**Assignment 11.1**

**Task 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  
**Prompt:**

Write a Python class named Stack that implements a stack data structure.

The class should include the following methods with proper docstrings:

- push(item): adds an item to the top of the stack

- pop(): removes and returns the top item of the stack, raises IndexError if empty

- peek(): returns the top item without removing it, raises IndexError if empty

- is\_empty(): returns True if the stack is empty, otherwise False

Provide clean, well-documented code.

**Code generated:**

**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 screenshot of a computer

AI-generated content may be incorrect.**

**Observation:**

Here’s an observation for the stack implementation:

The class Stack correctly follows the LIFO (Last In, First Out) principle.

Methods are well-defined:

push appends an element to the list.

pop removes the last element with error handling for empty stacks.

peek returns the last element without removing it, also with error handling.

is\_empty checks whether the stack is empty.

The implementation uses Python’s built-in list, which already provides efficient append and pop operations (O(1) average).

Proper docstrings make the code self-explanatory and easier to maintain.

Example usage confirms that the methods work as expected.

**Task 2**

Queue Implementation  
Task: Use AI to implement a Queue using Python lists.  
Sample Input Code:  
class Queue:  
pass  
**Prompt:**

Write a Python class named Queue that implements a FIFO queue using Python lists.

The class should include the following methods with proper docstrings:

- enqueue(item): adds an item to the end of the queue

- dequeue(): removes and returns the front item, raises IndexError if empty

- peek(): returns the front item without removing it, raises IndexError if empty

- size(): returns the number of items in the queue

Provide clean, well-documented code.

**Code generated:**

**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 black screen with blue text

AI-generated content may be incorrect.**

**Observation:**

The queue implementation correctly follows the **FIFO (First In, First Out)** principle. The enqueue method appends items to the end of the list, while dequeue removes elements from the front with error handling for empty queues. The peek method provides access to the front item without removal, ensuring safe inspection. The size method reliably reports the number of items. Internally, Python’s list is used, which makes enqueue efficient (O(1)) but dequeue slightly less efficient (O(n)) due to shifting elements. Docstrings make the class self-explanatory, and example usage confirms correctness. Overall, it is a simple and functional queue design.

**Task 3**

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

**Prompt:**

Write a Python program to implement a singly linked list.

Define two classes:

1. Node: represents each node with data and a next pointer.

2. LinkedList: manages the list with methods:

- insert(data): adds a new node at the end of the list

- display(): prints the linked list elements in order

Include proper docstrings and clear, well-structured code.  
**Code generated:**

**A screenshot 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:**

Here’s a clear **explanation** of the singly linked list code:

* **Node class**: Represents each element of the linked list. Each node has two parts:
  + data: stores the actual value.
  + next: a pointer (reference) to the next node, initially set to None.
* **LinkedList class**: Manages the list of nodes.
  + \_\_init\_\_: Starts with an empty list (head = None).
  + insert(data): Creates a new node and adds it at the end. If the list is empty, the new node becomes the head. Otherwise, it traverses to the last node and attaches the new node.
  + display(): Traverses from head to the last node, printing each value with -> to show links. Ends with None to indicate the list’s end.

**Task 4**

Binary Search Tree (BST)  
Task: Use AI to create a BST with insert and in-order traversal methods.  
**Prompt:**

Write a Python program to implement a Binary Search Tree (BST).

Define a BSTNode class and a BST class with the following methods:

1. insert(data): recursively insert a new node into the BST.

2. in\_order\_traversal(): recursively print the nodes in ascending order.

Include proper docstrings and clear code structure.

**Code generated:**

**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:**

The Binary Search Tree (BST) implementation is correct and follows standard principles. Each node has data, left, and right pointers, making hierarchical storage efficient. The insert method uses recursion to place new values: smaller values go to the left, larger to the right, and duplicates are ignored. The in\_order\_traversal method visits nodes in left–root–right order, ensuring the output is sorted in ascending order. Helper functions (\_insert\_recursive, \_in\_order\_recursive) keep the code clean and modular. Example usage demonstrates successful insertion and traversal, producing 20 30 40 50 60 70 80. Overall, it is a functional and well-structured BST design

Task 5

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

methods.

**Prompt:**

Write a Python program to implement a Hash Table using chaining.

Define a HashTable class with these methods:

1. insert(key, value): stores key-value pairs, handling collisions with chaining.

2. search(key): returns the value for a given key or None if not found.

3. delete(key): removes the key-value pair if it exists.

Include proper docstrings and comments for clarity.

**Code generated**:

A screenshot 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:**

1.Structure:

* The hash table is an array of buckets (self.table), each initially an empty list.
* Each bucket stores key-value pairs to handle collisions.

2.Hash Function:

* \_hash(key) uses Python’s built-in hash() function and modulo operation to map keys to bucket indices.

3.Insert:

* Computes the bucket index.
* Checks if the key exists: if yes, updates the value; otherwise, appends the new key-value pair.

4.Search:

* Looks in the corresponding bucket for the key and returns the value if found; else returns None.

5.Delete:

* Searches the bucket and removes the key-value pair if present.

6.Display:

* Prints all buckets with their key-value pairs for easy visualization.

**Task 6**

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

**Prompt:**

Write a Python program to implement a graph using an adjacency list.

Define a Graph class with the following methods:

1. add\_vertex(vertex): adds a vertex to the graph

2. add\_edge(v1, v2): adds an edge between two vertices

3. display(): prints each vertex and its connected vertices

Include proper docstrings and well-commented code.

**Code generated:**

**A screenshot 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:**

The graph implementation correctly uses an adjacency list to represent connections efficiently. Each vertex maps to a list of adjacent vertices, allowing fast lookups and easy addition of edges. The add\_vertex method ensures new vertices are initialized with empty adjacency lists. The add\_edge method updates the adjacency lists of both vertices (for undirected graphs) or one vertex (for directed graphs), supporting flexible edge creation. The display method provides a clear visual representation of each vertex and its connections. Overall, this approach is memory-efficient for sparse graphs and allows easy traversal and modification.