AI ASSISTED CODING

NAME:J.KEERTHI PRIYA

ROLL NO:2403A510G4

ASSIGNMENT:11.1

#TASK-1:

PROMPT:

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. with example input and output.

CODE:

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

OUTPUT:

A black screen with blue text

AI-generated content may be incorrect.

OBSERVATION:

1. **Initialization**
   * A stack object is created using stack = Stack().
   * At this point, the stack is empty → stack.is\_empty() returns **True**.
2. **Push Operation**
   * Elements (5, 10, 15) are pushed one by one.
   * Internally, they are stored in a Python list → [5, 10, 15].
   * Each push() call adds the element to the **top of the stack** (end of the list).
3. **Peek Operation**
   * stack.peek() checks the last element without removing it.
   * Returns **15** (the top element) while keeping the stack as [5, 10, 15].
4. **Pop Operation**
   * stack.pop() removes and returns the top element.
   * Returns **15** and updates the stack to [5, 10].
5. **Peek after Pop**
   * stack.peek() now returns **10** (new top of the stack).
6. **Final Check**
   * stack.is\_empty() returns **False** since stack still has [5, 10].

#TASK-2

PROMPT:

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:

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

OUTPUT:

A computer screen with blue text

AI-generated content may be incorrect.

OBSERVATION :

**Initialization**

* A queue object is created using q = Queue().
* At this point, the queue is empty → q.is\_empty() returns **True**.

**Enqueue Operations**

* Elements "A", "B", "C" are enqueued in order.
* Internally stored as a Python list → ["A", "B", "C"].
* Each enqueue() adds the item to the **rear** of the queue.

**Peek Operation**

* q.peek() looks at the **front** element without removing it.
* Returns "A" (first inserted element) while keeping queue unchanged → ["A", "B", "C"].

**Dequeue Operation**

* q.dequeue() removes and returns the **front** element.
* Returns "A" and updates queue to ["B", "C"].

**Peek after Dequeue**

* Now q.peek() returns "B" since "A" was removed.
* Queue remains ["B", "C"].

**Size Check**

* q.size() returns **2** because two elements remain ("B", "C").

**Final State**

* q.is\_empty() returns **False** since the queue still contains items

#TASK-3:

PROMPT:

generate a Singly Linked List with insert and display methods.

Sample Input Code:

class Node:

pass

class LinkedList:

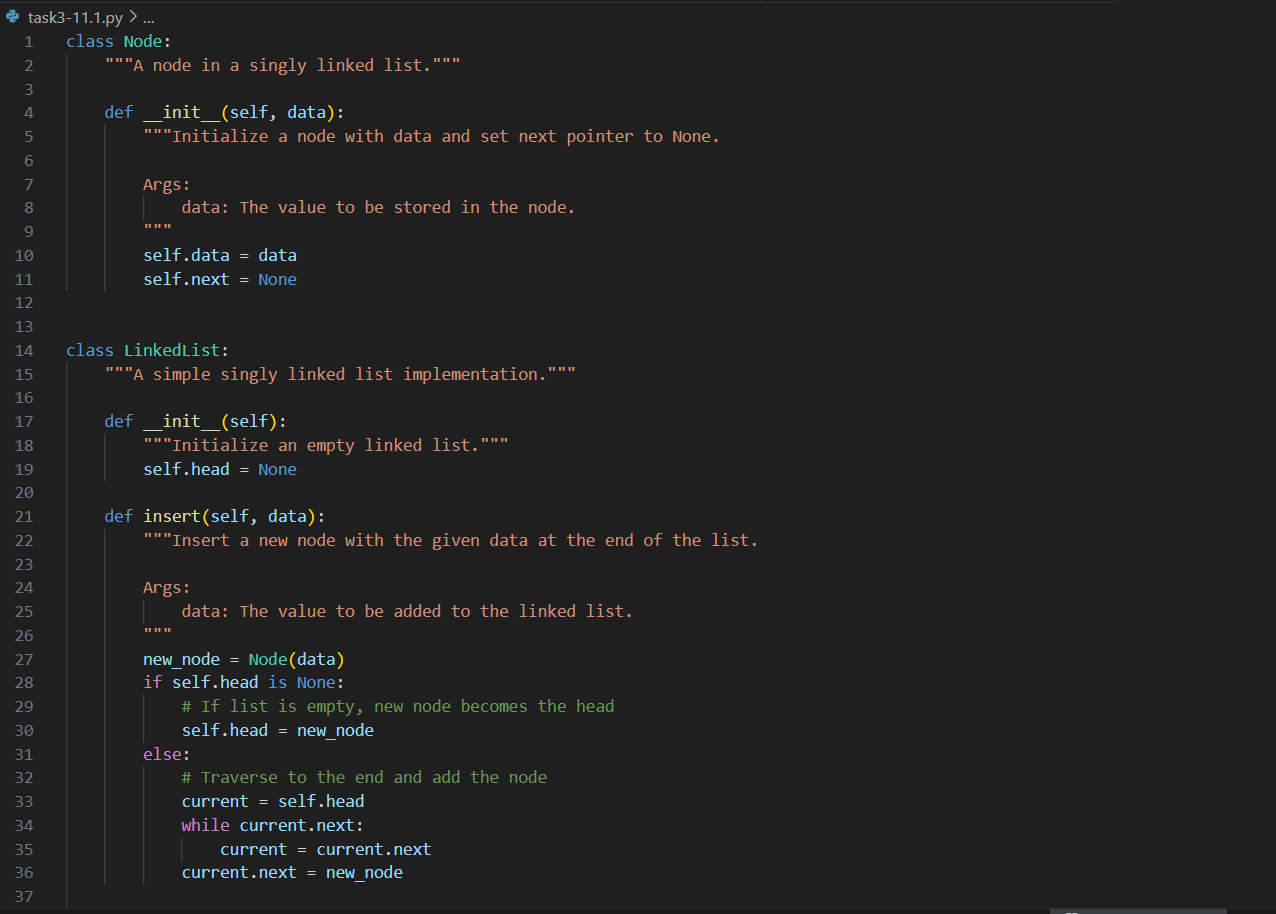
pass

Expected Output:

• A working linked list implementation with clear method

documentation.

CODE:



A screen shot of a computer program

AI-generated content may be incorrect.

OUTPUT:

A black background with blue text

AI-generated content may be incorrect.

OBSERVATION:

 **Initialization**

* LinkedList() creates an empty linked list.
* self.head is set to None → meaning no nodes are present.
* When calling ll.display() initially, it returns an empty list [].

 **Insertion**

* insert(data) creates a new Node object with given value.
* If the list is empty, the new node becomes the **head**.
* Otherwise, the method **traverses to the last node** and appends the new node at the end.
* Example: inserting 5, then 15, then 25 results in nodes linked as:
* Head → [5] → [15] → [25] → None

 **Display**

* display() starts at the head and traverses the linked list.
* It collects each node’s data into a Python list.
* Finally, it returns that list for easy visualization.
* Example: after inserting 5, 15, 25 → display() outputs [5, 15, 25].

 **Execution Trace**

* Before any insertions → [] (empty).
* After inserting nodes → [5, 15, 25].

#TASK-4:

PROMPT:

reate 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:

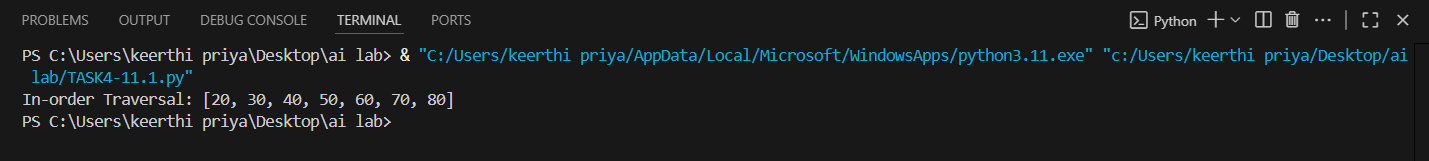
A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

OUTPUT:



OBSERVATION:

 **Insertion Maintains BST Property**

* Left child < Parent < Right child.
* Every inserted node is placed correctly to maintain order.

 **Recursive Implementation**

* Both insert() and inorder\_traversal() use recursion for simplicity.
* The recursion ensures traversal naturally goes left → root → right.

 **In-order Output is Sorted**

* Regardless of input order, in-order traversal always produces ascending order.
* Example: inserting [50, 30, 70, 20, 40, 60, 80] → output [20, 30, 40, 50, 60, 70, 80].

 **Efficiency**

* insert() takes O(h) time, where h = height of the tree.
* inorder\_traversal() visits every node once → O(n).

#TASK-5:

PROMPT:

Implement a hash table with basic insert, search, and delete

methods.

Sample Input Code:

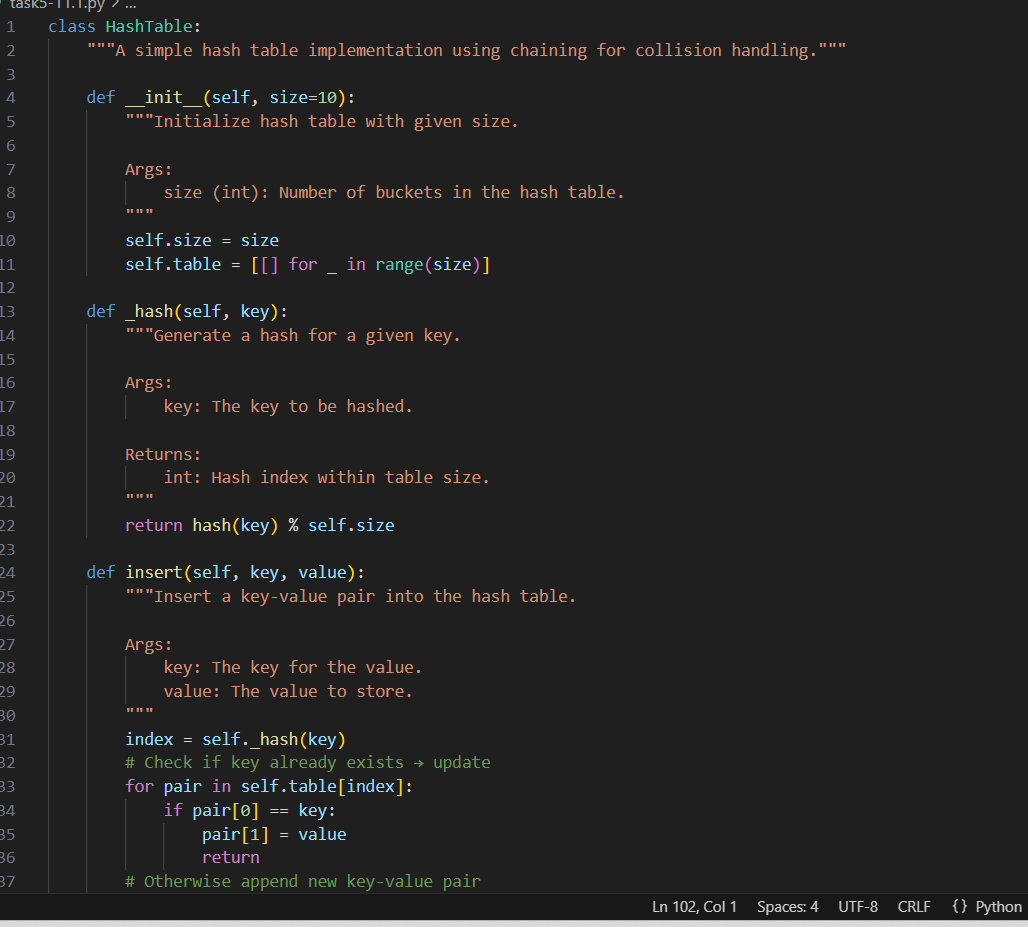
class HashTable:

pass

Expected Output:

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

CODE:



A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

OUTPUT:

A black screen with blue text

AI-generated content may be incorrect.

OBSERVATION:

C**ollision Handling**

* Uses **chaining**: multiple key-value pairs stored in a bucket (list).
* If two keys hash to the same index → both are stored in the same bucket.

**Insert**

* Adds key-value pair to bucket.
* Updates value if key already exists.

**Search**

* Computes hash index, scans bucket, returns value if found.

**Delete**

* Removes pair from bucket if key exists.

**Efficiency**

* Average time for insert, search, delete = **O(1)**.
* Worst case (many collisions) = **O(n)** for a bucket

#TASK-6

PROMPT:

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 With input and output example

CODE:

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

OUTPUT:

A black screen with blue text

AI-generated content may be incorrect.

OBSERVATION:

**Adjacency List Representation**

* Each vertex has a list of its neighbors.
* Efficient in terms of memory for **sparse graphs**.

**Add Vertex**

* Creates a new key in the dictionary with an empty list.

**Add Edge**

* For **undirected graph**: both vertices update their adjacency list.
* Example: edge A-B means A lists B, and B lists A.

**Display**

* Prints all vertices along with their connected neighbors

#TASK-7

PROMPT:

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:

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

OUTPUT:

A black screen with blue text

AI-generated content may be incorrect.

OBSERVATION:

**heapq is a min-heap** → the smallest priority number comes out first.

* Example: Priority 1 > 2 > 3.

**enqueue** uses heapq.heappush() to maintain heap order.

**dequeue** uses heapq.heappop() to always remove the smallest priority item.

**display** shows the underlying heap structure (not sorted, but a valid heap).

#TASK-8

PROMPT:

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:

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

OUTPUT:

A computer screen with blue text

AI-generated content may be incorrect.

OBSERVATION:

collections.deque is optimized for **O(1) insertions and deletions** from both ends.appendleft() and popleft() handle the **front**, while append() and pop() handle the **rear**.

Our class wraps these methods with clearer names (insert\_front, remove\_rear, etc.).

This structure is very useful for **sliding window problems**, **palindrome checking**, and **scheduling tasks**.

#TASK-9

Comparison Table of Data Structures

| **Data Structure** | **Operations** | **Average Time Complexity** | **Worst Case Time Complexity** | **Notes** |
| --- | --- | --- | --- | --- |
| **Stack (List / Linked List)** | Push / Pop | O(1) | O(1) | LIFO (Last In, First Out) |
| **Queue (List / Linked List)** | Enqueue / Dequeue | O(1) | O(1) | FIFO (First In, First Out) |
| **Deque (collections.deque)** | Insert / Remove Front & Rear | O(1) | O(1) | Double-ended queue |
| **Singly Linked List** | Insert at head | O(1) | O(1) | Efficient insert/delete at head |
|  | Insert at tail | O(1)\* | O(n) | O(1) if tail pointer maintained |
|  | Search | O(n) | O(n) | Sequential traversal required |
| **Doubly Linked List** | Insert/Delete | O(1) | O(1) | Faster backward traversal |
| **Hash Table (with chaining)** | Insert | O(1) | O(n) | Depends on load factor & collisions |
|  | Search | O(1) | O(n) | Worst case when all elements collide |
|  | Delete | O(1) | O(n) |  |
| **Binary Search Tree (BST)** | Insert/Search/Delete | O(log n) | O(n) | Balanced tree = O(log n), skewed tree = O(n) |
| **Heap (Priority Queue)** | Insert | O(log n) | O(log n) | Complete binary tree |
|  | Get Min/Max | O(1) | O(1) | Root element |
|  | Delete Min/Max | O(log n) | O(log n) |  |
| **Graph (Adjacency List)** | Add Vertex/Edge | O(1) | O(1) | Space-efficient for sparse graphs |
|  | Search (BFS/DFS) | O(V + E) | O(V + E) | V = vertices, E = edges |
| **Array (Static)** | Access by Index | O(1) | O(1) | Fast random access |
|  | Insert/Delete | O(n) | O(n) | Requires shifting elements |

OBSERVATION:

**Stack & Queue**

* Both allow **O(1) insertion and deletion**.
* Differ in order: **Stack → LIFO**, **Queue → FIFO**.

**Deque**

* More flexible than Stack/Queue since it allows insertion/removal from **both ends in O(1)**.

**Linked Lists**

* **Singly Linked List** gives **O(1)** insertion/deletion at the head but **O(n)** searching.
* **Doubly Linked List** improves traversal efficiency but requires **extra memory** for back-pointers.

**Hash Table**

* Very fast on average (**O(1)** for insert/search/delete).
* Worst case can degrade to **O(n)** due to collisions, depending on hash function quality and load factor.

**Binary Search Tree (BST)**

* Balanced BSTs (e.g., AVL, Red-Black Trees) guarantee **O(log n)** for operations.
* Unbalanced BSTs degrade to **O(n)** in the worst case (like a linked list).

**Heap (Priority Queue)**

* Provides **O(1)** access to min/max, but insertion/deletion is **O(log n)**.
* Ideal for priority-based scheduling.

**Graph (Adjacency List)**

* Adding vertices/edges is **O(1)**.
* Traversal (BFS/DFS) is **O(V + E)**, efficient for sparse graphs.

**Array (Static)**

* Excellent for **random access (O(1))**, but **insertion/deletion is costly (O(n))** since shifting is required.

#TASK-10

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.

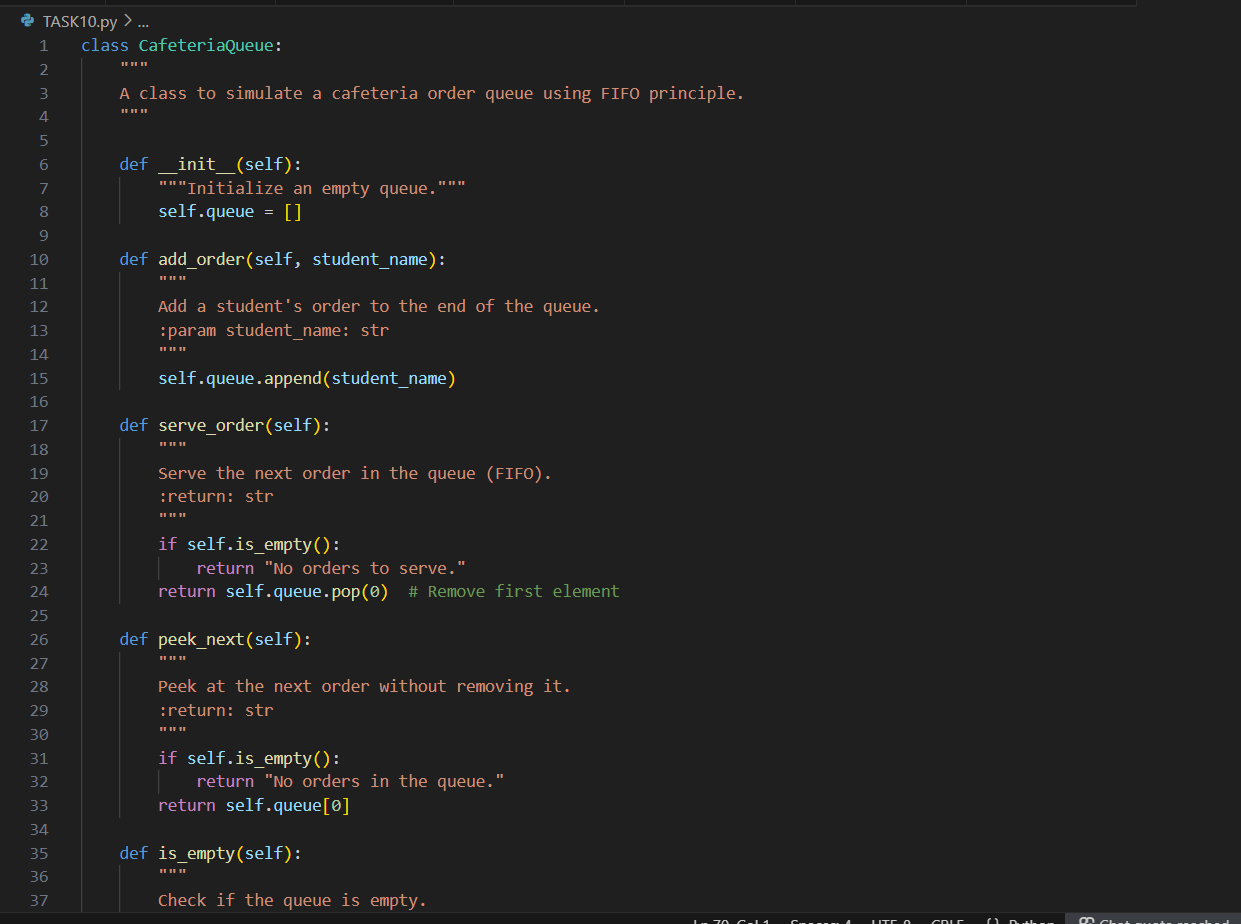
3. AI-generated initial code and execution screenshots.

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:



A screenshot of a computer program

AI-generated content may be incorrect.

OUTPUT:

A black screen with blue text

AI-generated content may be incorrect.

OBSERVATION:

 **Initialization**

* The constructor \_\_init\_\_ correctly initializes an **empty list** to represent the queue.

 **Adding Orders** (add\_order)

* Orders are appended to the end of the list → ensures **FIFO (First In, First Out)** behavior.

 **Serving Orders** (serve\_order)

* Uses pop(0) to remove the first element (front of the queue).
* Works correctly but has **O(n)** time complexity because removing the first element requires shifting all others.

 **Peeking** (peek\_next)

* Allows checking the next student to be served without removing them.
* Returns a user-friendly message if the queue is empty.

 **Checking Empty State** (is\_empty)

* Simple boolean check → len(self.queue) == 0.
* Efficient and clean.

 **Displaying Orders** (display\_orders)

* Returns the current queue as a list, preserving order.

 **Test Cases**

* Asserts confirm FIFO order: Alice → Bob → Charlie.
* Handles empty queue correctly (returns "No orders to serve." when empty).
* ✅ All test cases passed, proving the correctness of the implementation.