AI ASSISTANT CODING
ASSIGNMENT-11.5

Course Title : AI Assisted Coding

Name : Saini Kirthan

Batch No : 34

H.NO : 2303A52157

Lab 11 – Data Structures with AI: Implementing Fundamental Structures

Task Description #1 – Stack Implementation

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

Sample Input Code:

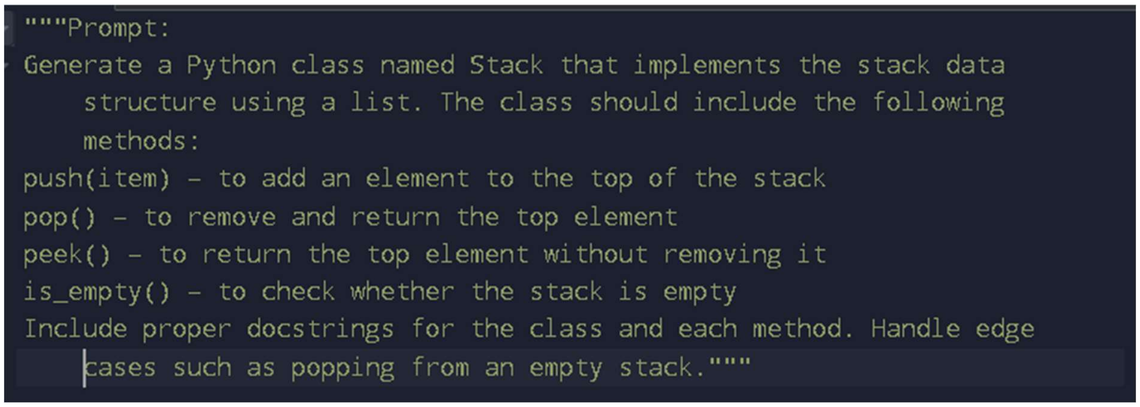
class Stack:

pass

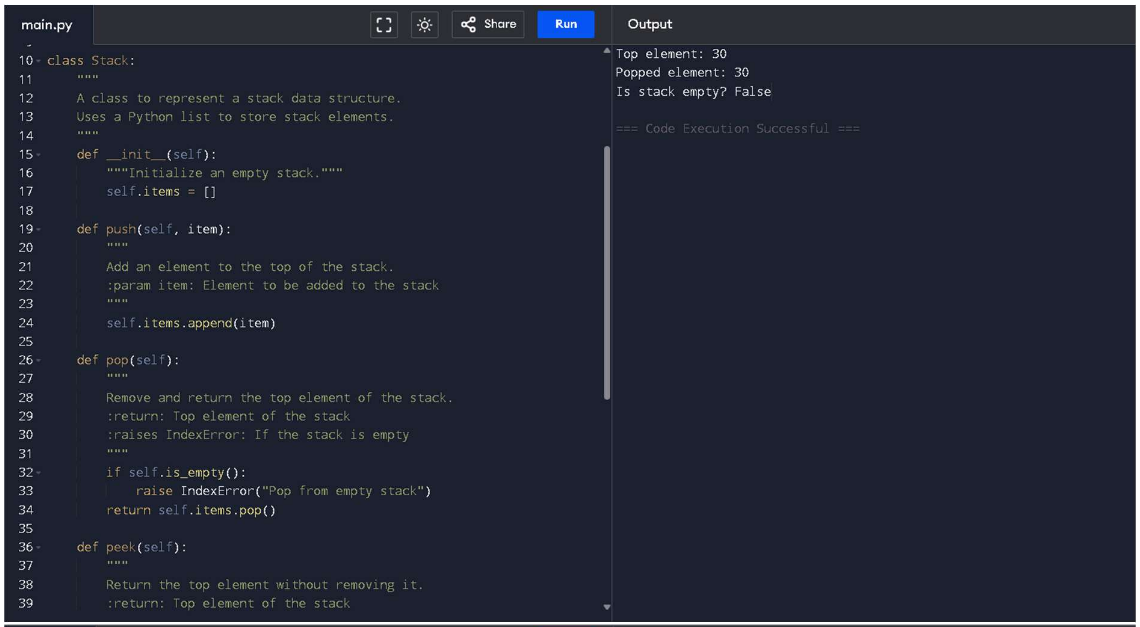
Expected Output:

• A functional stack implementation with all required methods and

docstrings.



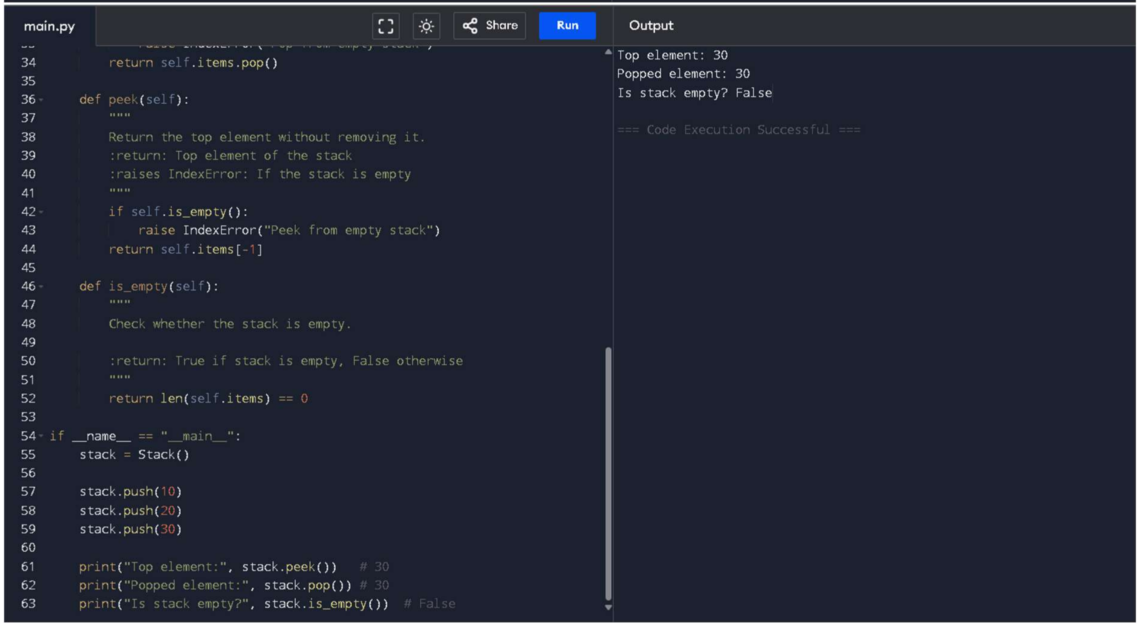
```
"""Prompt:
Generate a Python class named Stack that implements the stack data
    structure using a list. The class should include the following
    methods:
push(item) - to add an element to the top of the stack
pop() - to remove and return the top element
peek() - to return the top element without removing it
is_empty() - to check whether the stack is empty
Include proper docstrings for the class and each method. Handle edge
    cases such as popping from an empty stack."""
```


main.py

```python
class Stack:
    """
    A class to represent a stack data structure.
    Uses a Python list to store stack elements.
    """
    def __init__(self):
        """Initialize an empty stack."""
        self.items = []

    def push(self, item):
        """
        Add an element to the top of the stack.
        :param item: Element to be added to the stack
        """
        self.items.append(item)

    def pop(self):
        """
        Remove and return the top element of the stack.
        :return: Top element 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):
        """
        Return the top element without removing it.
        :return: Top element of the stack
```

Output

```
Top element: 30
Popped element: 30
Is stack empty? False

=== Code Execution Successful ===
```


main.py

```python
        return self.items.pop()

    def peek(self):
        """
        Return the top element without removing it.
        :return: Top element 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):
        """
        Check whether the stack is empty.

        :return: True if stack is empty, False otherwise
        """
        return len(self.items) == 0

if __name__ == "__main__":
    stack = Stack()

    stack.push(10)
    stack.push(20)
    stack.push(30)

    print("Top element:", stack.peek())   # 30
    print("Popped element:", stack.pop()) # 30
    print("Is stack empty?", stack.is_empty())  # False
```

Output

```
Top element: 30
Popped element: 30
Is stack empty? False

=== Code Execution Successful ===
```

Task Description #2 – Queue Implementation

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

Sample Input Code:

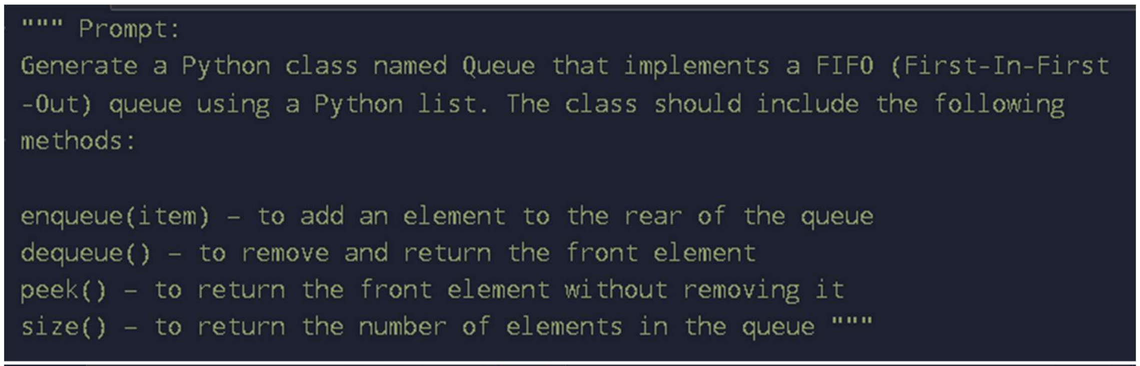
class Queue:

pass

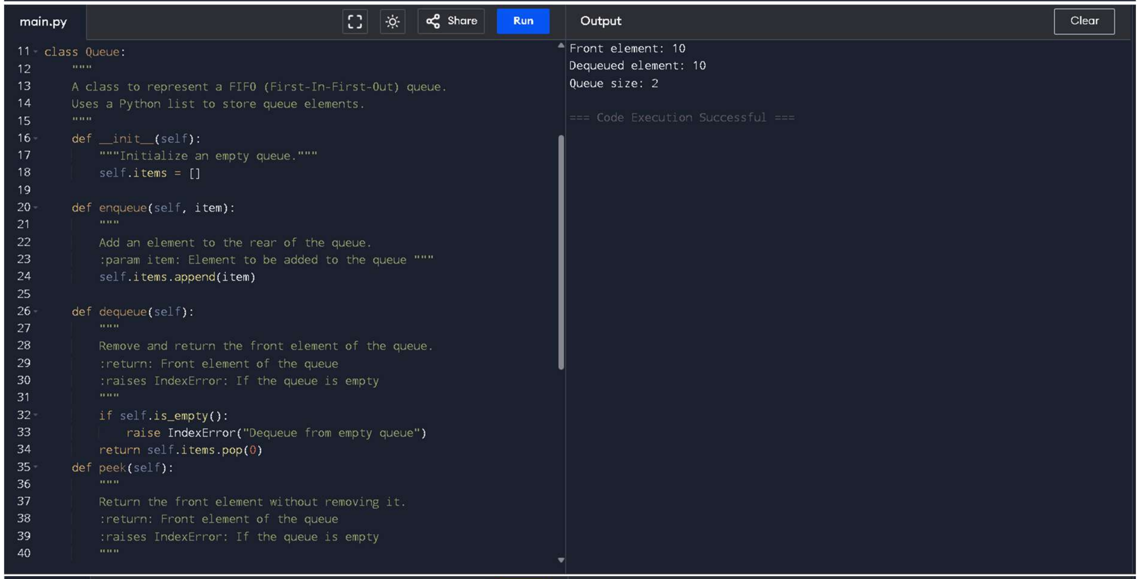
Expected Output:

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

methods.



```
""" Prompt:
Generate a Python class named Queue that implements a FIFO (First-In-First
-Out) queue using a Python list. The class should include the following
methods:

enqueue(item) – to add an element to the rear of the queue
dequeue() – to remove and return the front element
peek() – to return the front element without removing it
size() – to return the number of elements in the queue """
```

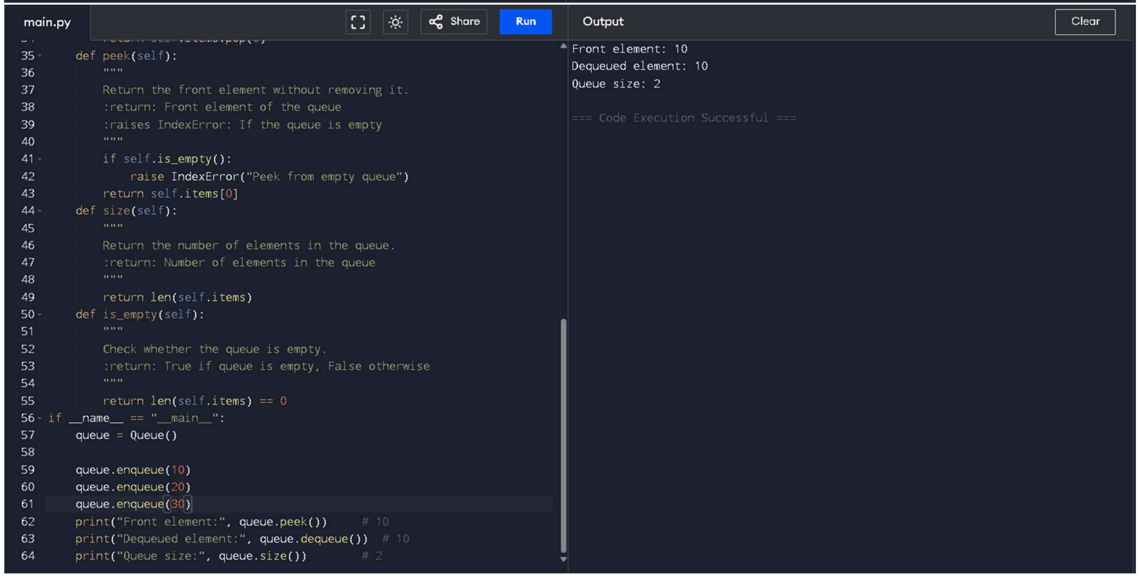


main.py

```python
class Queue:
    """
    A class to represent a FIFO (First-In-First-Out) queue.
    Uses a Python list to store queue elements.
    """
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []

    def enqueue(self, item):
        """
        Add an element to the rear of the queue.
        :param item: Element to be added to the queue """
        self.items.append(item)

    def dequeue(self):
        """
        Remove and return the front element of the queue.
        :return: Front element 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)
    def peek(self):
        """
        Return the front element without removing it.
        :return: Front element of the queue
        :raises IndexError: If the queue is empty
        """
```

Output

```
Front element: 10
Dequeued element: 10
Queue size: 2

=== Code Execution Successful ===
```



main.py

```python
    def peek(self):
        """
        Return the front element without removing it.
        :return: Front element 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 size(self):
        """
        Return the number of elements in the queue.
        :return: Number of elements in the queue
        """
        return len(self.items)
    def is_empty(self):
        """
        Check whether the queue is empty.
        :return: True if queue is empty, False otherwise
        """
        return len(self.items) == 0
if __name__ == "__main__":
    queue = Queue()

    queue.enqueue(10)
    queue.enqueue(20)
    queue.enqueue(30)
    print("Front element:", queue.peek())      # 10
    print("Dequeued element:", queue.dequeue())  # 10
    print("Queue size:", queue.size())         # 2
```

Output

```
Front element: 10
Dequeued element: 10
Queue size: 2

=== Code Execution Successful ===
```

Task Description #3 – Linked List

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

Sample Input Code:

class Node:

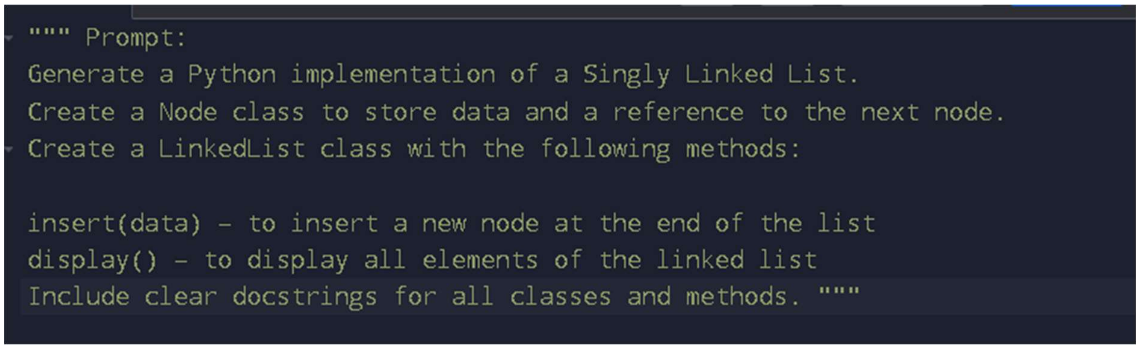
pass

class LinkedList:

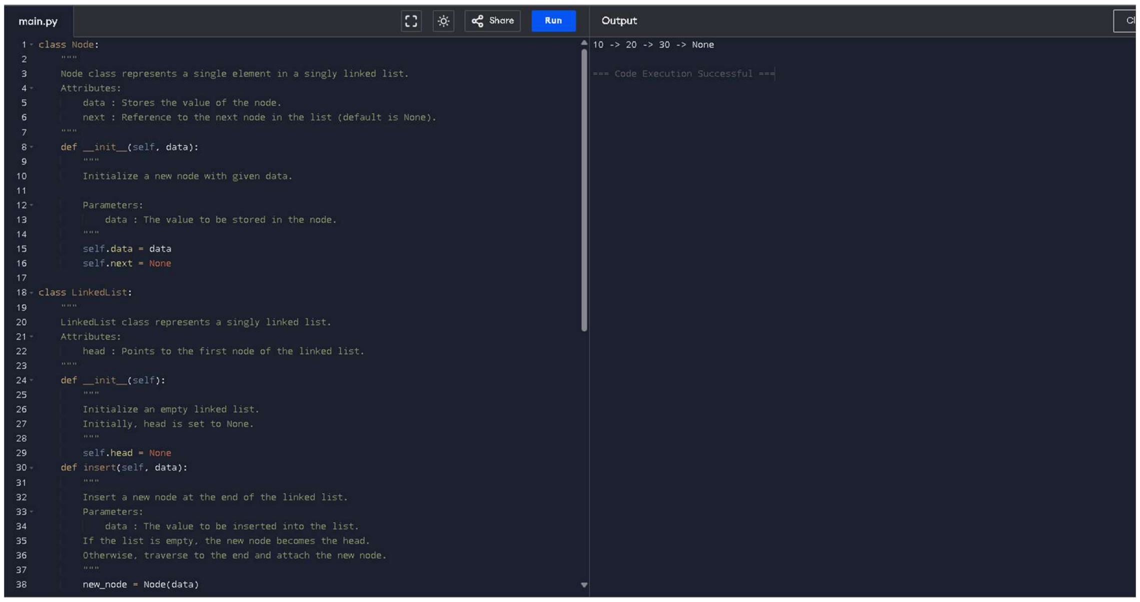
pass

Expected Output:

• A working linked list implementation with clear method documentation

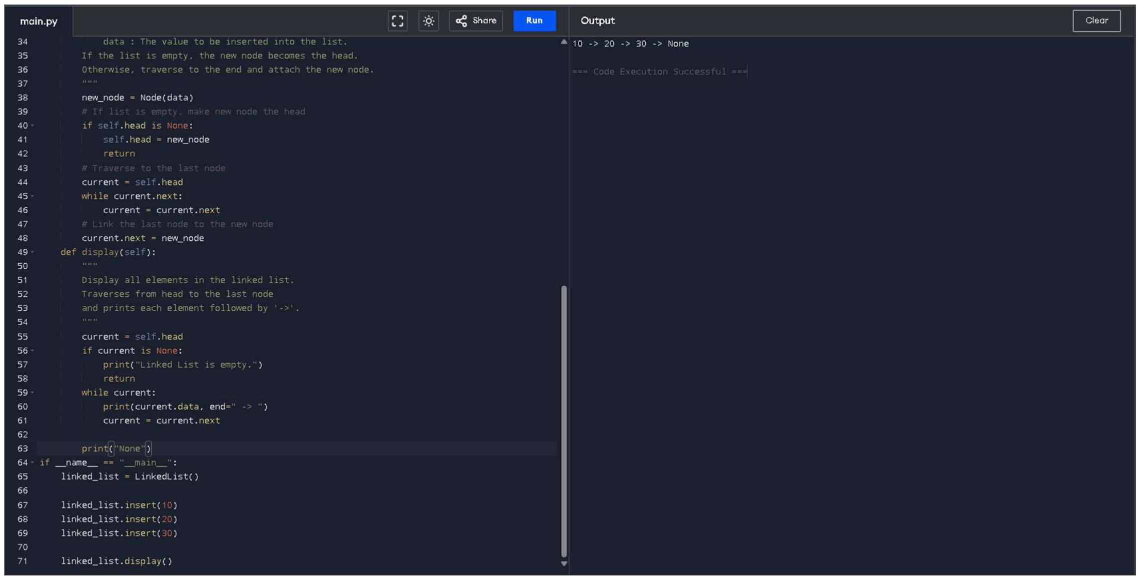


```
""" Prompt:
Generate a Python implementation of a Singly Linked List.
Create a Node class to store data and a reference to the next node.
Create a LinkedList class with the following methods:

insert(data) – to insert a new node at the end of the list
display() – to display all elements of the linked list
Include clear docstrings for all classes and methods. """
```



```python
class Node:
    """
    Node class represents a single element in a singly linked list.
    Attributes:
        data : Stores the value of the node.
        next : Reference to the next node in the list (default is None).
    """
    def __init__(self, data):
        """
        Initialize a new node with given data.

        Parameters:
            data : The value to be stored in the node.
        """
        self.data = data
        self.next = None

class LinkedList:
    """
    LinkedList class represents a singly linked list.
    Attributes:
        head : Points to the first node of the linked list.
    """
    def __init__(self):
        """
        Initialize an empty linked list.
        Initially, head is set to None.
        """
        self.head = None
    def insert(self, data):
        """
        Insert a new node at the end of the linked list.
        Parameters:
            data : The value to be inserted into the list.
        If the list is empty, the new node becomes the head.
        Otherwise, traverse to the end and attach the new node.
        """
        new_node = Node(data)
```

```
10 -> 20 -> 30 -> None

=== Code Execution Successful ===
```



```python
            data : The value to be inserted into the list.
        If the list is empty, the new node becomes the head.
        Otherwise, traverse to the end and attach the new node.
        """
        new_node = Node(data)
        # If list is empty, make new node the head
        if self.head is None:
            self.head = new_node
            return
        # Traverse to the last node
        current = self.head
        while current.next:
            current = current.next
        # Link the last node to the new node
        current.next = new_node
    def display(self):
        """
        Display all elements in the linked list.
        Traverses from head to the last node
        and prints each element followed by '->'.
        """
        current = self.head
        if current is None:
            print("Linked List is empty.")
            return
        while current:
            print(current.data, end=" -> ")
            current = current.next

        print("None")
if __name__ == "__main__":
    linked_list = LinkedList()

    linked_list.insert(10)
    linked_list.insert(20)
    linked_list.insert(30)

    linked_list.display()
```

```
10 -> 20 -> 30 -> None

=== Code Execution Successful ===
```

Task Description #4 – Hash Table

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

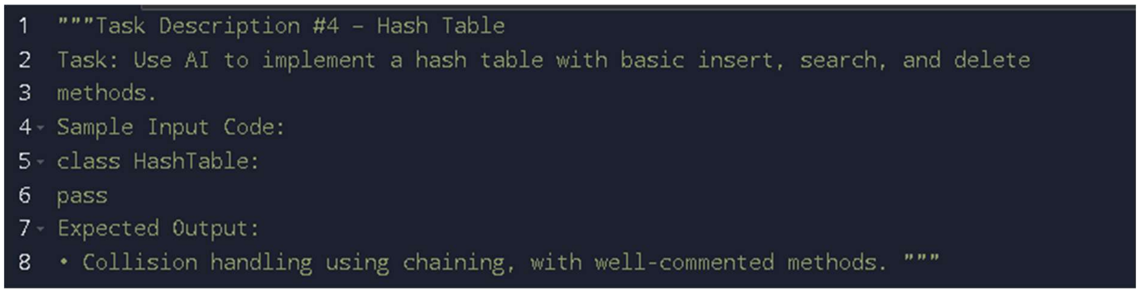
Sample Input Code:

class HashTable:

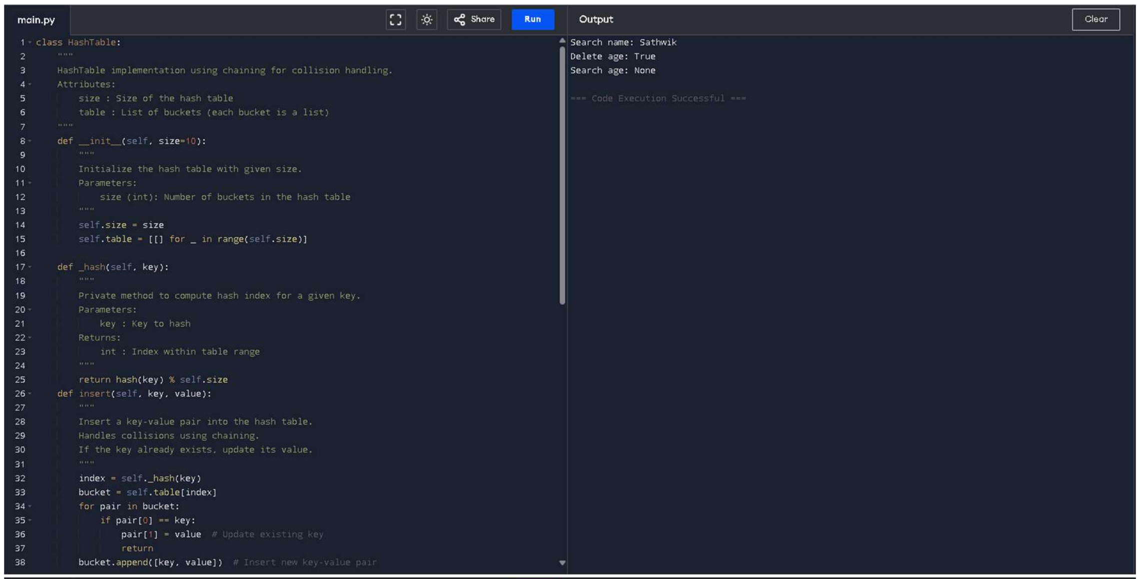
pass

Expected Output:

• Collision handling using chaining, with well-commented methods



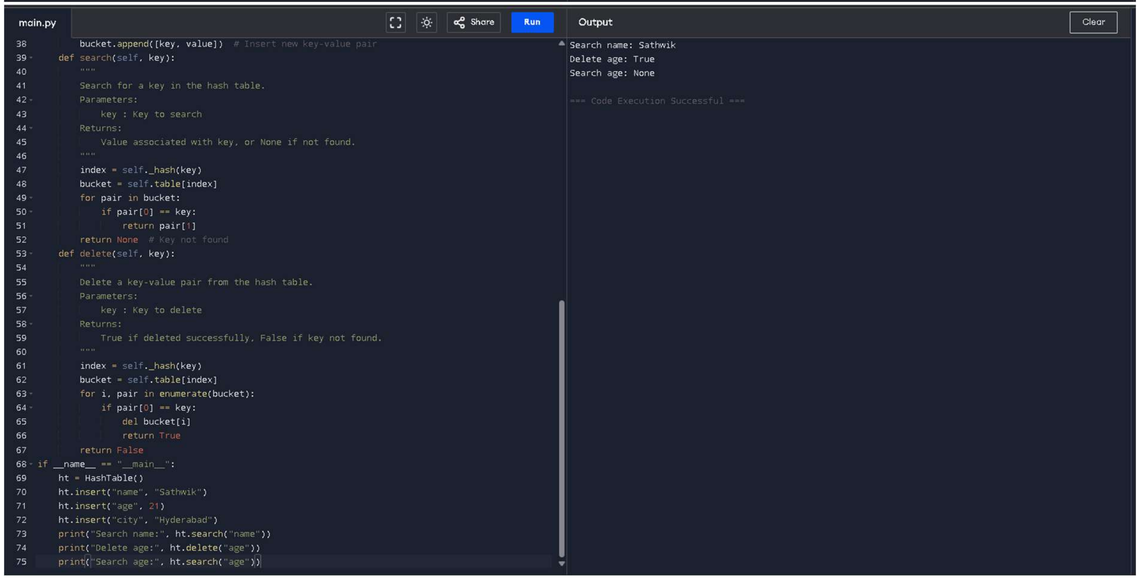
```
"""Task Description #4 – Hash Table
Task: Use AI to implement a hash table with basic insert, search, and delete
methods.
Sample Input Code:
class HashTable:
pass
Expected Output:
• Collision handling using chaining, with well-commented methods. """
```


main.py

```python
class HashTable:
    """
    HashTable implementation using chaining for collision handling.
    Attributes:
        size : Size of the hash table
        table : List of buckets (each bucket is a list)
    """
    def __init__(self, size=10):
        """
        Initialize the hash table with given size.
        Parameters:
            size (int): Number of buckets in the hash table
        """
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def _hash(self, key):
        """
        Private method to compute hash index for a given key.
        Parameters:
            key : Key to hash
        Returns:
            int : Index within table range
        """
        return hash(key) % self.size
    def insert(self, key, value):
        """
        Insert a key-value pair into the hash table.
        Handles collisions using chaining.
        If the key already exists, update its value.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for pair in bucket:
            if pair[0] == key:
                pair[1] = value  # Update existing key
                return
        bucket.append([key, value])  # Insert new key-value pair
```

Output

```
Search name: Sathwik
Delete age: True
Search age: None

=== Code Execution Successful ===
```


main.py

```python
        bucket.append([key, value])  # Insert new key-value pair
    def search(self, key):
        """
        Search for a key in the hash table.
        Parameters:
            key : Key to search
        Returns:
            Value associated with key, or None if not found.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for pair in bucket:
            if pair[0] == key:
                return pair[1]
        return None  # Key not found
    def delete(self, key):
        """
        Delete a key-value pair from the hash table.
        Parameters:
            key : Key to delete
        Returns:
            True if deleted successfully, False if key not found.
        """
        index = self._hash(key)
        bucket = self.table[index]
        for i, pair in enumerate(bucket):
            if pair[0] == key:
                del bucket[i]
                return True
        return False
if __name__ == "__main__":
    ht = HashTable()
    ht.insert("name", "Sathwik")
    ht.insert("age", 21)
    ht.insert("city", "Hyderabad")
    print("Search name:", ht.search("name"))
    print("Delete age:", ht.delete("age"))
    print("Search age:", ht.search("age"))
```

Output

```
Search name: Sathwik
Delete age: True
Search age: None

=== Code Execution Successful ===
```

Task Description #5 – Graph Representation

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

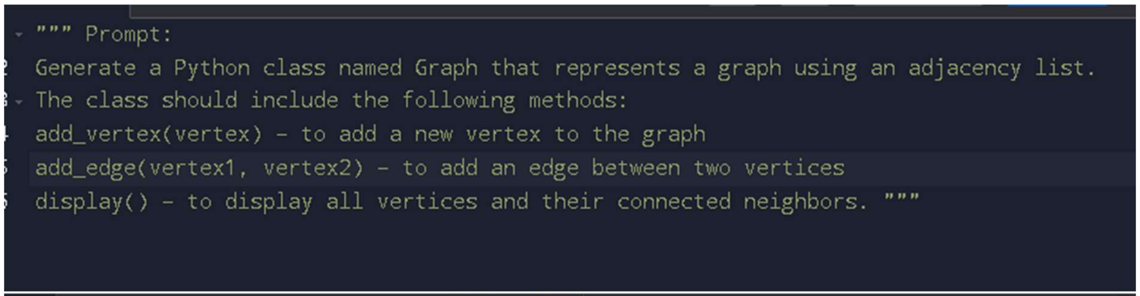
Sample Input Code:

class Graph:

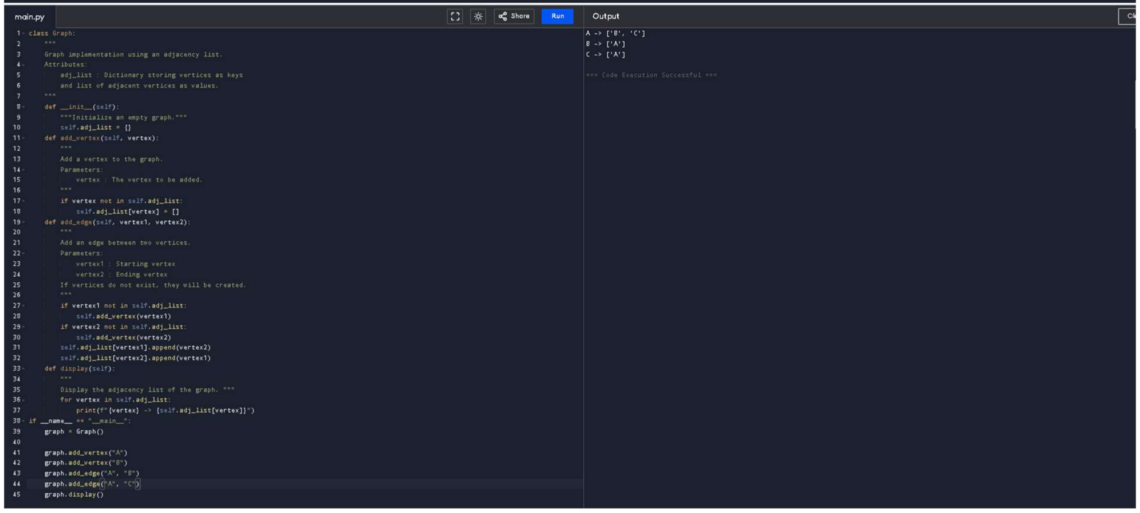
pass

Expected Output:

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



```
""" Prompt:
Generate a Python class named Graph that represents a graph using an adjacency list.
The class should include the following methods:
add_vertex(vertex) - to add a new vertex to the graph
add_edge(vertex1, vertex2) - to add an edge between two vertices
display() - to display all vertices and their connected neighbors. """
```



```python
class Graph:
    """
    Graph implementation using an adjacency list.
    Attributes:
        adj_list : Dictionary storing vertices as keys
        and list of adjacent vertices as values.
    """
    def __init__(self):
        """Initialize an empty graph."""
        self.adj_list = {}
    def add_vertex(self, vertex):
        """
        Add a vertex to the graph.
        Parameters:
            vertex : The vertex to be added.
        """
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []
    def add_edge(self, vertex1, vertex2):
        """
        Add an edge between two vertices.
        Parameters:
            vertex1 : Starting vertex
            vertex2 : Ending vertex
        If vertices do not exist, they will be created.
        """
        if vertex1 not in self.adj_list:
            self.add_vertex(vertex1)
        if vertex2 not in self.adj_list:
            self.add_vertex(vertex2)
        self.adj_list[vertex1].append(vertex2)
        self.adj_list[vertex2].append(vertex1)
    def display(self):
        """
        Display the adjacency list of the graph. """
        for vertex in self.adj_list:
            print(f"{vertex} -> {self.adj_list[vertex]}")
if __name__ == "__main__":
    graph = Graph()

    graph.add_vertex("A")
    graph.add_vertex("B")
    graph.add_edge("A", "B")
    graph.add_edge("A", "C")
    graph.display()
```

```
A -> ['B', 'C']
B -> ['A']
C -> ['A']

=== Code Execution Successful ===
```

Task Description #6: Smart Hospital Management System – Data Structure Selection

A hospital wants to develop a Smart Hospital Management System that handles:

1. Patient Check-In System – Patients are registered and treated in order of arrival.

2. Emergency Case Handling – Critical patients must be treated first.

3. Medical Records Storage – Fast retrieval of patient details using ID.

4. Doctor Appointment Scheduling – Appointments sorted by time.

5. Hospital Room Navigation – Represent connections between wards and rooms.

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.

Smart Hospital Management System – Data Structure Selection

---

1. Patient Check-In System

Selected Data Structure: Queue

Justification:
A Queue follows the First-In-First-Out (FIFO) principle. Patients are treated in the order they arrive, ensuring fairness. Therefore, a Queue is the most suitable structure for managing regular patient check-ins.

---

2. Emergency Case Handling

Selected Data Structure: Priority Queue

Justification:
Emergency patients must be treated based on severity rather than arrival time. A Priority Queue allows higher-priority (critical) patients to be treated first. This ensures immediate attention to life-threatening cases.

---

3. Medical Records Storage

Selected Data Structure: Hash Table

Justification:
Patient records need fast retrieval using a unique patient ID. A Hash Table provides average O(1) time complexity for search, insert, and delete operations. This makes record management efficient and scalable.

---

4. Doctor Appointment Scheduling

Selected Data Structure: Binary Search Tree (BST)

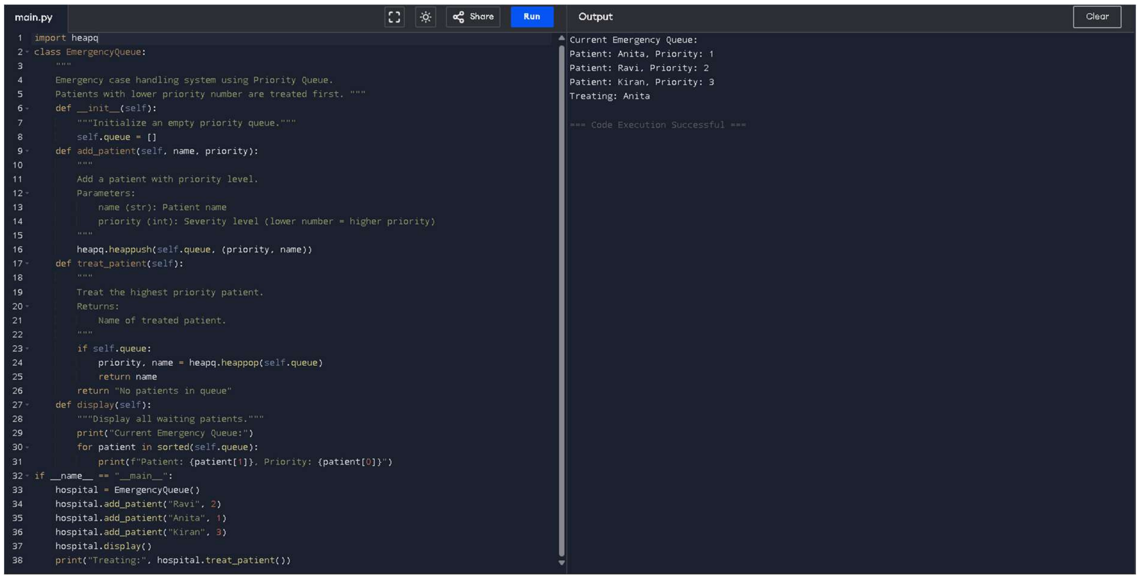
Justification:
Appointments must be maintained in sorted order based on time. A Binary Search Tree automatically keeps elements sorted. This makes scheduling and time-based retrieval efficient.

---

5. Hospital Room Navigation

Selected Data Structure: Graph

Justification:
Hospital rooms and wards can be represented as nodes, and pathways as edges. A Graph effectively models connections between locations. It also supports pathfinding for navigation within the hospital.



```python
import heapq
class EmergencyQueue:
    """
    Emergency case handling system using Priority Queue.
    Patients with lower priority number are treated first. """
    def __init__(self):
        """Initialize an empty priority queue."""
        self.queue = []
    def add_patient(self, name, priority):
        """
        Add a patient with priority level.
        Parameters:
            name (str): Patient name
            priority (int): Severity level (lower number = higher priority)
        """
        heapq.heappush(self.queue, (priority, name))
    def treat_patient(self):
        """
        Treat the highest priority patient.
        Returns:
            Name of treated patient.
        """
        if self.queue:
            priority, name = heapq.heappop(self.queue)
            return name
        return "No patients in queue"
    def display(self):
        """Display all waiting patients."""
        print("Current Emergency Queue:")
        for patient in sorted(self.queue):
            print(f"Patient: {patient[1]}, Priority: {patient[0]}")
if __name__ == "__main__":
    hospital = EmergencyQueue()
    hospital.add_patient("Ravi", 2)
    hospital.add_patient("Anita", 1)
    hospital.add_patient("Kiran", 3)
    hospital.display()
    print("Treating:", hospital.treat_patient())
```

```
Current Emergency Queue:
Patient: Anita, Priority: 1
Patient: Ravi, Priority: 2
Patient: Kiran, Priority: 3
Treating: Anita

=== Code Execution Successful ===
```

Task Description #7: Smart City Traffic Control System

A city plans a Smart Traffic Management System that includes:

1. Traffic Signal Queue – Vehicles waiting at signals.

2. Emergency Vehicle Priority Handling – Ambulances and fire trucks prioritized.

3. Vehicle Registration Lookup – Instant access to vehicle details.

4. Road Network Mapping – Roads and intersections connected logically.

5. Parking Slot Availability – Track available and occupied slots.

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.

Smart City Traffic Control System

---

1. Traffic Signal Queue

Selected Data Structure: Queue

Justification:
Vehicles waiting at a traffic signal follow the First-In-First-Out (FIFO) principle. The vehicle that arrives first should move first when the signal turns green. Therefore, a Queue is the most appropriate structure for managing traffic signals fairly and efficiently.

---

2. Emergency Vehicle Priority Handling

Selected Data Structure: Priority Queue

Justification:
Emergency vehicles such as ambulances and fire trucks must be given immediate access regardless of arrival time. A Priority Queue allows higher-priority vehicles to be processed before regular vehicles. This ensures faster emergency response and improved public safety.

---

3. Vehicle Registration Lookup

Selected Data Structure: Hash Table

Justification:
Vehicle details must be accessed instantly using a registration number. A Hash Table provides average O(1) time complexity for search operations. This makes it ideal for fast and efficient vehicle information retrieval.

---

4. Road Network Mapping

Selected Data Structure: Graph

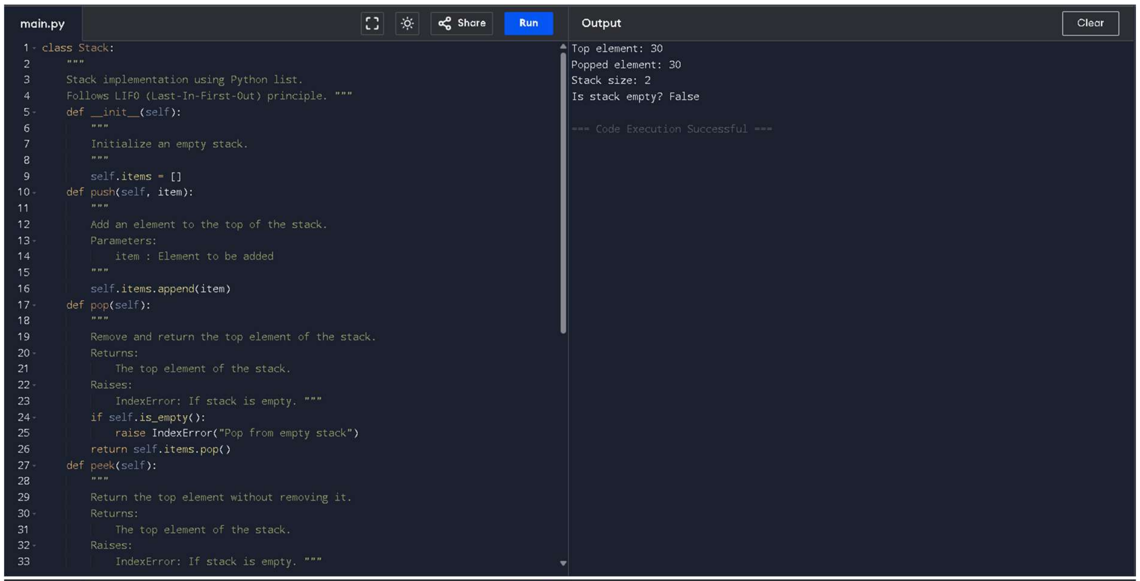
Justification:
Roads and intersections can be represented as nodes and edges. A Graph efficiently models connections between locations and supports route-finding algorithms. This makes it suitable for navigation and traffic management systems.

---

5. Parking Slot Availability

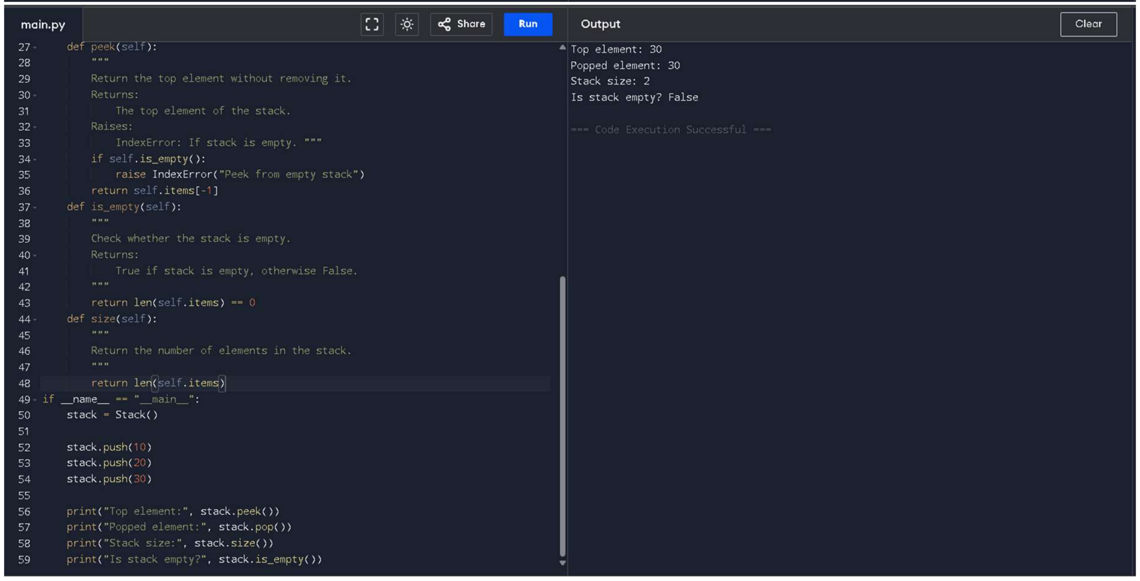
Selected Data Structure: Binary Search Tree (BST)

Justification:
Parking slots can be stored in sorted order based on slot numbers. A Binary Search Tree maintains elements in sorted order and allows efficient insertion, deletion, and searching. This makes it suitable for tracking available and occupied parking slots.



```python
class Stack:
    """
    Stack implementation using Python list.
    Follows LIFO (Last-In-First-Out) principle. """
    def __init__(self):
        """
        Initialize an empty stack.
        """
        self.items = []
    def push(self, item):
        """
        Add an element to the top of the stack.
        Parameters:
            item : Element to be added
        """
        self.items.append(item)
    def pop(self):
        """
        Remove and return the top element of the stack.
        Returns:
            The top element of the stack.
        Raises:
            IndexError: If stack is empty. """
        if self.is_empty():
            raise IndexError("Pop from empty stack")
        return self.items.pop()
    def peek(self):
        """
        Return the top element without removing it.
        Returns:
            The top element of the stack.
        Raises:
            IndexError: If stack is empty. """
```

```
Top element: 30
Popped element: 30
Stack size: 2
Is stack empty? False

=== Code Execution Successful ===
```



```python
    def peek(self):
        """
        Return the top element without removing it.
        Returns:
            The top element of the stack.
        Raises:
            IndexError: If stack is empty. """
        if self.is_empty():
            raise IndexError("Peek from empty stack")
        return self.items[-1]
    def is_empty(self):
        """
        Check whether the stack is empty.
        Returns:
            True if stack is empty, otherwise False.
        """
        return len(self.items) == 0
    def size(self):
        """
        Return the number of elements in the stack.
        """
        return len(self.items)
if __name__ == "__main__":
    stack = Stack()

    stack.push(10)
    stack.push(20)
    stack.push(30)

    print("Top element:", stack.peek())
    print("Popped element:", stack.pop())
    print("Stack size:", stack.size())
    print("Is stack empty?", stack.is_empty())
```

```
Top element: 30
Popped element: 30
Stack size: 2
Is stack empty? False

=== Code Execution Successful ===
```

Task Description #8: 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.

Smart E-Commerce Platform – Data Structure Selection

---

1. Shopping Cart Management

Selected Data Structure: Linked List

Justification:
Products in a shopping cart are added and removed dynamically. A Linked List allows

efficient insertion and deletion without shifting elements. This makes it suitable for managing frequently changing cart items.

---

2. Order Processing System

Selected Data Structure: Queue

Justification:
Orders must be processed in the order they are placed. A Queue follows the First-In-First-Out (FIFO) principle, ensuring fairness. Therefore, it is ideal for handling customer orders.

---

3. Top-Selling Products Tracker

Selected Data Structure: Priority Queue

Justification:
Products need to be ranked based on sales count. A Priority Queue allows items with higher sales to be accessed first. This ensures efficient tracking of top-selling products.

---

4. Product Search Engine

Selected Data Structure: Hash Table

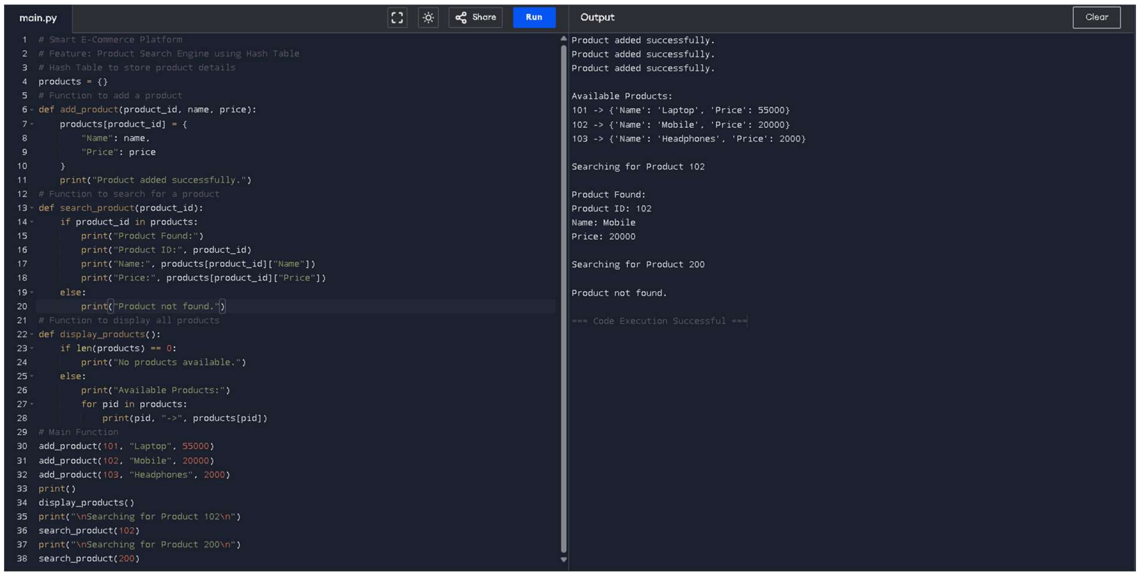
Justification:
Products are searched using unique product IDs. A Hash Table provides average O(1) time complexity for search operations. This makes product lookup fast and efficient.

---

5. Delivery Route Planning

Selected Data Structure: Graph

Justification:
Warehouses and delivery locations can be represented as nodes, and roads as edges. A Graph efficiently models connections and supports shortest path algorithms. This makes it suitable for route optimization and logistics planning.



```python
# Smart E-Commerce Platform
# Feature: Product Search Engine using Hash Table
# Hash Table to store product details
products = {}
# Function to add a product
def add_product(product_id, name, price):
    products[product_id] = {
        "Name": name,
        "Price": price
    }
    print("Product added successfully.")
# Function to search for a product
def search_product(product_id):
    if product_id in products:
        print("Product Found:")
        print("Product ID:", product_id)
        print("Name:", products[product_id]["Name"])
        print("Price:", products[product_id]["Price"])
    else:
        print("Product not found.")
# Function to display all products
def display_products():
    if len(products) == 0:
        print("No products available.")
    else:
        print("Available Products:")
        for pid in products:
            print(pid, "->", products[pid])
# Main Function
add_product(101, "Laptop", 55000)
add_product(102, "Mobile", 20000)
add_product(103, "Headphones", 2000)
print()
display_products()
print("\nSearching for Product 102\n")
search_product(102)
print("\nSearching for Product 200\n")
search_product(200)
```

```
Product added successfully.
Product added successfully.
Product added successfully.

Available Products:
101 -> {'Name': 'Laptop', 'Price': 55000}
102 -> {'Name': 'Mobile', 'Price': 20000}
103 -> {'Name': 'Headphones', 'Price': 2000}

Searching for Product 102

Product Found:
Product ID: 102
Name: Mobile
Price: 20000

Searching for Product 200

Product not found.

=== Code Execution Successful ===
```