if not self._queue:
            print("Priority Queue is empty.")
            return None
        priority, index, item = heapq.heappop(self._queue)
        print(f"Dequeued: '{item}' (original priority {-priority})")
        return item

    def display(self):
        """
        Displays the current contents of the priority queue.
        Note: The displayed order might not reflect priority order directly because
        it's the internal heap representation.
        """
        if not self._queue:
            print("Priority Queue is empty.")
            return
        print("\nCurrent Priority Queue (internal representation):")
        for neg_p, idx, item in self._queue:
            print(f"  Item: '{item}', Original Priority: {-neg_p}, Order Index: {idx}")
```



```python
        if not self._queue:
            print("Priority Queue is empty.")
            return
        print("\nCurrent Priority Queue (internal representation):")
        for neg_p, idx, item in self._queue:
            print(f"  Item: '{item}', Original Priority: {-neg_p}, Order Index: {idx}")

# Example Usage:
pq = PriorityQueue()

pq.enqueue('Task A', 3)
pq.enqueue('Task B', 1)
pq.enqueue('Task C', 5)
pq.enqueue('Task D', 1) # Same priority as Task B
pq.display()

pq.dequeue() # Should be Task B (highest priority, earliest inserted if tie)
pq.display()

pq.enqueue('Task E', 2)
pq.display()

pq.dequeue() # Should be Task D
pq.dequeue() # Should be Task E
pq.dequeue() # Should be Task A
pq.dequeue() # Queue is empty
pq.display()
```

OUTPUT:



```
Enqueued: 'Task A' with priority 3
Enqueued: 'Task B' with priority 1
Enqueued: 'Task C' with priority 5
Enqueued: 'Task D' with priority 1

Current Priority Queue (internal representation):
  Item: 'Task C', Original Priority: 5, Order Index: 2
  Item: 'Task B', Original Priority: 1, Order Index: 1
  Item: 'Task A', Original Priority: 3, Order Index: 0
  Item: 'Task D', Original Priority: 1, Order Index: 3
Dequeued: 'Task C' (original priority 5)

Current Priority Queue (internal representation):
  Item: 'Task A', Original Priority: 3, Order Index: 0
  Item: 'Task B', Original Priority: 1, Order Index: 1
  Item: 'Task D', Original Priority: 1, Order Index: 3
Enqueued: 'Task E' with priority 2

Current Priority Queue (internal representation):
  Item: 'Task A', Original Priority: 3, Order Index: 0
  Item: 'Task E', Original Priority: 2, Order Index: 4
  Item: 'Task D', Original Priority: 1, Order Index: 3
  Item: 'Task B', Original Priority: 1, Order Index: 1
Dequeued: 'Task A' (original priority 3)
Dequeued: 'Task E' (original priority 2)
Dequeued: 'Task B' (original priority 1)
Dequeued: 'Task D' (original priority 1)
Priority Queue is empty.
```

Task Description #8 – Deque

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

collections.deque.

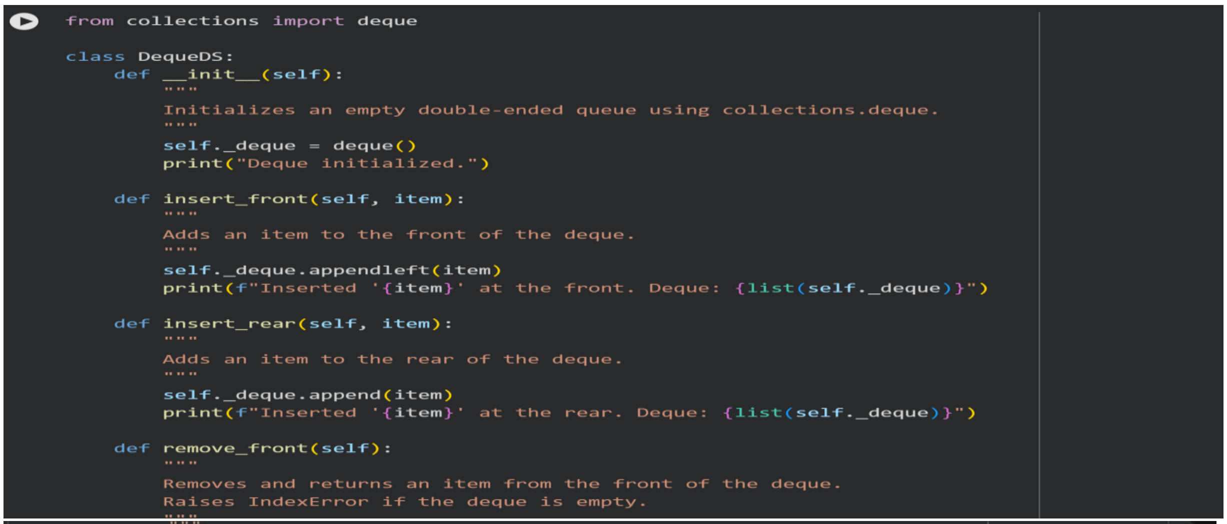
Sample Input Code:

class DequeDS:

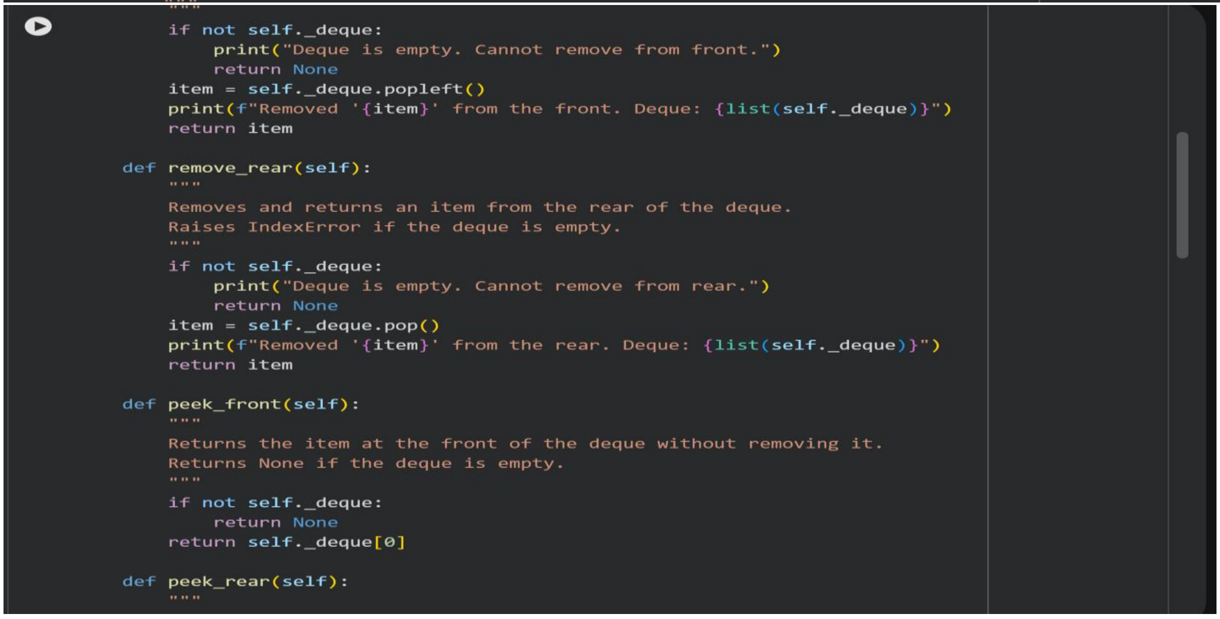
pass

Expected Output:

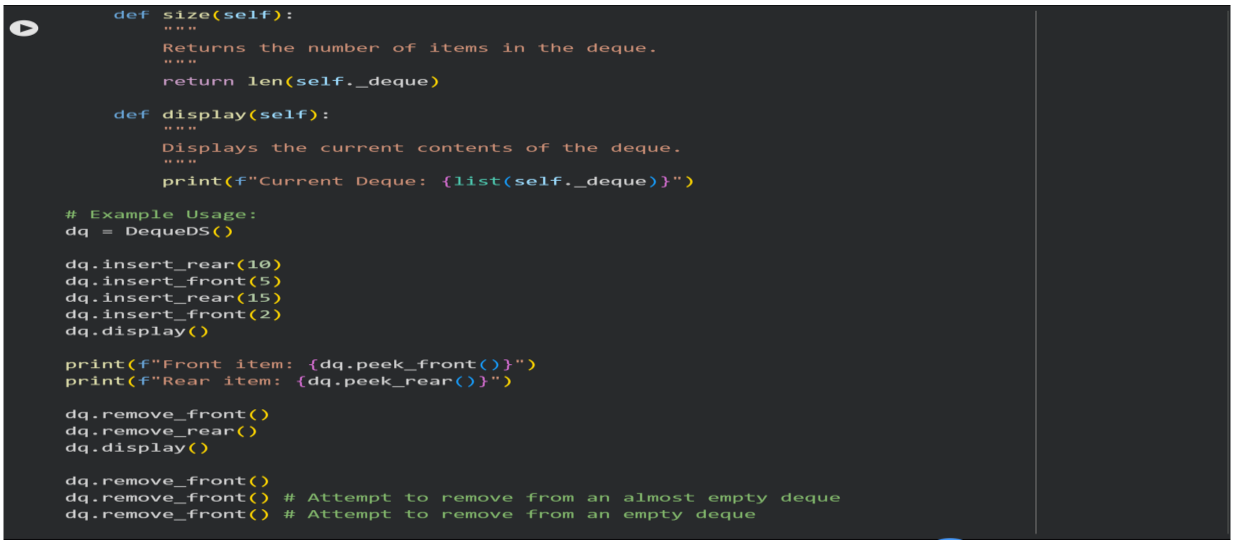
• Insert and remove from both ends with docstrings.
CODE:



```python
from collections import deque

class DequeDS:
    def __init__(self):
        """
        Initializes an empty double-ended queue using collections.deque.
        """
        self._deque = deque()
        print("Deque initialized.")

    def insert_front(self, item):
        """
        Adds an item to the front of the deque.
        """
        self._deque.appendleft(item)
        print(f"Inserted '{item}' at the front. Deque: {list(self._deque)}")

    def insert_rear(self, item):
        """
        Adds an item to the rear of the deque.
        """
        self._deque.append(item)
        print(f"Inserted '{item}' at the rear. Deque: {list(self._deque)}")

    def remove_front(self):
        """
        Removes and returns an item from the front of the deque.
        Raises IndexError if the deque is empty.
        """
```

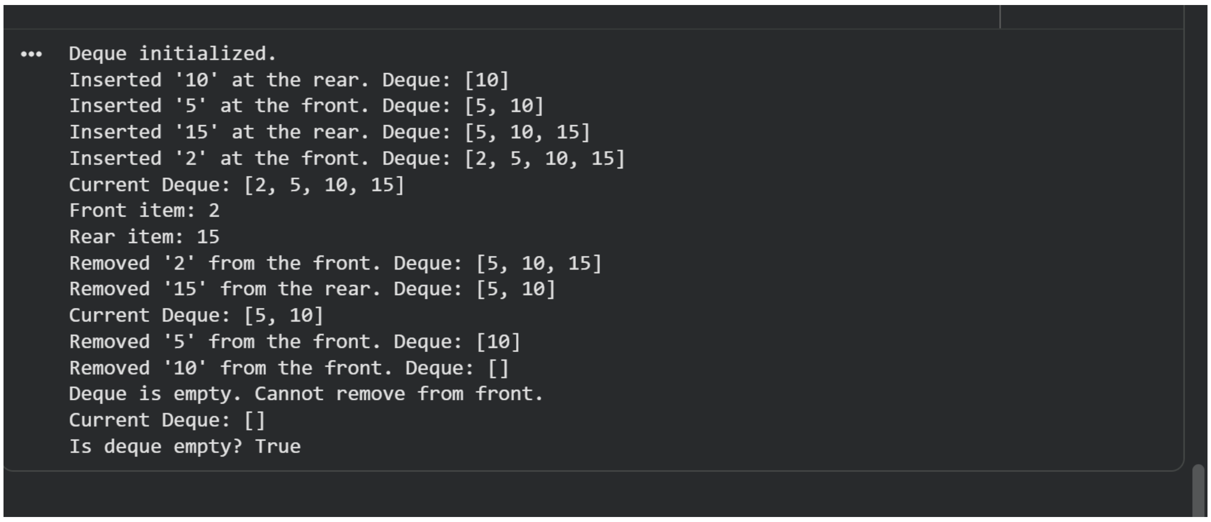


```python
        if not self._deque:
            print("Deque is empty. Cannot remove from front.")
            return None
        item = self._deque.popleft()
        print(f"Removed '{item}' from the front. Deque: {list(self._deque)}")
        return item

    def remove_rear(self):
        """
        Removes and returns an item from the rear of the deque.
        Raises IndexError if the deque is empty.
        """
        if not self._deque:
            print("Deque is empty. Cannot remove from rear.")
            return None
        item = self._deque.pop()
        print(f"Removed '{item}' from the rear. Deque: {list(self._deque)}")
        return item

    def peek_front(self):
        """
        Returns the item at the front of the deque without removing it.
        Returns None if the deque is empty.
        """
        if not self._deque:
            return None
        return self._deque[0]

    def peek_rear(self):
        """
```



```python
    def size(self):
        """
        Returns the number of items in the deque.
        """
        return len(self._deque)

    def display(self):
        """
        Displays the current contents of the deque.
        """
        print(f"Current Deque: {list(self._deque)}")

# Example Usage:
dq = DequeDS()

dq.insert_rear(10)
dq.insert_front(5)
dq.insert_rear(15)
dq.insert_front(2)
dq.display()

print(f"Front item: {dq.peek_front()}")
print(f"Rear item: {dq.peek_rear()}")

dq.remove_front()
dq.remove_rear()
dq.display()

dq.remove_front()
dq.remove_front() # Attempt to remove from an almost empty deque
dq.remove_front() # Attempt to remove from an empty deque
```

OUTPUT:



```
•••  Deque initialized.
     Inserted '10' at the rear. Deque: [10]
     Inserted '5' at the front. Deque: [5, 10]
     Inserted '15' at the rear. Deque: [5, 10, 15]
     Inserted '2' at the front. Deque: [2, 5, 10, 15]
     Current Deque: [2, 5, 10, 15]
     Front item: 2
     Rear item: 15
     Removed '2' from the front. Deque: [5, 10, 15]
     Removed '15' from the rear. Deque: [5, 10]
     Current Deque: [5, 10]
     Removed '5' from the front. Deque: [10]
     Removed '10' from the front. Deque: []
     Deque is empty. Cannot remove from front.
     Current Deque: []
     Is deque empty? True
```

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

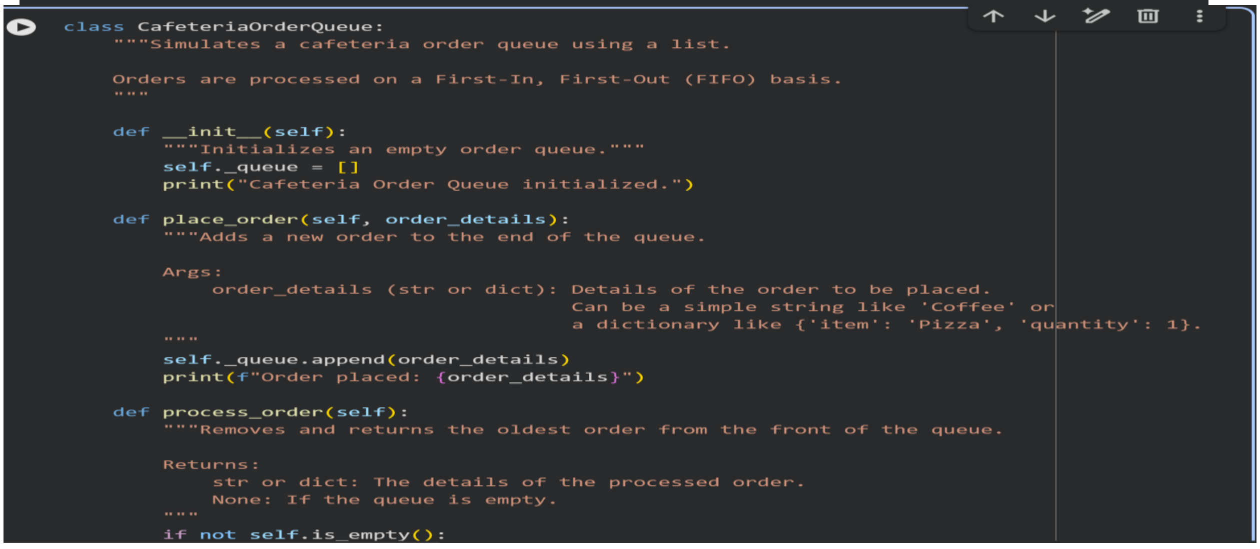
CODE:

Data Structure Table

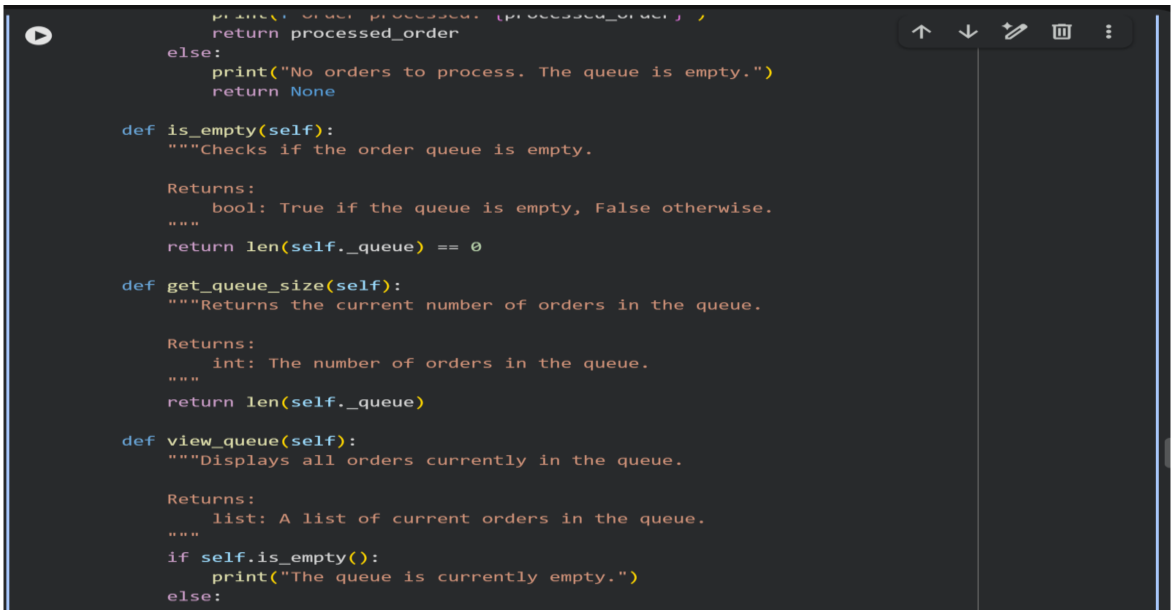
Feature	**Chosen Data Structure**	**Justification**
Student Attendance Tracking	Hash Table	A Hash Table is ideal for student attendance tracking because it allows for very efficient O(1) average time complexity for adding, removing, and looking up student records by ID. This ensures quick access to individual student attendance data, which is crucial for real-time verification.
Event Registration System	Hash Table (for registrants) and Queue (for waiting list)	A Hash Table can efficiently manage registered participants, offering quick lookups and modifications by registrant ID. For managing a waiting list or cancellations where order of arrival matters, a Queue would be suitable to process requests on a first-come, first-served basis.
Library Book	Hash Table	A Hash Table is well-suited for tracking books and borrowers, allowing for fast searches by ISBN or user ID. This enables quick

Feature	Chosen Data Structure	Justification
Borrowing		retrieval of book status (available/borrowed) and borrower information, essential for a dynamic library system.
Bus Scheduling System	Graph	A Graph is the most appropriate data structure for a bus scheduling system as it can represent routes as edges and stops as nodes. This structure inherently supports complex operations like finding optimal paths, calculating travel times, and identifying connections between stops.
Cafeteria Order Queue	Queue	A Queue perfectly models a first-come, first-served (FIFO) order processing system, ensuring that orders are handled in the exact sequence they are received. This guarantees fairness and accurate processing of cafeteria orders.

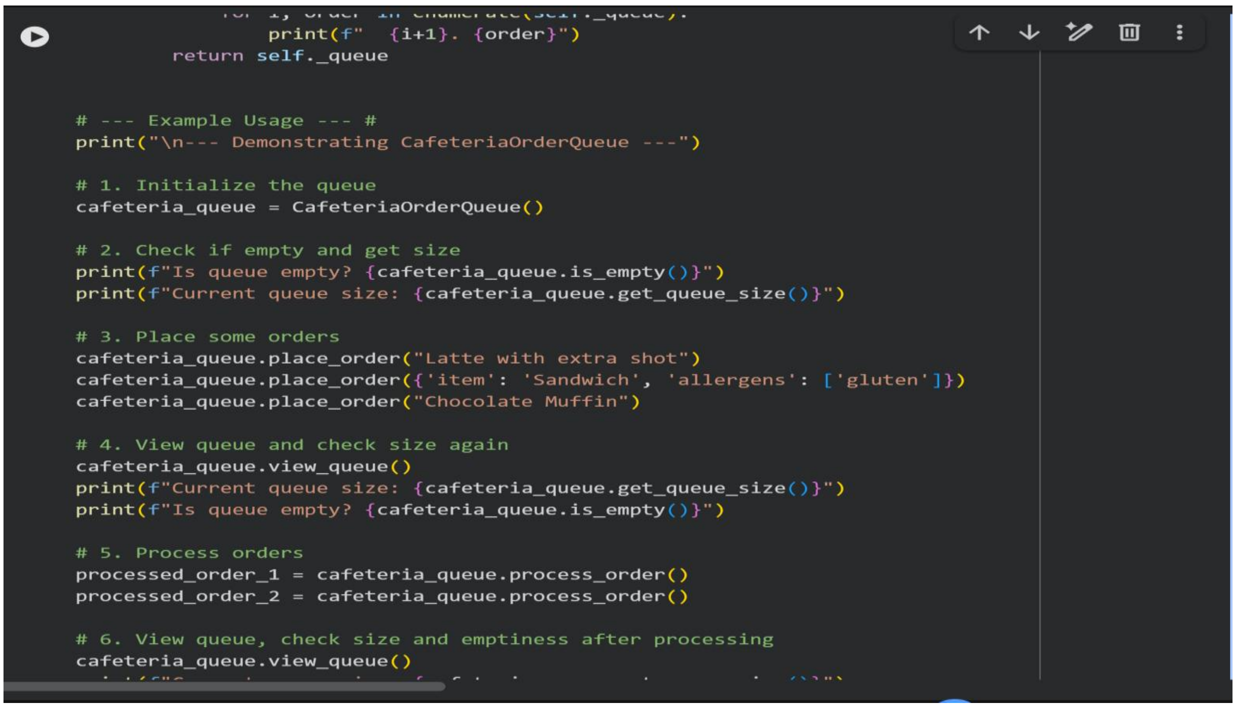
\



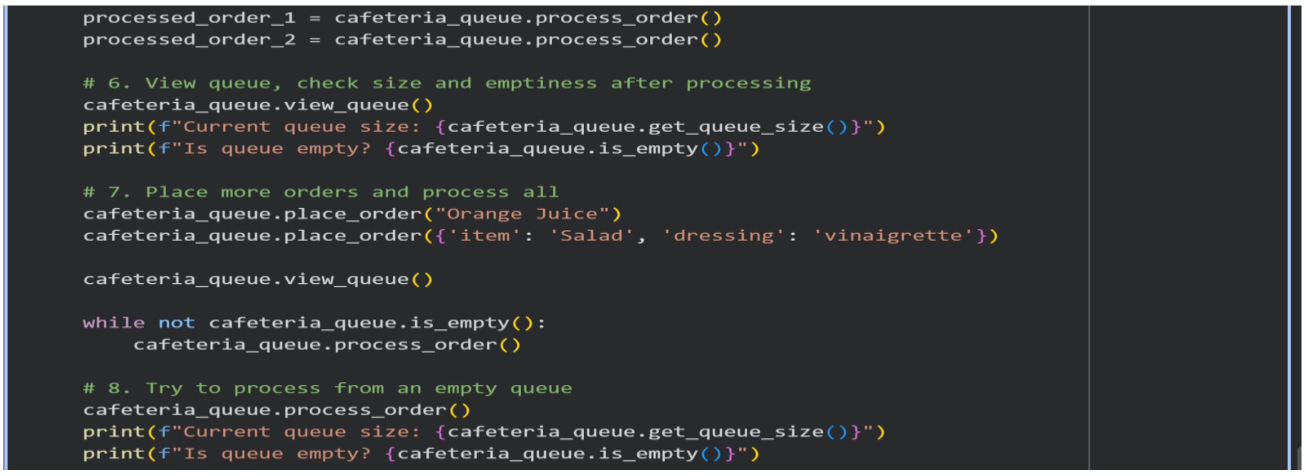
```python
class CafeteriaOrderQueue:
    """Simulates a cafeteria order queue using a list.

    Orders are processed on a First-In, First-Out (FIFO) basis.
    """

    def __init__(self):
        """Initializes an empty order queue."""
        self._queue = []
        print("Cafeteria Order Queue initialized.")

    def place_order(self, order_details):
        """Adds a new order to the end of the queue.

        Args:
            order_details (str or dict): Details of the order to be placed.
                                         Can be a simple string like 'Coffee' or
                                         a dictionary like {'item': 'Pizza', 'quantity': 1}.
        """
        self._queue.append(order_details)
        print(f"Order placed: {order_details}")

    def process_order(self):
        """Removes and returns the oldest order from the front of the queue.

        Returns:
            str or dict: The details of the processed order.
            None: If the queue is empty.
        """
        if not self.is_empty():
```



```python
            return processed_order
        else:
            print("No orders to process. The queue is empty.")
            return None

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

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

    def get_queue_size(self):
        """Returns the current number of orders in the queue.

        Returns:
            int: The number of orders in the queue.
        """
        return len(self._queue)

    def view_queue(self):
        """Displays all orders currently in the queue.

        Returns:
            list: A list of current orders in the queue.
        """
        if self.is_empty():
            print("The queue is currently empty.")
        else:
```



```python
                print(f"  {i+1}. {order}")
        return self._queue


# --- Example Usage --- #
print("\n--- Demonstrating CafeteriaOrderQueue ---")

# 1. Initialize the queue
cafeteria_queue = CafeteriaOrderQueue()

# 2. Check if empty and get size
print(f"Is queue empty? {cafeteria_queue.is_empty()}")
print(f"Current queue size: {cafeteria_queue.get_queue_size()}")

# 3. Place some orders
cafeteria_queue.place_order("Latte with extra shot")
cafeteria_queue.place_order({'item': 'Sandwich', 'allergens': ['gluten']})
cafeteria_queue.place_order("Chocolate Muffin")

# 4. View queue and check size again
cafeteria_queue.view_queue()
print(f"Current queue size: {cafeteria_queue.get_queue_size()}")
print(f"Is queue empty? {cafeteria_queue.is_empty()}")

# 5. Process orders
processed_order_1 = cafeteria_queue.process_order()
processed_order_2 = cafeteria_queue.process_order()

# 6. View queue, check size and emptiness after processing
cafeteria_queue.view_queue()
```



```python
processed_order_1 = cafeteria_queue.process_order()
processed_order_2 = cafeteria_queue.process_order()

# 6. View queue, check size and emptiness after processing
cafeteria_queue.view_queue()
print(f"Current queue size: {cafeteria_queue.get_queue_size()}")
print(f"Is queue empty? {cafeteria_queue.is_empty()}")

# 7. Place more orders and process all
cafeteria_queue.place_order("Orange Juice")
cafeteria_queue.place_order({'item': 'Salad', 'dressing': 'vinaigrette'})

cafeteria_queue.view_queue()

while not cafeteria_queue.is_empty():
    cafeteria_queue.process_order()

# 8. Try to process from an empty queue
cafeteria_queue.process_order()
print(f"Current queue size: {cafeteria_queue.get_queue_size()}")
print(f"Is queue empty? {cafeteria_queue.is_empty()}")
```

Final Task

Subtask:

Provide the generated table of features, chosen data structures, and justifications, along with the Python program for the Cafeteria Order Queue implementation.

Summary:

Data Analysis Key Findings

- **Student Attendance Tracking**: A Hash Table was selected as the most appropriate data structure due to its average O(1) time complexity for adding, removing, and looking up student records by ID, ensuring quick real-time access.
- **Event Registration System**: A Hash Table was chosen for efficiently managing registered participants with quick lookups, while a Queue was recommended for handling waiting lists on a first-come, first-served basis.
- **Library Book Borrowing**: A Hash Table was identified as suitable for tracking books and borrowers, enabling fast searches by ISBN or user ID.
- **Bus Scheduling System**: A Graph was deemed most appropriate for this system, as it effectively represents routes as edges and stops as nodes, supporting complex operations like optimal pathfinding.
- **Cafeteria Order Queue**: A Queue was chosen to model a first-come, first-served (FIFO) order processing system, ensuring fair and accurate handling of orders.
- **Data Structure Table Generation**: A markdown table was successfully created and populated, clearly mapping each feature to its chosen data structure and providing a concise justification for each selection.
- **Cafeteria Order Queue Implementation**: A Python class, CafeteriaOrderQueue, was successfully implemented using a Python list to simulate a FIFO queue. It includes methods for placing orders, processing orders, checking if the queue is empty, getting the queue size, and viewing the queue's contents. The implementation correctly demonstrated FIFO behavior and handled edge cases like processing from an empty queue.

Insights or Next Steps

- The exercise successfully demonstrated how different data structures are uniquely suited for specific real-time application requirements, prioritizing efficiency and logical order (e.g., O(1) average lookup for Hash Tables, FIFO for Queues).
- For the CafeteriaOrderQueue implementation, consider enhancing performance by using Python's collections.deque module, which offers more efficient appending and popping from both ends compared to a standard list's pop(0) operation.

ask 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.

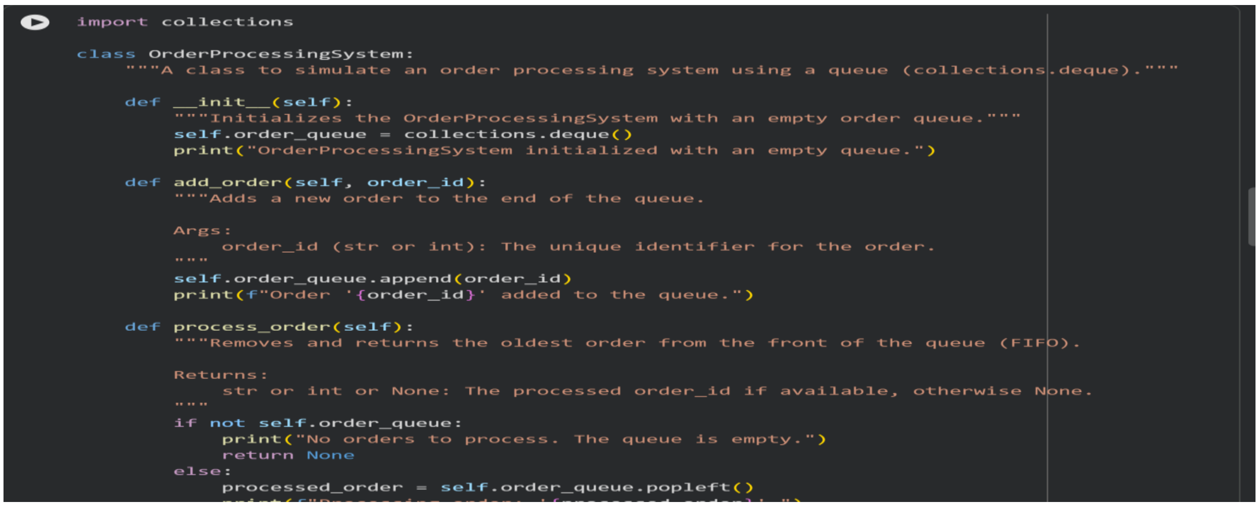
• Implement one selected feature as a working Python program with AI-assisted code generation.

Expected Output:

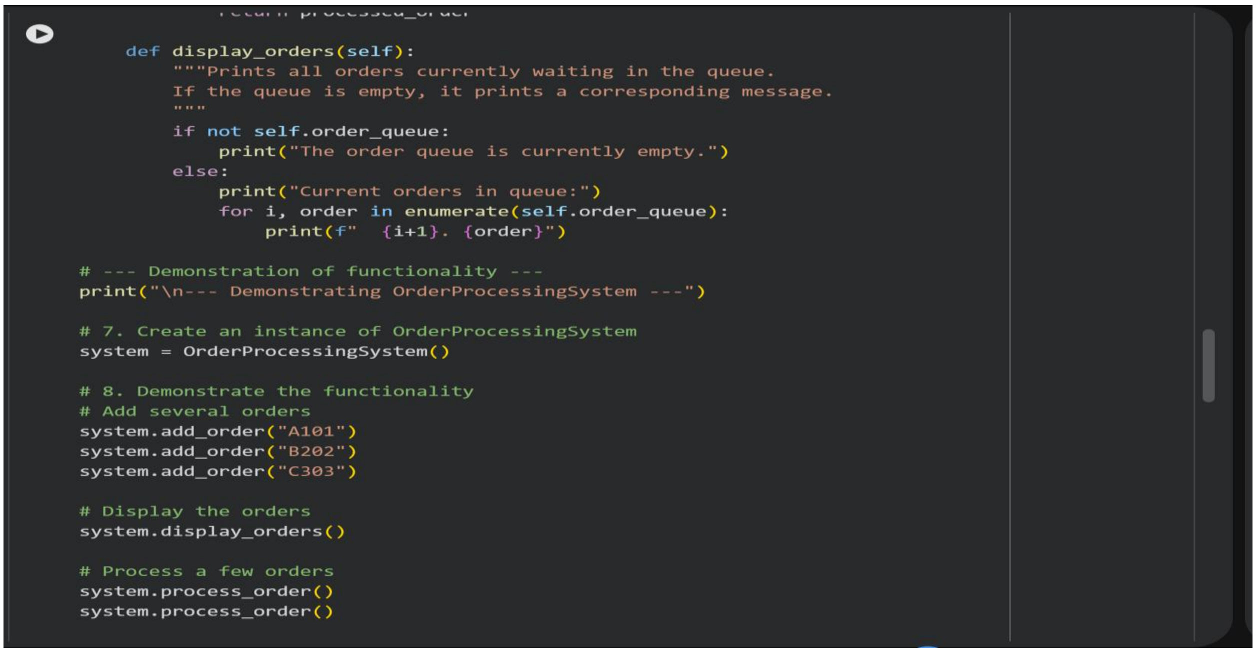
• A table mapping feature → chosen data structure → justification.

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

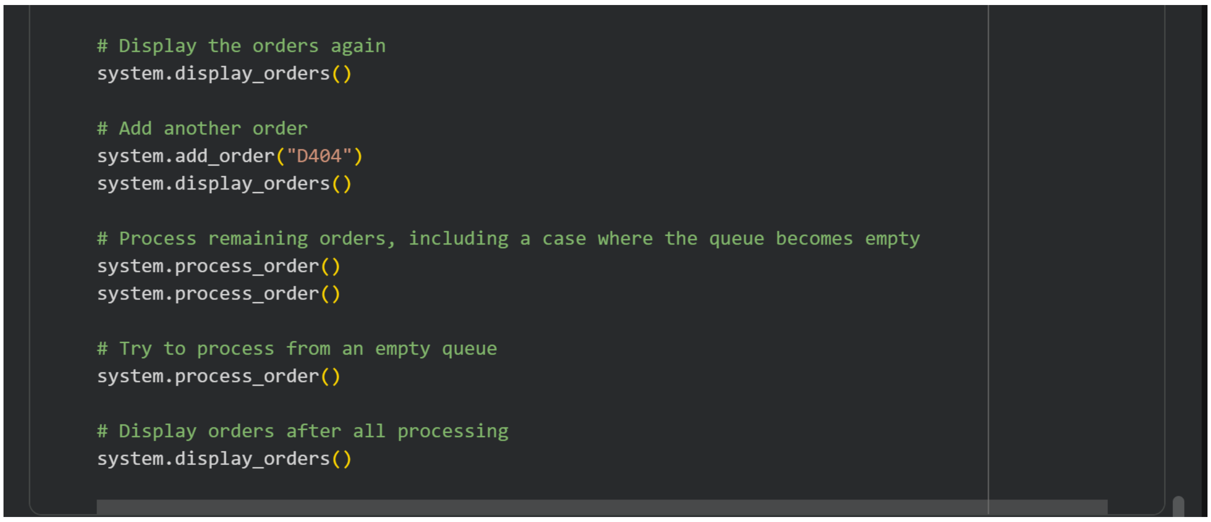
CODE:



```python
import collections

class OrderProcessingSystem:
    """A class to simulate an order processing system using a queue (collections.deque)."""

    def __init__(self):
        """Initializes the OrderProcessingSystem with an empty order queue."""
        self.order_queue = collections.deque()
        print("OrderProcessingSystem initialized with an empty queue.")

    def add_order(self, order_id):
        """Adds a new order to the end of the queue.

        Args:
            order_id (str or int): The unique identifier for the order.
        """
        self.order_queue.append(order_id)
        print(f"Order '{order_id}' added to the queue.")

    def process_order(self):
        """Removes and returns the oldest order from the front of the queue (FIFO).

        Returns:
            str or int or None: The processed order_id if available, otherwise None.
        """
        if not self.order_queue:
            print("No orders to process. The queue is empty.")
            return None
        else:
            processed_order = self.order_queue.popleft()
            print(f"Processing order: '{processed_order}'.")
```



```python
            return processed_order

    def display_orders(self):
        """Prints all orders currently waiting in the queue.
        If the queue is empty, it prints a corresponding message.
        """
        if not self.order_queue:
            print("The order queue is currently empty.")
        else:
            print("Current orders in queue:")
            for i, order in enumerate(self.order_queue):
                print(f"  {i+1}. {order}")

# --- Demonstration of functionality ---
print("\n--- Demonstrating OrderProcessingSystem ---")

# 7. Create an instance of OrderProcessingSystem
system = OrderProcessingSystem()

# 8. Demonstrate the functionality
# Add several orders
system.add_order("A101")
system.add_order("B202")
system.add_order("C303")

# Display the orders
system.display_orders()

# Process a few orders
system.process_order()
system.process_order()
```



```python
# Display the orders again
system.display_orders()

# Add another order
system.add_order("D404")
system.display_orders()

# Process remaining orders, including a case where the queue becomes empty
system.process_order()
system.process_order()

# Try to process from an empty queue
system.process_order()

# Display orders after all processing
system.display_orders()
```

Final Task

Subtask:

Present the table mapping features to chosen data structures with justifications. Also, provide the implemented Python program for the Order Processing System, demonstrating its functionality.

Summary:

Data Analysis Key Findings

- **E-commerce Feature to Data Structure Mapping:**
  - **Shopping Cart Management:** Best handled by a Linked List or Array List for efficient item insertion/deletion and order maintenance.
  - **Order Processing System:** A Queue (FIFO principle) is ideal for sequential order handling, ensuring fairness.
  - **Top-Selling Products Tracker:** A Min-Heap (or Priority Queue) efficiently stores and updates product sales, allowing quick retrieval of top N products.
  - **Product Search Engine:** A Trie (Prefix Tree) or Inverted Index optimizes keyword-based searches and autocomplete functionality with fast search times.
  - **Delivery Route Planning:** A Graph naturally represents locations and paths, enabling algorithms like Dijkstra's for optimal route finding.
- **Order Processing System Implementation:**
  - A Python OrderProcessingSystem class was successfully implemented using collections.deque as the underlying queue for efficient operations.
  - The add_order, process_order, and display_orders methods functioned correctly, adhering to the FIFO principle.
  - The system robustly handled empty queue scenarios during processing and display, printing appropriate messages.
  - The demonstration effectively showcased the adding, processing, and displaying of orders, confirming the system's functionality and adherence to requirements.

Insights or Next Steps

- The chosen data structures align well with the operational requirements of each e-commerce feature, demonstrating a strong understanding of data structure utility in real-world applications.

- The implemented Order Processing System provides a solid foundation; further development could include incorporating priority levels for urgent orders (using a priority queue) or persistent storage for orders.
