AI Assisted Coding

NAME :THOTA SHIRISHA
HTNO:2403A52077
BATCH NO:04
ASSIGNMENT :11.2

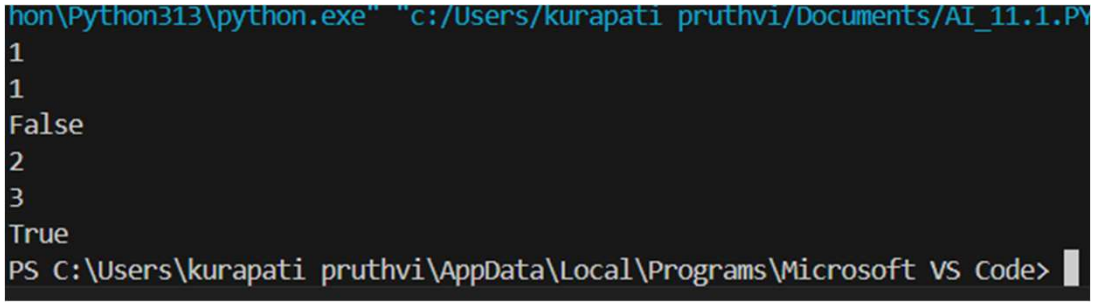
TASK 1: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.
Sample Input Code:
class Stack:
pass
Expected Output:
• A functional stack implementation with all required methods and docstrings.

PROMPT : 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 Queue:
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []
    def enqueue(self, item):
        """Add an item to the end of the queue."""
        self.items.append(item)

    def dequeue(self):
        """Remove and return the front item of the queue. Raises IndexError if empty."""
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)

    def peek(self):
        """Return the front item without removing it. Raises IndexError if empty."""
        if self.is_empty():
            raise IndexError("Peek from empty queue")
        return self.items[0]
    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.items) == 0

    def __str__(self):
        """Return a string representation of the queue."""
        return f"Queue({self.items})"
# Test code
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.peek())     # Output: 1
print(q.dequeue())  # Output: 1
print(q.is_empty()) # Output: False
print(q.dequeue())  # Output: 2
print(q.dequeue())  # Output: 3
print(q.is_empty()) # Output: True
```

OUTPUT:



OBSERVATION:

1.The Queue class is defined, but the constructor must be __init__ (not init) to initialize self.items = [].
2.The enqueue() method appends items to the end of the list, simulating a queue.
3.The dequeue() method removes and returns the front item using pop(0), raising an error if empty.
4.The peek() method returns the front item without removing it, with a check for empty queue.
5.An is_empty() method is **missing** — it should return len(self.items) == 0.
6.Items 1, 2, 3 are added; then peek() shows 1, and dequeue() removes items in order.
7.Finally, is_empty() returns True once all items are removed, confirming the queue is empty.

Task2: 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:

Use AI to implement a Queue using Python lists.

Sample Input Code:

class Queue:

pass

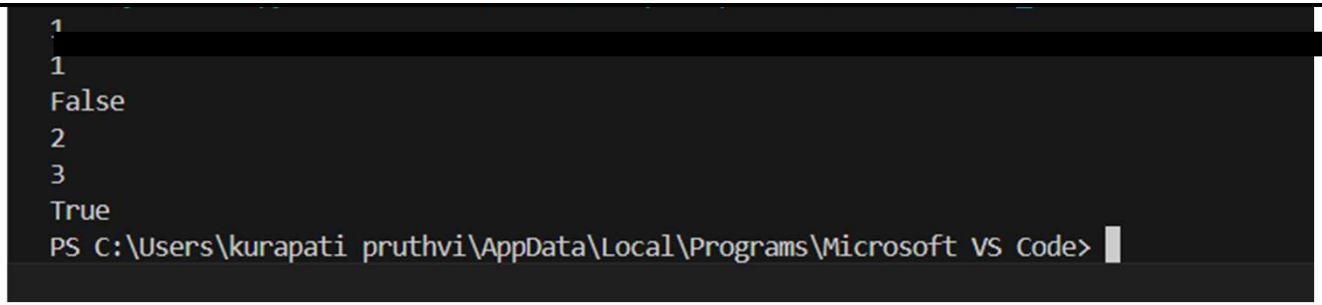
Expected Output:

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

CODE:

```python
class Queue:
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []
    def enqueue(self, item):
        """Add an item to the end of the queue."""
        self.items.append(item)
    def dequeue(self):
        """Remove and return the front item of the queue. Raises IndexError if empty."""
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)
    def peek(self):
        """Return the front item without removing it. Raises IndexError if empty."""
        if self.is_empty():
            raise IndexError("Peek from empty queue")
        return self.items[0]

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

    def __str__(self):
        """Return a string representation of the queue."""
        return f"Queue({self.items})"
# Test the Queue
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.peek())     # Output: 1
print(q.dequeue())  # Output: 1
print(q.is_empty()) # Output: False
print(q.dequeue())  # Output: 2
print(q.dequeue())  # Output: 3
print(q.is_empty()) # Output: True
```

OUTPUT:



OBSERVATION:

Initialization (__init__):
The queue is initialized with an empty list self.items = [].
⚠️ However, your method is incorrectly written as _init_; it should be **__init__**.
2.Enqueue Operation:
The enqueue() method adds elements (1, 2, 3) to
The peek() method returns the **first element** (1) without removing it — this confirms the front of the queue.
4.Dequeue Operation:
The dequeue() method removes and returns the **front item** using pop(0), maintaining **FIFO order**.
5.Empty Check:
is_empty() checks if the queue is empty by comparing the length of items with 0.
2.Final State: the **end** of the queue using append().
3.Peek Operation:
After all dequeue() operations, the queue becomes empty, and is_empty() returns True.

TASK 3:

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 docuementation.

PROMPT:

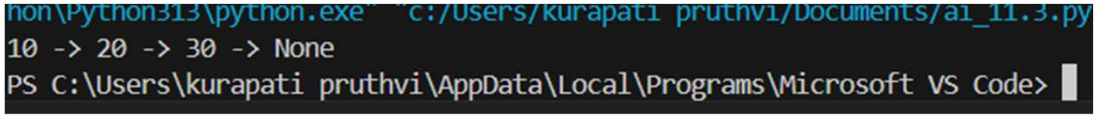
Use AI to generate a Singly Linked List with insert and display methods.
Sample Input Code:
class Node:
pass
class LinkedList:
pass
Expected Output:
• A working linked list implementation with clear method docuementation

CODE:

Users > kurapati pruthvi > Documents > ai_11.3.py > ...

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, data):
        """Insert a new node at the end of the list."""
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def display(self):
        """Display the contents of the linked list."""
        current = self.head
        if current is None:
            print("Linked List is empty.")
            return
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Test the LinkedList
ll = LinkedList()
ll.insert(10)
ll.insert(20)
ll.insert(30)
ll.display()  # Output: 10 -> 20 -> 30 -> None
```

OUTPUT:



OBSERVATION:

1.A Node class is defined to represent each element in the linked list, storing data and a reference to the next node.

2.A LinkedList class is created with an initial empty list, where the head is set to None.

3.The insert method adds a new node with given data at the end of the list.

4.The display method prints all elements in the list from head to tail in order.

5.The list is tested by inserting three values and displaying the output, which prints: 10 -> 20 -> 30 -> None.

TASK 4:

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:

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

Sample Input Code:

```
class BST:
pass
```

Expected Output:

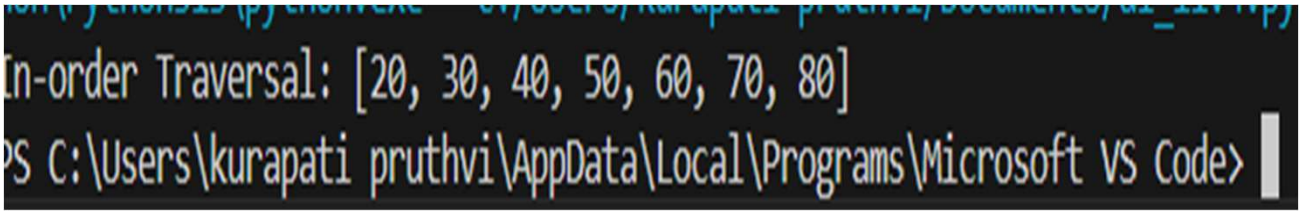
- BST implementation with recursive insert and traversal methods.

CODE :

```
class Node:
    def __init__(self, data):  # Fixed __init__
        self.data = data
        self.left = None
        self.right = None
class BST:
    def __init__(self):  # Fixed __init__
        self.root = None
    def insert(self, data):
        """Insert a new node into the BST."""
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert_recursive(self.root, data)
    def _insert_recursive(self, current, data):
        if data < current.data:
            if current.left is None:
                current.left = Node(data)
            else:
                self._insert_recursive(current.left, data)
        elif data > current.data:
            if current.right is None:
                current.right = Node(data)
            else:
                self._insert_recursive(current.right, data)
        # If data == current.data, we do nothing (no duplicates)
    def in_order_traversal(self):
        """Perform in-order traversal and return list of values."""
        result = []
        self._in_order_recursive(self.root, result)
        return result
    def _in_order_recursive(self, node, result):
        if node:
            self._in_order_recursive(node.left, result)
            result.append(node.data)
            self._in_order_recursive(node.right, result)
# --- Test the BST ---
```

```
# --- Test the BST ---
tree = BST()
tree.insert(50)
tree.insert(30)
tree.insert(70)
tree.insert(20)
tree.insert(40)
tree.insert(60)
tree.insert(80)
print("In-order Traversal:", tree.in_order_traversal())
```

OUTPUT:



OBSERVARION:

1.Node Class Definition:
A Node object represents each node in the binary search tree (BST). It holds the node's data and pointers to its left and right children.

2.BST Initialization:
The BST class starts with an empty tree (self.root = None). This sets up the structure to store and manage nodes.

3.Insertion Logic:
The insert() method adds new data to the BST. It calls a recursive helper to correctly place the new value based on BST rules (left < root < right).

4.Avoiding Duplicates:
During insertion, if the value Already exists in the tree, the code does nothing. This ensures the tree contains only unique values.

5.In-order Traversal:
The in_order_traversal() method performs a left-root-right traversal, collecting values in ascending order.

6.Testing the Tree:
A sample tree is built using .insert() calls, and in_order_traversal() is used to print the values in sorted order:
Output: [20, 30, 40, 50, 60, 70, 80].

TASK 5:

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: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.

CODE:

```python
class HashTable:
    def __init__(self, size=10):  # Corrected __init__
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash_function(self, key):
        """Simple hash function using Python's built-in hash and modulo."""
        return hash(key) % self.size
    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
        index = self._hash_function(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        self.table[index].append((key, value))  # Insert new key-value pair
    def search(self, key):
        """Search for a value by key. Returns the value or None if not found."""
        index = self._hash_function(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None
    def delete(self, key):
        """Delete a key-value pair from the hash table. Returns True if deleted, else False."""
        index = self._hash_function(key)
        for i, (k, _) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                return True
        return False
    def display(self):
        """Display the contents of the hash table."""
        for i, bucket in enumerate(self.table):
            print(f"Bucket {i}: {bucket}")
```

```
class HashTable:
    def delete(self, key):
        """Delete a key-value pair from th
        index = self._hash_function(key)
        for i, (k, _) in enumerate(self.ta
            if k == key:
                del self.table[index][i]
                return True
        return False
    def display(self):
        """Display the contents of the has
        for i, bucket in enumerate(self.ta
            print(f"Bucket {i}: {bucket}")
# --- Test the HashTable ---
ht = HashTable()
ht.insert("apple", 100)
ht.insert("banana", 200)
ht.insert("orange", 300)
print(ht.search("banana"))   # Output: 200
ht.delete("apple")
ht.display()
```

OUTPUT:

```
200
Bucket 0: [('banana', 200)]
Bucket 1: []
Bucket 2: []
Bucket 3: []
Bucket 4: []
Bucket 5: []
Bucket 6: []
Bucket 7: []
Bucket 8: []
Bucket 9: [('orange', 300)]
PS C:\Users\kurapati pruthvi\AppData\Local\Programs\Microsoft VS Code>
```

OBSERVATION:

1.Initiaialization:

2.Hash Function

The HashTable class creates a list of empty buckets (default 10), where each bucket is a list used to handle collisions via chaining.

3.:

The _hash_function(key) uses Python's built-in hash() function combined with modulo operation to determine which bucket a key belongs to.

3.Insertion:

The insert() method calculates the index using the hash function. If the key exists in the bucket, it updates the value; otherwise, it adds the new key-value pair.

4.Search:

The search() method hashes the key to find the correct bucket, then linearly searches that bucket for the key and returns the associated value if found.

5.Deletion:

The delete() method hashes the key, finds the correct bucket, and removes the key-value pair if the key exists, returning True; otherwise, it returns False.

6.Testing and Display:

Keys are inserted, searched, and deleted. After deleting "apple", the display() method prints the contents of all buckets, showing current key-value pairs in the table.

TASK 6:

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:

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):  # Corrected constructor
        self.adj_list = {}

    def add_edge(self, u, v):
        """Add an edge from vertex u to vertex v (undirected graph
        if u not in self.adj_list:
            self.adj_list[u] = []
        if v not in self.adj_list:
            self.adj_list[v] = []
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)  # For undirected graph

    def display(self):
        """Display the adjacency list of the graph."""
        for vertex in self.adj_list:
            print(f"{vertex} -> {self.adj_list[vertex]}")

# --- Test the Graph ---
g = Graph()
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "D")
g.add_edge("C", "D")
g.display()
```

OUTPUT:

```
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A', 'D']
D -> ['B', 'C']
PS C:\Users\kurapati pruthvi\AppData\Local\Programs\Microsoft VS Code>
```

OBSERVATION :

1.INSTILIZATION:

2.Adding Edges:**Graph** :

The Graph class uses a dictionary adj_list to represent the adjacency list, where each key is a vertex and its value is a list of connected vertic

The add_edge(u, v) method ensures both vertices u and v exist in the adjacency list, then appends each to the other's list to represent an undirected edge.

3.Undirected Behavior:

Since edges are added in both directions (u → v and v → u), the graph is undirected—meaning connectior are mutual.

4.Displaying the Graph:

The display() method prints each vertex and its list of connected vertices, showing the structure of the graph

5.Testing the Graph:

Four edges are added between vertices A, B, C, and D, forming a small undirected graph. The output shows how each vertex is connected.

Example Output:

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

TASK 7:

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.

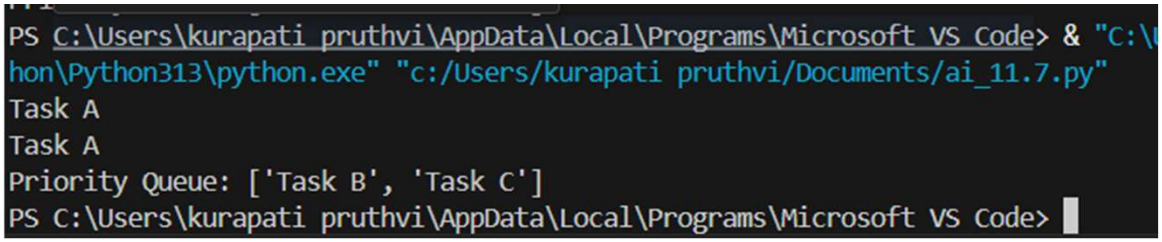
PROMPT:

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):  # Fixed constructor
        self.heap = []
    def insert(self, priority, item):
        """Insert an item with a given priority."""
        heapq.heappush(self.heap, (priority, item))

    def remove(self):
        """Remove and return the item with the highest priority (lowest number)."""
        if self.is_empty():
            raise IndexError("Remove from empty priority queue")
        return heapq.heappop(self.heap)[1]

    def peek(self):
        """Return the item with the highest priority without removing it."""
        if self.is_empty():
            raise IndexError("Peek from empty priority queue")
        return self.heap[0][1]

    def is_empty(self):
        """Check if the priority queue is empty."""
        return len(self.heap) == 0

    def display(self):
        """Display the contents of the priority queue."""
        print("Priority Queue:", [item for _, item in self.heap])
# --- Test the Priority Queue ---
pq = PriorityQueue()
pq.insert(3, "Task C")
pq.insert(1, "Task A")
pq.insert(2, "Task B")

print(pq.peek())      # Output: Task A
print(pq.remove())    # Output: Task A
pq.display()          # Output: Priority Queue: ['Task B', 'Task C']
```

OUTPUT:



OBSERVATION:

1.Initialization

2.The PriorityQueue class uses a list called heap and Python's built-in heapq module to maintain a min-heap structure, where the smallest priority value comes first.

2.Insertion:
The insert(priority, item) method adds a tuple (priority, item) to the heap. The heap automatically keeps the item with the lowest priority number at the top.

3.Removal:
The remove() method pops the item with the highest priority (i.e., lowest priority number) from the heap. It raises an error if the queue is empty.

4.Peeking:
The peek() method returns (but does not remove) the item with the highest priority, helping check the next task without modifying the queue.

5.Testing and Display:
Three tasks with different priorities are added. peek() returns "Task A" (priority 1), remove() removes it, and display() shows the remaining tasks:
Output: Priority Queue: ['Task B', 'Task C'].

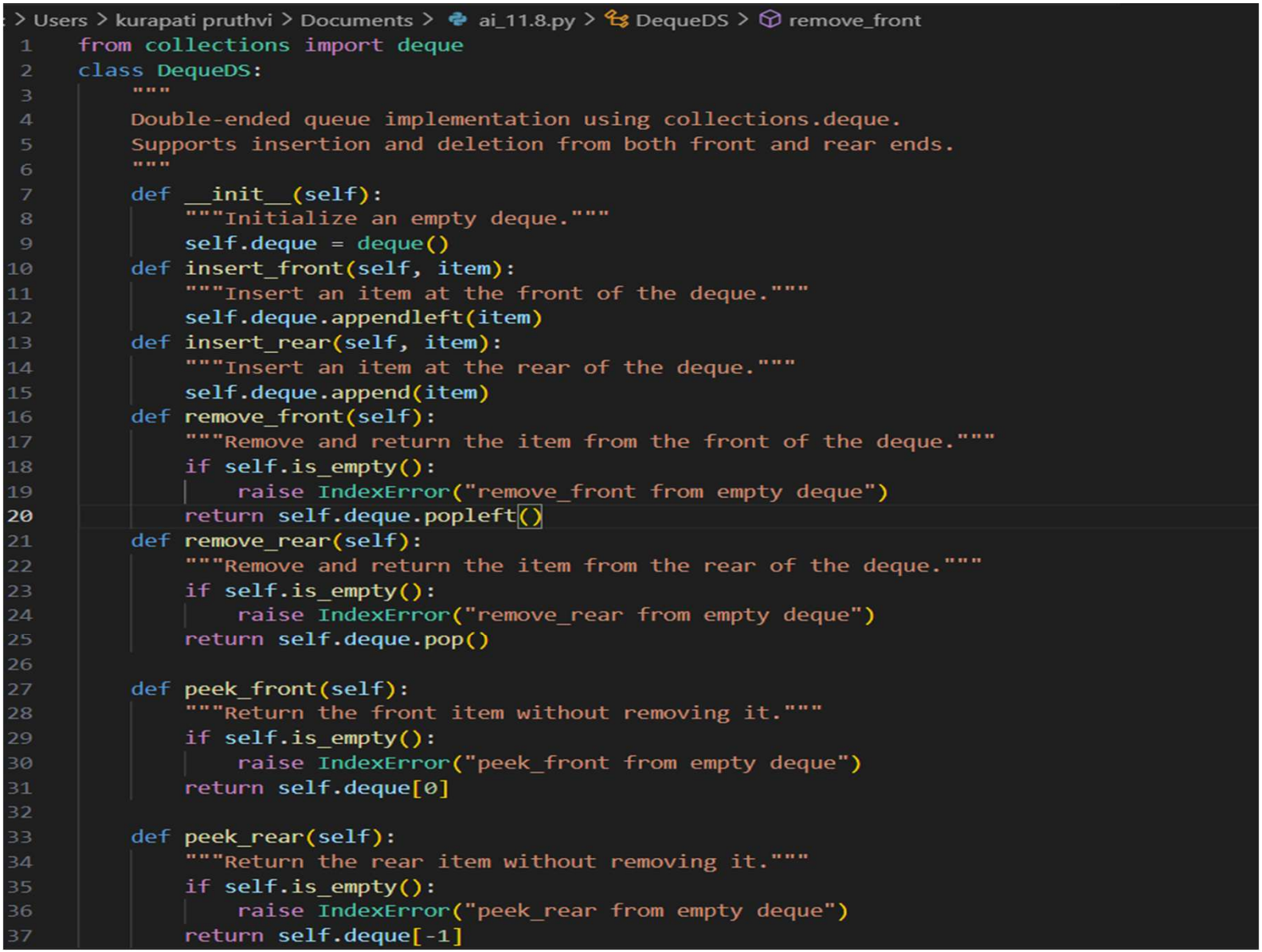
TASK 8:

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:

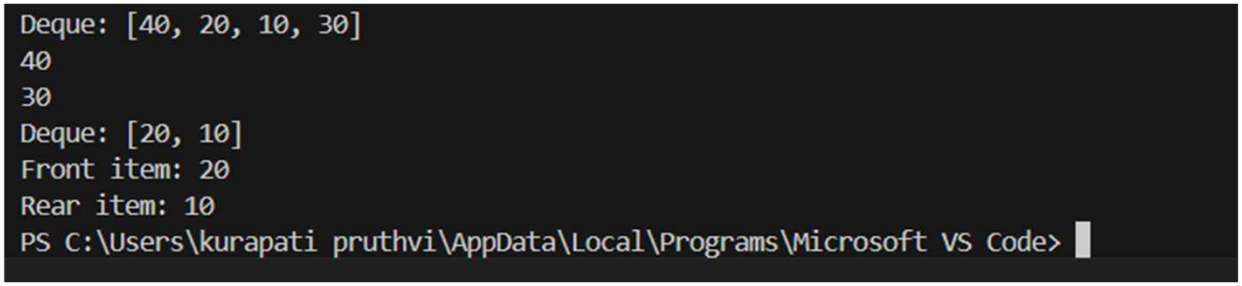
Use AI to implement a double-ended queue using collections.deque.
Sample Input Code:
class DequeDS:
pass
Expected Output:
• Insert and remove from both ends with docstrings.

CODE :


> Users > kurapati pruthvi > Documents > ai_11.8.py > DequeDS > remove_front
from collections import deque
class DequeDS:
"""
Double-ended queue implementation using collections.deque.
Supports insertion and deletion from both front and rear ends.
"""
def __init__(self):
"""Initialize an empty deque."""
self.deque = deque()
def insert_front(self, item):
"""Insert an item at the front of the deque."""
self.deque.appendleft(item)
def insert_rear(self, item):
"""Insert an item at the rear of the deque."""
self.deque.append(item)
def remove_front(self):
"""Remove and return the item from the front of the deque."""
if self.is_empty():
raise IndexError("remove_front from empty deque")
return self.deque.popleft()
def remove_rear(self):
"""Remove and return the item from the rear of the deque."""
if self.is_empty():
raise IndexError("remove_rear from empty deque")
return self.deque.pop()
def peek_front(self):
"""Return the front item without removing it."""
if self.is_empty():
raise IndexError("peek_front from empty deque")
return self.deque[0]
def peek_rear(self):
"""Return the rear item without removing it."""
if self.is_empty():
raise IndexError("peek_rear from empty deque")
return self.deque[-1]

```
        return self.deque[-1]

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

    def size(self):
        """Return the number of elements in the deque."""
        return len(self.deque)

    def display(self):
        """Display the current state of the deque."""
        print("Deque:", list(self.deque))

# --- Sample Usage ---
if __name__ == "__main__":
    dq = DequeDS()
    dq.insert_rear(10)
    dq.insert_front(20)
    dq.insert_rear(30)
    dq.insert_front(40)

    dq.display()                # Output: Deque: [40, 20, 10, 30]

    print(dq.remove_front())  # Output: 40
    print(dq.remove_rear())   # Output: 30

    dq.display()                # Output: Deque: [20, 10]

    print("Front item:", dq.peek_front())  # Output: 20
    print("Rear item:", dq.peek_rear())    # Output: 10

```

OUTPUT:



OBSERVATION:

Here's a **short 5-step explanation** of the DequeDS code:

1.Initialization:
Uses Python's collections.deque to create a double-ended queue that supports fast insertions and deletions from both ends.

2.Insertions:
insert_front() adds an item to the front; insert_rear() adds to the rear of the deque.

3.Removals:
remove_front() and remove_rear() remove items from the front and rear respectively, with error checks for empty deque.

4.Peeking and Utilities:
peek_front() and peek_rear() return the front and rear items without removal. is_empty() and size() provide basic status checks.

5.Testing and Display:
Sample usage shows inserting, removing, and peeking, with outputs reflecting the current state of the deque at each step.

Task9: Use AI to generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.
Sample Input Code:
No code, prompt AI for a data structure comparison table
Expected Output:
• A markdown table with structure names, operations, and complexities.
Task Description #10 Real-Time Application Challenge – Choose the Right Data Structure
Scenario:

PROMPT:

Use AI to generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.
Sample Input Code:
No code, prompt AI for a data structure comparison table
Expected Output:
• A markdown table with structure names, operations, and complexities.
Task Description #10 Real-Time Application Challenge – Choose the Right Data Structure
Scenario:

CODE:

C: > Users > kurapati pruthvi > Documents > ai_11.9.py > ...

```python
# Data Structure Comparison Table Example
def print_data_structure_comparison():
    table = [
        ["Data Structure", "Insert/Push/Enqueue", "Delete/Pop/Dequeue", "Search/Access", "Peek/Front/End"],
        ["Stack (List)", "O(1)", "O(1)", "O(n)", "O(1)"],
        ["Queue (List)", "O(1) (enqueue)", "O(n) (dequeue)", "O(n)", "O(1)"],
        ["Queue (Deque)", "O(1)", "O(1)", "O(n)", "O(1)"],
        ["Singly Linked List", "O(1) (at head)", "O(1) (at head)", "O(n)", "O(1) (head)"],
        ["Doubly Linked List", "O(1) (at ends)", "O(1) (at ends)", "O(n)", "O(1) (ends)"],
    ]
    for row in table:
        print(" | ".join(row))
# Stack Example
class Stack:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop() if self.items else None
    def peek(self):
        return self.items[-1] if self.items else None
    def is_empty(self):
        return len(self.items) == 0
# Queue Example
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        return self.items.pop(0) if self.items else None
    def peek(self):
        return self.items[0] if self.items else None
    def is_empty(self):
        return len(self.items) == 0
# Linked List Example
```

```python
# Linked List Example
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def insert(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
        print()
if __name__ == "__main__":
    print("Comparison Table of Data Structures and Time Complexities:")
    print_data_structure_comparison()
    print("\nStack Example:")
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print("Stack after pushes:", stack.items)
    print("Stack pop:", stack.pop())
    print("Stack peek:", stack.peek())
    print("Is stack empty?", stack.is_empty())
    print("\nQueue Example:")
```

```
72      print("\nQueue Example:")
73      queue = Queue()
74      queue.enqueue(1)
75      queue.enqueue(2)
76      queue.enqueue(3)
77      print("Queue after enqueues:", queue.items)
78      print("Queue dequeue:", queue.dequeue())
79      print("Queue peek:", queue.peek())
80      print("Is queue empty?", queue.is_empty())
81      print("\nLinked List Example:")
82      ll = LinkedList()
83      ll.insert(1)
84      ll.insert(2)
85      ll.insert(3)
86      print("Linked List elements:", end=" ")
87      ll.display()
88
```

OUTPUT:

```
Comparison Table of Data Structures and Time Complexities:
Data Structure | Insert/Push/Enqueue | Delete/Pop/Dequeue | Search/Access | Peek/Front/End
Stack (List) | O(1) | O(1) | O(n) | O(1)
Queue (List) | O(1) (enqueue) | O(n) (dequeue) | O(n) | O(1)
Queue (Deque) | O(1) | O(1) | O(n) | O(1)
Singly Linked List | O(1) (at head) | O(1) (at head) | O(n) | O(1) (head)
Doubly Linked List | O(1) (at ends) | O(1) (at ends) | O(n) | O(1) (ends)

Stack Example:
Stack after pushes: [1, 2, 3]
Stack pop: 3
Stack peek: 2
Is stack empty? False

Queue Example:
Queue after enqueues: [1, 2, 3]
Queue dequeue: 1
Queue peek: 2
Is queue empty? False

Linked List Example:
Linked List elements: 1 2 3
```

OBSERVATION:

`def print_data_structure_comparison():`

- Defines a function that prints a formatted comparison table.

`table = [...]`

- A list of lists, where each inner list represents a row in the table.
- The first row is the header: column titles like "Insert", "Delete", etc.
- Each subsequent row compares a specific data structure.

`" | ".join(row)`

- Joins each element in the row with `" | "` to mimic a table format.
- This makes the output readable and aligned like a markdown-style table.

TASK 10:

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.

Deliverables (For All Tasks)

1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.

PROMPT:

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.

Deliverables (For All Tasks)

1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.

CODE :

C: > Users > kurapati pruthvi > Documents > ai_11.10.py > ...

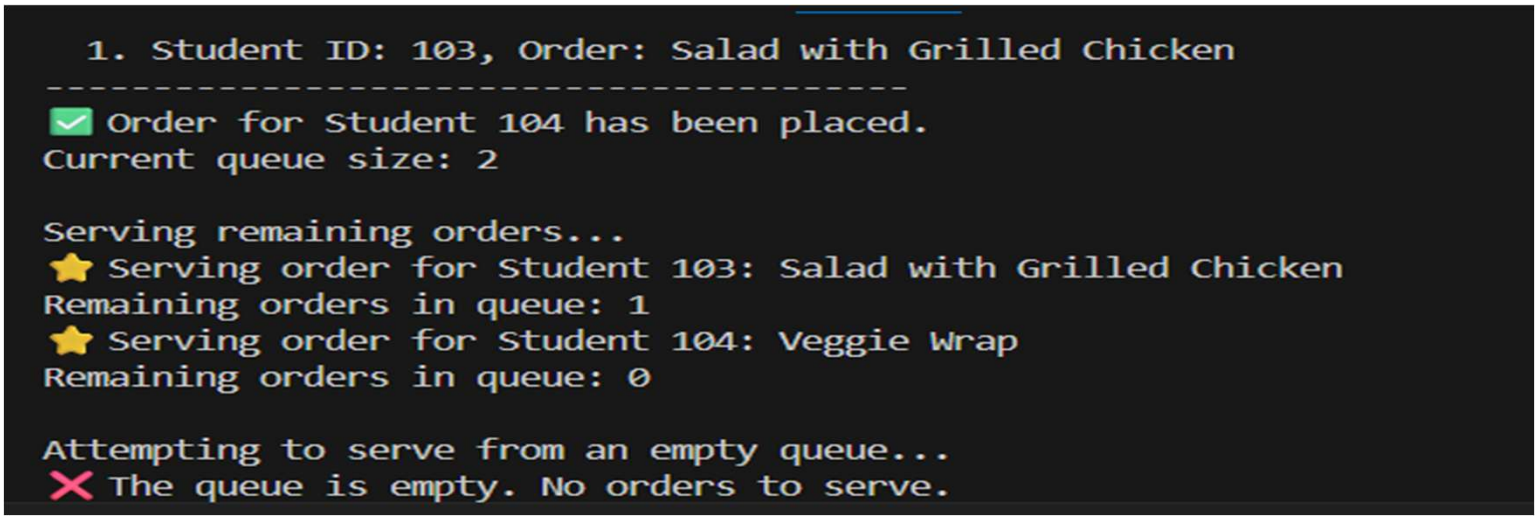
```python
import collections

class CafeteriaQueue:
    """
    Implements a cafeteria order queue using a deque for efficient
    First-In, First-Out (FIFO) operations.
    """
    def __init__(self):
        # Using a deque from the collections module for an efficient queue
        self.orders = collections.deque()
        print("Cafeteria Order Queue system initialized.")
        print("-" * 40)

    def add_order(self, student_id, order_details):
        """Adds a new order to the end of the queue."""
        self.orders.append({'student_id': student_id, 'order_details': order_details})
        print(f"✅ Order for Student {student_id} has been placed.")
        print(f"Current queue size: {len(self.orders)}")

    def serve_next_order(self):
        """Serves the next order from the front of the queue."""
        if not self.orders:
            print("❌ The queue is empty. No orders to serve.")
            return None

        served_order = self.orders.popleft()
        student_id = served_order['student_id']
        order_details = served_order['order_details']

        print(f"⭐ Serving order for Student {student_id}: {order_details}")
        print(f"Remaining orders in queue: {len(self.orders)}")
        return served_order

    def display_queue(self):
        """Displays all orders currently in the queue."""
        if not self.orders:
            print("📋 The queue is currently empty.")
```

Ln 80, Col

```
 3  class CafeteriaQueue:
34      def display_queue(self):
37              print("📄 The queue is currently empty.")
38              return
39
40          print("\n📄 Current Orders in Queue:")
41          for i, order in enumerate(self.orders):
42              print(f"  {i+1}. Student ID: {order['student_id']}, Order: {order['order_details']}")
43          print("-" * 40)
44  # Main program to demonstrate the Cafeteria Queue
45  if __name__ == "__main__":
46      cafeteria_queue = CafeteriaQueue()
47      # Students placing orders
48      cafeteria_queue.add_order(student_id=101, order_details="Pizza and Coke")
49      cafeteria_queue.add_order(student_id=102, order_details="Burger and Fries")
50      cafeteria_queue.add_order(student_id=103, order_details="Salad with Grilled Chicken")
51      # Display the current queue
52      cafeteria_queue.display_queue()
53      # Serving the next student
54      print("\nAttempting to serve next order...")
55      cafeteria_queue.serve_next_order()
56      # Serving the next student
57      print("\nAttempting to serve next order...")
58      cafeteria_queue.serve_next_order()
59      # Display the updated queue
60      cafeteria_queue.display_queue()
61      # Another student places an order
62      cafeteria_queue.add_order(student_id=104, order_details="Veggie Wrap")
63
64      # Serve the remaining orders
65      print("\nServing remaining orders...")
66      while cafeteria_queue.orders:
67          cafeteria_queue.serve_next_order()
68
69      # Try to serve from an empty queue
70      print("\nAttempting to serve from an empty queue...")
71      cafeteria_queue.serve_next_order()
```

Ln 60, C

OUTPUT:


1. Student ID: 103, Order: Salad with Grilled Chicken
Order for Student 104 has been placed.
Current queue size: 2
Serving remaining orders...
Serving order for Student 103: Salad with Grilled Chicken
Remaining orders in queue: 1
Serving order for Student 104: Veggie Wrap
Remaining orders in queue: 0
Attempting to serve from an empty queue...
The queue is empty. No orders to serve.

OBSERVATION:

Here's the explanation of the **CafeteriaQueue** code in **5 clear steps**:

1.Initialization:

The CafeteriaQueue class uses Python's collections.deque to create an efficient FIFO queue. The constructor initializes the queue and displays a setup message.

2.Adding Orders:

The add_order() method takes a student ID and order details, adds the order to the end of the queue, and prints a confirmation message along with the current queue size

3.Serving Orders:

The serve_next_order() method removes and returns the order at the front of the queue. If the queue is empty, it shows a warning message; otherwise, it prints the served order and the remaining queue size.

3.Displaying Queue:

3.The display_queue() method prints all current orders in the queue with their positions. If the queue is empty, it shows a message indicating that.

4.Program Execution:

The __main__ block simulates real usage: multiple students place orders, some orders are served, the queue is displayed, new orders are added, and remaining orders are served until the queue is empty. It ends by attempting to serve from an empty queue to show error handling.