ASSIGNMENT-11.1

M.Paramesh

HNO: 2303A53021

Batch-46

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 used:

Generate a Python Stack class with the following methods:(), pop(), peek(), and is_empty(). Include proper docstrings for each method.

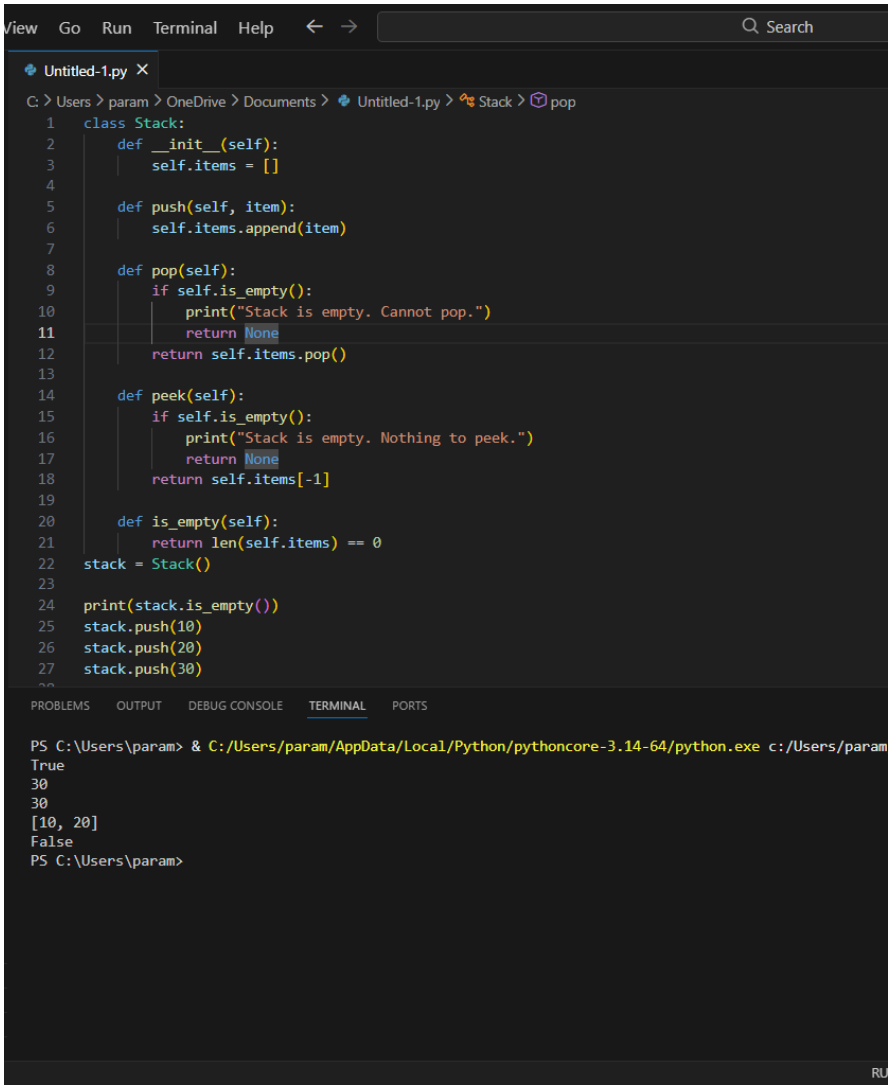
Code :

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.is_empty():
```

```
            print("Stack is empty. Cannot pop.")
            return None
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            print("Stack is empty. Nothing to peek.")
            return None
        return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0
stack = Stack()

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

print(stack.peek())
print(stack.pop())
print(stack.items)
print(stack.is_empty())
```

explanation : A stack follows **LIFO (Last In, First Out)** order.
This class uses a Python list to efficiently implement stack operations like push, pop, and peek.

Output:



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 used:

Generate a Queue data structure in Python using lists. The queue should follow FIFO order and support enqueue, dequeue, peek, and size operations.

Code:

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)
        print(item, "enqueued")

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty")
        else:
            print(self.queue.pop(0), "dequeued")

    def peek(self):
        if self.is_empty():
            print("Queue is empty")
        else:
            print("Front element:", self.queue[0])

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

    def is_empty(self):
        return len(self.queue) == 0
q = Queue()

q.enqueue(10)
```

q.enqueue(20)

q.enqueue(30)

q.peek()

q.size()

q.dequeue()

q.peek()

q.size()

explaination :

This Queue implementation uses a Python list and follows the **FIFO (First In First Out)** principle.
Elements are added using enqueue() and removed from the front using dequeue().

Output:

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

Prompt Used:

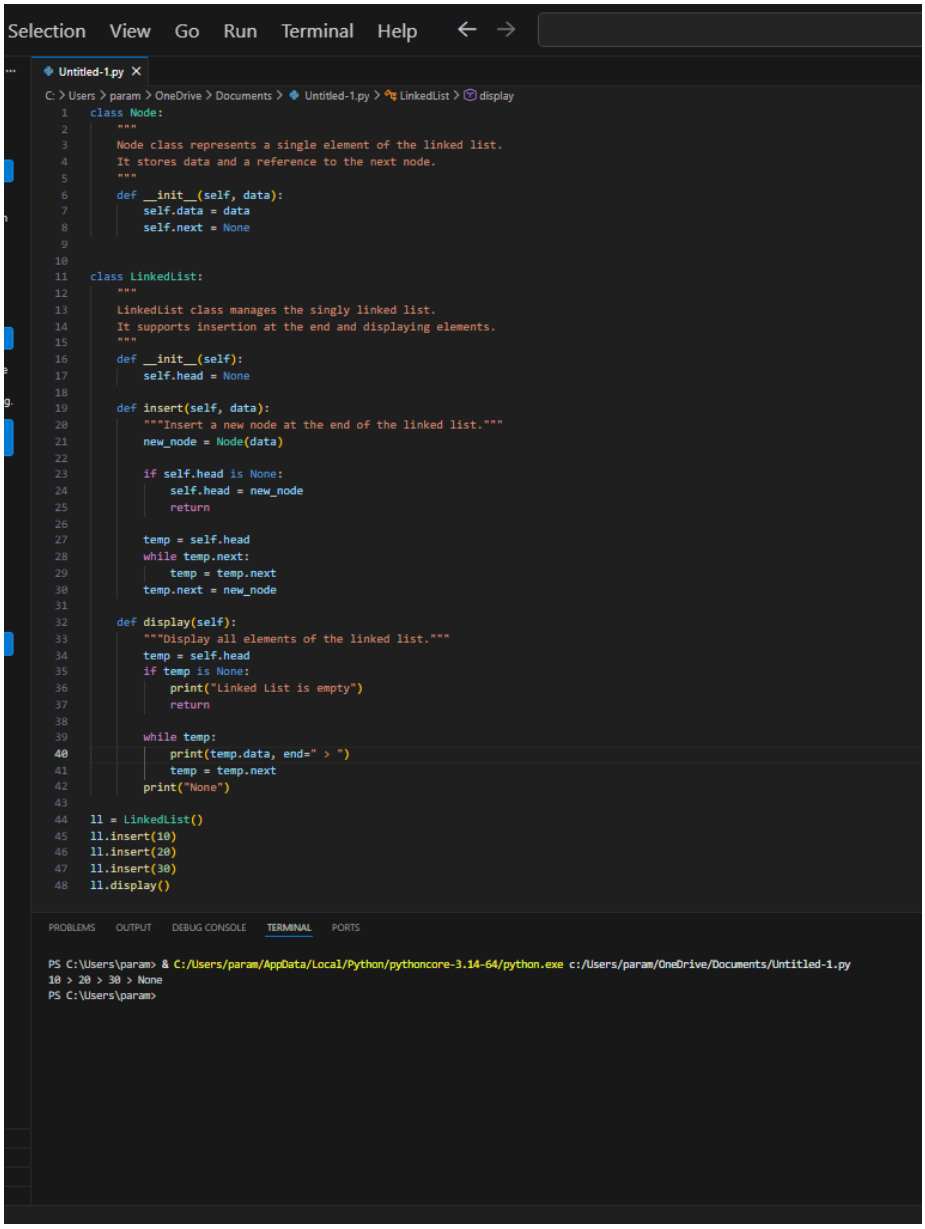
Generate a Singly Linked List in Python with Node and LinkedList classes. Implement insert and display methods with clear documentation.

Code:

```
class Node:
    """
    Node class represents a single element of the linked list.
    It stores data and a reference to the next node.
    """
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    """
    LinkedList class manages the singly linked list.
    It supports insertion at the end and displaying elements.
    """
    def __init__(self):
        self.head = None

    def insert(self, data):
        """Insert a new node at the end of the linked list."""
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return

        temp = self.head
        while temp.next:
```

```
            temp = temp.next
        temp.next = new_node

    def display(self):
        """Display all elements of the linked list."""
        temp = self.head
        if temp is None:
            print("Linked List is empty")
            return

        while temp:
            print(temp.data, end=" > ")
            temp = temp.next
        print("None")

ll = LinkedList()
ll.insert(10)
ll.insert(20)
ll.insert(30)
ll.display()
```

output:



Explaination :

A singly linked list stores elements in nodes where each node points to the next node. The insert() method adds elements at the end, and display() prints the list sequentially.

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.

Prompt used :

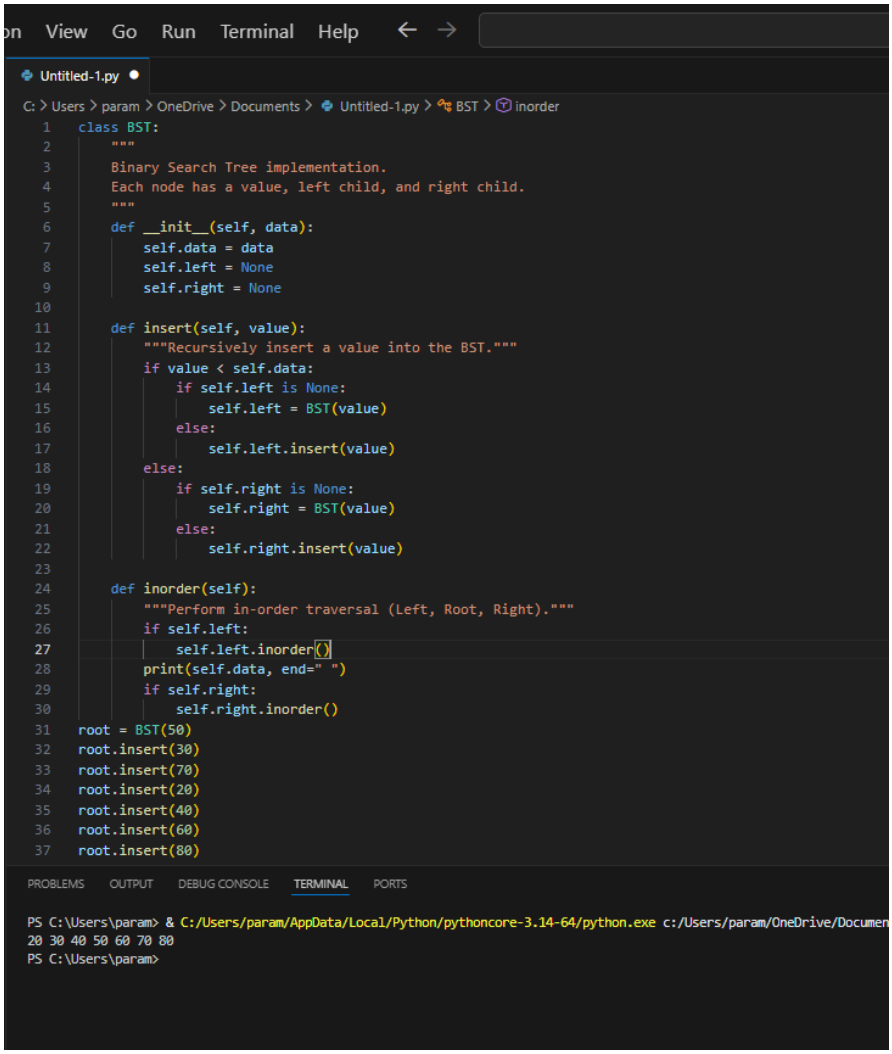
Generate a Binary Search Tree (BST) in Python with recursive insert and in-order traversal methods.

Code:

```
class BST:
    """
    Binary Search Tree implementation.
    Each node has a value, left child, and right child.
    """
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def insert(self, value):
        """Recursively insert a value into the BST."""
        if value < self.data:
            if self.left is None:
                self.left = BST(value)
            else:
                self.left.insert(value)
        else:
            if self.right is None:
                self.right = BST(value)
            else:
```

```
            self.right.insert(value)

    def inorder(self):
        """Perform in-order traversal (Left, Root, Right)."""
        if self.left:
            self.left.inorder()
        print(self.data, end=" ")
        if self.right:
            self.right.inorder()
root = BST(50)
root.insert(30)
root.insert(70)
root.insert(20)
root.insert(40)
root.insert(60)
root.insert(80)
root.inorder()
```

output:



Explaination : A Binary Search Tree stores smaller values in the left subtree and larger values in the right subtree.
In-order traversal prints elements in **sorted order** using recursion.

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.

Prompt used:

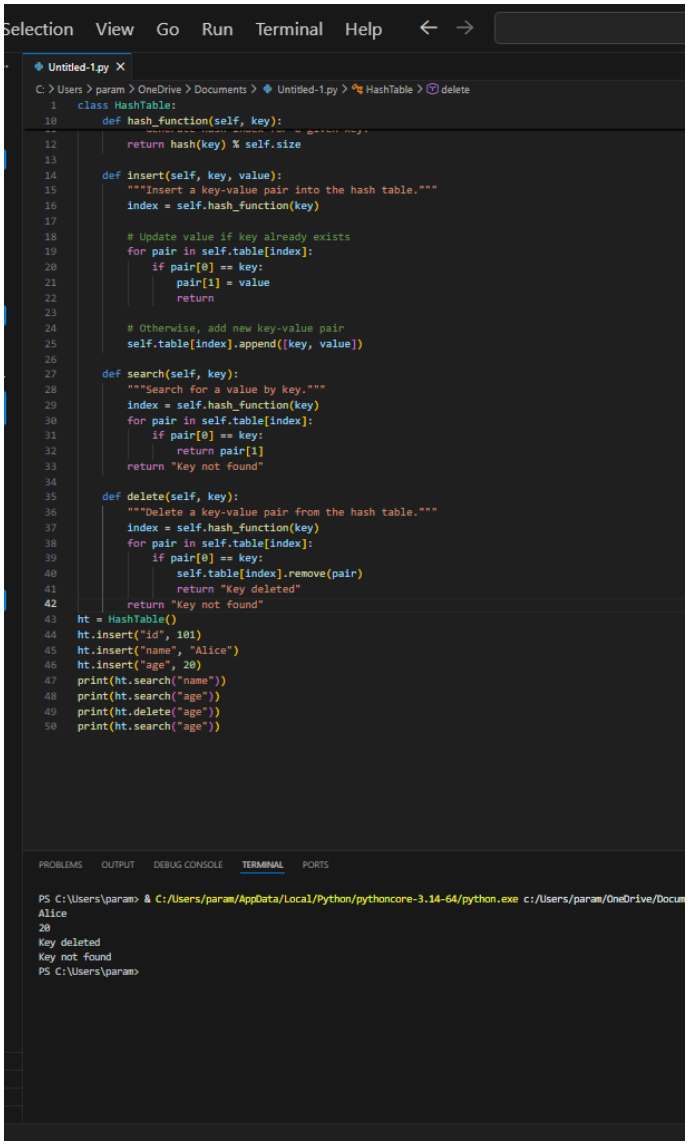
Generate *a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods with proper comments.*

Code:

```
class HashTable:
    """
    Hash Table implementation using chaining to handle collisions.
    Each index of the table stores a list of key-value pairs.
    """
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def hash_function(self, key):
        """Generate hash index for a given key."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
        index = self.hash_function(key)

        # Update value if key already exists
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return

        # Otherwise, add new key-value pair
        self.table[index].append([key, value])
```

```
    def search(self, key):
        """Search for a value by key."""
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return "Key not found"

    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                self.table[index].remove(pair)
                return "Key deleted"
        return "Key not found"
ht = HashTable()
ht.insert("id", 101)
ht.insert("name", "Alice")
ht.insert("age", 20)
print(ht.search("name"))
print(ht.search("age"))
print(ht.delete("age"))
print(ht.search("age"))
```

output:



Explaination :

A hash table stores data using key-value pairs and computes an index using a hash function. **Chaining** handles collisions by storing multiple pairs in a list at the same index.

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.

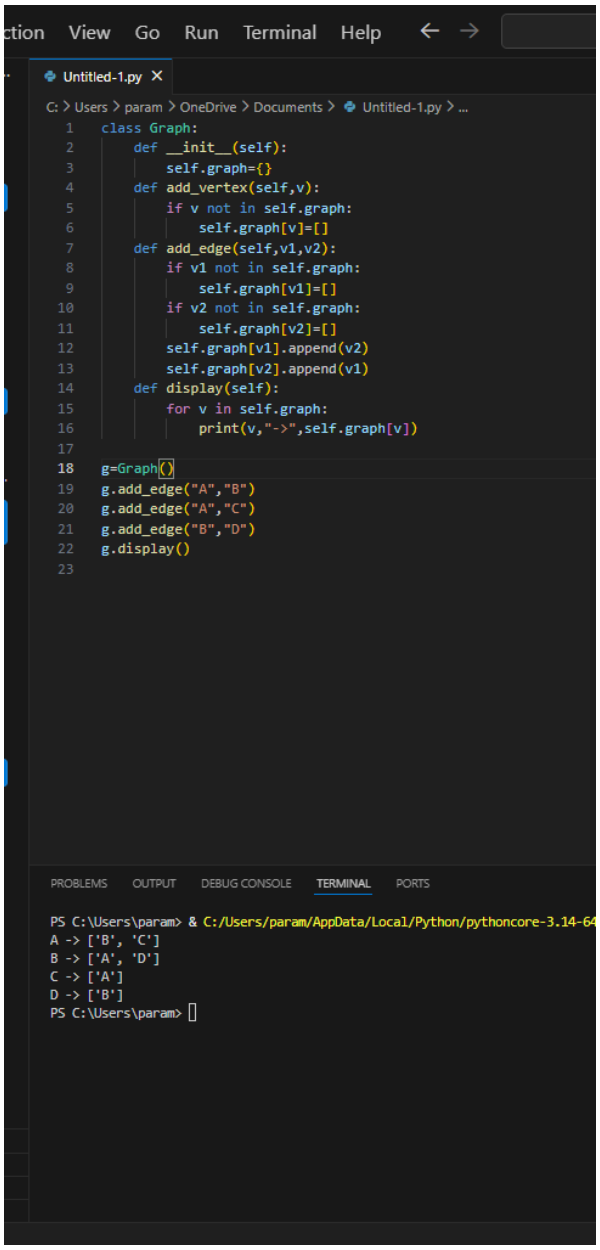
Prompt used:

Generate a Graph in Python using an adjacency list. Include methods to add vertices, add edges, and display the graph connections.

Code ;

```
class Graph:
  def __init__(self):
    self.graph={}
  def add_vertex(self,v):
    if v not in self.graph:
      self.graph[v]=[]
  def add_edge(self,v1,v2):
    if v1 not in self.graph:
      self.graph[v1]=[]
    if v2 not in self.graph:
      self.graph[v2]=[]
    self.graph[v1].append(v2)
    self.graph[v2].append(v1)
  def display(self):
    for v in self.graph:
      print(v,"->",self.graph[v])


g=Graph()
g.add_edge("A","B")
g.add_edge("A","C")
g.add_edge("B","D")
g.display()
```

output:



Explaination:

A graph is represented using an **adjacency list**, where each vertex stores its connected vertices.
This structure is memory-efficient and commonly used for sparse graphs.

ask 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.

Prompt used:

Generate a Priority Queue in Python using the heapq module with enqueue, dequeue, and display methods.

Code :

```
import heapq


class PriorityQueue:
    def __init__(self):
        self.pq=[]


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


    def dequeue(self):
        if not self.pq:
            return "Queue is empty"
        return heapq.heappop(self.pq)


    def display(self):
        print(self.pq)
p=PriorityQueue()
p.enqueue(3,"Low")
p.enqueue(1,"High")
p.enqueue(2,"Medium")
```

print(p.dequeue())

p.display()

output;

Explaination :

A priority queue processes elements based on priority instead of insertion order. The heapq module maintains a min-heap, so the lowest priority value is removed first.

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.

Prompt used:

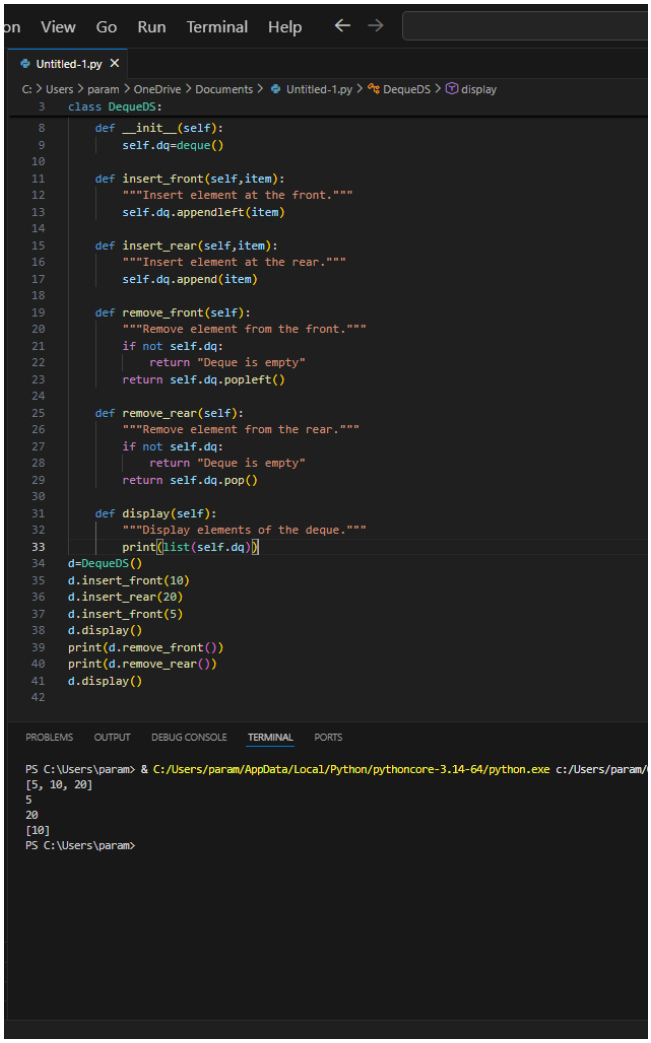
Generate a double-ended queue in Python using collections.deque. Support insert and remove operations from both ends with proper docstrings.

Code :

```
from collections import deque

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

    def insert_front(self,item):
        """Insert element at the front."""
        self.dq.appendleft(item)

    def insert_rear(self,item):
        """Insert element at the rear."""
        self.dq.append(item)

    def remove_front(self):
        """Remove element from the front."""
        if not self.dq:
            return "Deque is empty"
        return self.dq.popleft()
```

```
    def remove_rear(self):
        """Remove element from the rear."""
        if not self.dq:
            return "Deque is empty"
        return self.dq.pop()

    def display(self):
        """Display elements of the deque."""
        print(list(self.dq))
d=DequeDS()
d.insert_front(10)
d.insert_rear(20)
d.insert_front(5)
d.display()
print(d.remove_front())
print(d.remove_rear())
d.display()
```

output:



Explaination :

A deque allows insertion and deletion from both ends efficiently. Python's collections.deque provides fast O(1) operations at both ends.

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.

Prompt used :

Generate a cafeteria order system using a queue in Python where students are served in FIFO order.

Code :

```
class CafeteriaQueue:
    def __init__(self):
        self.q=[]
    def place_order(self,s):
        self.q.append(s)
        print(s,"order placed")
    def serve_order(self):
```

```
        if not self.q:
            print("No orders")
            return
        print(self.q.pop(0),"order served")
    def display(self):
        print(self.q)

c=CafeteriaQueue()
c.place_order("Amit")
c.place_order("Riya")
c.place_order("Suresh")
c.display()
c.serve_order()
c.display()
```

output:

Explaination :

The cafeteria system uses a **Queue** to ensure fairness by serving students in arrival order. The place_order() method enqueues students, while serve_order() dequeues the first order, following FIFO.

Task Description #10: Smart E-Commerce Platform – Data Structure Challenge

An e-commerce company wants to build a Smart Online Shopping System with:

1. Shopping Cart Management – Add and remove products dynamically.

2. Order Processing System – Orders processed in the order they are placed.

3. Top-Selling Products Tracker – Products ranked by sales count.

4. Product Search Engine – Fast lookup of products using product ID.

5. Delivery Route Planning – Connect warehouses and delivery locations.

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.

Prompt used :

Generate an order processing system using a queue in Python where orders are processed in FIFO order.

Code :

```
class OrderQueue:
  def __init__(self):
    self.q=[]
  def place(self,o):
    self.q.append(o)
    print(o,"placed")
  def process(self):
```

```
        if not self.q:
            print("No orders")
            return
        print(self.q.pop(0),"processed")
    def display(self):
        print(self.q)

o=OrderQueue()
o.place(101)
o.place(102)
o.place(103)
o.display()
o.process()
o.display()
```

output :

Explaination :

The order processing system uses a **Queue** to ensure orders are handled in the order they arrive.
This guarantees fairness and prevents newer orders from skipping earlier ones.