ASSIGNMENT-11.1

Name : P .Reshmitha Reddy

Roll.no : 2403A510A7

Course : AI Assisted Coding

Batch : 05

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

Output:
• A functional stack implementation with all required methods and
docstrings.

Prompt :#generate a Stack class with push, pop, peek, and is_empty
methods.

Code :

```python
from typing import Generic, TypeVar, List

T = TypeVar("T")

class Stack(Generic[T]):
    def __init__(self) -> None:  # Fixed constructor name
        self._items: List[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T:
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self) -> T:
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]

    def is_empty(self) -> bool:
        return len(self._items) == 0

```

```python
# Usage
s = Stack[int]()
s.push(10)
s.push(20)
print(s.peek())  # Output: 20
print(s.pop())   # Output: 20
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & C:/Users/Reshm/anaconda3/python.exe c:/Users/Reshm/Desktop/AIAC/11.1.py
20
20
PS C:\Users\Reshm\Desktop\AIAC>
```

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 :# implement a Queue using Python lists.
Sample

Code :

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

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

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.items.pop(0)

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.items[0]

    def size(self):
        return len(self.items)
    def is_empty(self):
        return len(self.items) == 0
```

```
22
23  # Usage example
24  q = Queue()
25  q.enqueue(10)
26  q.enqueue(20)
27  print(q.peek())
28  print(q.dequeue())
29  print(q.size())
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & "C:/Program Files/Python313/python.exe" c:/Users/Reshm/Desktop/AIAC/11.1.
py
10
10
1
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The Queue class uses a Python list to store elements, following the FIFO (First-In-First-Out) principle.

The enqueue method adds elements to the end of the list.

The dequeue method removes and returns the element at the front (index 0). It raises an IndexError if the queue is empty.

The peek method returns the front element without removing it, also raising an IndexError if empty.

The size method returns the number of elements in the queue.

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 :# generate a Singly Linked List with insert and display methods.

Code :

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

class SinglyLinkedList:
    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=" -> " if current.next else "\n")
            current = current.next

# Usage example
ll = SinglyLinkedList()
ll.insert(10)
ll.insert(20)
ll.insert(30)
ll.display()
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & "C:/Program Files/Python313/python.exe" c:/Users/Reshm/Desktop/AIAC/11.1.py
10 -> 20 -> 30
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The code defines a Node class to represent each element in the list, holding data and a reference to the next node.

The SinglyLinkedList class manages the linked list, with a head pointer to the first node.

The insert method adds a new node to the end of the list. If the list is empty, the new node becomes the head.

The display method traverses the list and prints each node's data, showing the sequence of elements.

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 :# create a BST with insert and in-order traversal methods.

Code :

```python
class BSTNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, data):
        def _insert(node, data):
            if node is None:
                return BSTNode(data)
            if data < node.data:
                node.left = _insert(node.left, data)
            else:
                node.right = _insert(node.right, data)
            return node
        self.root = _insert(self.root, data)

    def in_order_traversal(self):
        def _in_order(node):
```

```
            if node:
                _in_order(node.left)
                print(node.data, end=" ")
                _in_order(node.right)
        _in_order(self.root)
        print()

# Usage example
bst = BST()
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
bst.insert(60)
bst.insert(80)
bst.in_order_traversal()
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & "C:/Program Files/Python313/python.exe" c:/Users/Reshm/Desktop/AIAC/11.1.py
20 30 40 50 60 70 80
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The code defines a BSTNode class to represent each node, storing data and references to left and right children.

The BST class manages the tree, with a root attribute pointing to the top node.

The insert method is implemented recursively, placing new values in the correct position based on BST properties (left for smaller, right for larger or equal).

The in_order_traversal method is also recursive and prints the tree's values in ascending order.

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 : # implement a hash table with basic insert, search, and delete methods.

Code :

```python
class HashTable:
    def __init__(self, size=10):
        # Initialize the hash table with empty lists for chaining
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        # Compute 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."""
        idx = self._hash(key)
        # Check if key already exists and update
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                self.table[idx][i] = (key, value)
                return
        # If key does not exist, append new key-value pair
        self.table[idx].append((key, value))

    def search(self, key):
        """Search for a value by key in the hash table."""
```

```
        idx = self._hash(key)
        for k, v in self.table[idx]:
            if k == key:
                return v
        return None

    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        idx = self._hash(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                del self.table[idx][i]
                return True
        return False

    def __str__(self):
        # For displaying the hash table
        return str(self.table)
```

```

# Usage example
ht = HashTable()
ht.insert("apple", 1)
ht.insert("banana", 2)
print(ht.search("apple"))
ht.delete("apple")
print(ht.search("apple"))
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & "C:/Program Files/Python313/python.exe" c:/Users/Reshm/Desktop/AIAC/11.1.
py
1
None
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The HashTable class uses a list of lists (chaining) to handle collisions, where each bucket contains key-value pairs.

The _hash method computes the index for a key using Python's built-in hash() function and modulo operation.

The insert method adds a new key-value pair or updates the value if the key already exists in the chain.

The search method looks for a key in the appropriate chain and returns its value if found, otherwise returns None.

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 :# implement a graph using an adjacency list.

Code :

```python
class Graph:
    def __init__(self):
        # Initialize an empty adjacency list
        self.adj_list = {}

    def add_vertex(self, vertex):
        """Add a vertex to the graph."""
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []

    def add_edge(self, src, dest):
        """Add an edge from src to dest (directed graph)."""
        if src not in self.adj_list:
            self.add_vertex(src)
        if dest not in self.adj_list:
            self.add_vertex(dest)
        self.adj_list[src].append(dest)
        # For undirected graph, also add: self.adj_list[dest].append(src)

    def display(self):
        """Display the adjacency list of the graph."""
        for vertex, neighbors in self.adj_list.items():
            print(f"{vertex}: {neighbors}")
```

```python

# Usage example
g = Graph()
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "C")
g.display()
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & "C:/Program Files/Python313/python.exe" c:/Users/Reshm/Desktop/AIAC/11.1.
py
A: ['B', 'C']
B: ['C']
C: []
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

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 vertices.

The add_vertex method adds a new vertex to the graph if it does not already exist.

The add_edge method adds a directed edge from the source to the destination. If either vertex does not exist, it is added automatically.

The display method prints each vertex and its list of neighbors,

Task 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 :#implement a priority queue using Python's heapq module.

Code :

```python
import heapq

class PriorityQueue:
    def __init__(self):
        # Initialize an empty heap
        self.heap = []

    def enqueue(self, priority, item):
        """Add an item with a given priority to the queue."""
        heapq.heappush(self.heap, (priority, item))

    def dequeue(self):
        """Remove and return the item with the highest priority (lowest value)."""
        if not self.is_empty():
            return heapq.heappop(self.heap)[1]
        raise IndexError("dequeue from empty priority queue")

    def display(self):
        """Display the contents of the priority queue."""
        print([item for priority, item in sorted(self.heap)])

    def is_empty(self):
        return len(self.heap) == 0
```

```python
    # Usage example
pq = PriorityQueue()
pq.enqueue(2, "task2")
pq.enqueue(1, "task1")
pq.enqueue(3, "task3")
pq.display()
print(pq.dequeue())
pq.display()
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & "C:/Program Files/Python313/python.exe" c:/Users/Reshm/Desktop/AIAC/11.1.
py
['task1', 'task2', 'task3']
task1
['task2', 'task3']
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The PriorityQueue class uses Python's heapq module to maintain a min-heap, ensuring efficient priority queue operations.

The enqueue method adds an item with a given priority to the queue. Lower priority values are considered higher priority.

The dequeue method removes and returns the item with the highest priority (lowest priority value). It raises an IndexError if the queue is empty.

The display method prints the items in the queue, sorted by priority.

The is_empty method checks if the queue is empty.

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 :# implement a double-ended queue using collections.deque.

Code :

```python
from collections import deque

class DequeDS:
    """Double-ended queue supporting insertions and removals from both ends."""

    def __init__(self):
        """Initialize an empty deque."""
        self.deque = deque()

    def add_front(self, item):
        """Insert an item at the front of the deque."""
        self.deque.appendleft(item)

    def add_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):
        if self.is_empty():
            raise IndexError("remove_rear from empty deque")
        return self.deque.pop()

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

    def display(self):
        """Display the contents of the deque."""
        print(list(self.deque))

# Usage example
dq = DequeDS()
dq.add_front(1)
dq.add_rear(2)
dq.add_front(0)
dq.display()
print(dq.remove_rear())
print(dq.remove_front())
dq.display()
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & "C:/Program Files/Python313/python.exe" c:/Users/Reshm/Desktop/AIAC/11.1.
py
[0, 1, 2]
2
0
[1]
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The DequeDS class uses Python's collections.deque to efficiently support insertions and removals from both ends.

The add_front and add_rear methods insert items at the front and rear, respectively.

The remove_front and remove_rear methods remove and return items from the front and rear, raising an IndexError if the deque is empty.

The is_empty method checks if the deque is empty.

Task Description #9 – AI-Generated Data Structure Comparisons
Task: 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.

Prompt :# generate a comparison table of different data structures (stack,

queue, linked list, etc.) including time complexities.

Code :

```
| Data Structure        | Access   | Search   | Insertion | Deletion | Notes                                  |
|-----------------------|----------|----------|-----------|----------|----------------------------------------|
| Array (static)        | O(1)     | O(n)     | O(n)      | O(n)     | Fixed size                             |
| Array (dynamic/list)  | O(1)     | O(n)     | O(1)*     | O(n)     | Amortized O(1) for append              |
| Stack (list)          | O(n)     | O(n)     | O(1)      | O(1)     | LIFO, insert/delete at end             |
| Queue (list)          | O(n)     | O(n)     | O(1)      | O(n)     | FIFO, dequeue O(n) in list             |
| Queue (deque)         | O(n)     | O(n)     | O(1)      | O(1)     | Efficient at both ends                 |
| Singly Linked List    | O(n)     | O(n)     | O(1)      | O(1)     | Insert/delete at head O(1)             |
| Doubly Linked List    | O(n)     | O(n)     | O(1)      | O(1)     | Insert/delete at both ends O(1)        |
| Hash Table            | N/A      | O(1)     | O(1)      | O(1)     | Worst case O(n) for collisions         |
| Binary Search Tree    | O(log n) | O(log n) | O(log n)  | O(log n) | Balanced BST, worst case O(n)          |
| Heap (Priority Q)     | O(n)     | O(n)     | O(log n)  | O(log n) | Access min/max O(1), not sorted order  |

*Note: For dynamic arrays (like Python lists), insertion is amortized O(1) when appending, but O(n) for insert else
```

Observation :

The code provides a markdown table comparing various data structures (array, stack, queue, linked list, hash table, BST, heap) and their time complexities for access, search, insertion, and deletion operations.

The table is intended for documentation or study purposes, not for execution in Python.

It helps quickly compare the efficiency of different data structures for common operations.

Notes and footnotes clarify special cases, such as amortized insertion time for dynamic arrays and worst-case scenarios for hash tables and BSTs.

Task Description #10 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 :#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.

Code :

Feature	Data Structure	Justification
Student Attendance Tracking	Linked List	A linked list allows efficient insertion of attendance logs as st
Event Registration System	Hash Table	A hash table enables fast search, registration, and removal of pa
Library Book Borrowing	Binary Search Tree (BST)	A BST allows efficient tracking, searching, and updating
Bus Scheduling System	Graph	A graph models bus routes and stop connections, supporting route
Cafeteria Order Queue	Queue	A queue ensures students are served in the order they arrive (FIF

```python
class CafeteriaOrderQueue:
    """Queue to manage cafeteria orders in FIFO order."""

    def __init__(self):
        """Initialize an empty order queue."""
        self.queue = []

    def add_order(self, student_name, order):
        """Add a new order to the queue."""
        self.queue.append((student_name, order))

    def serve_order(self):
        """Serve the next order in the queue."""
        if self.is_empty():
            print("No orders to serve.")
            return None
        student_name, order = self.queue.pop(0)
        print(f"Serving {student_name}: {order}")
        return (student_name, order)

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

```python

    def display_orders(self):
        """Display all pending orders in the queue."""
        if self.is_empty():
            print("No pending orders.")
        else:
            print("Pending orders:")
            for idx, (student_name, order) in enumerate(self.queue, 1):
                print(f"{idx}. {student_name}: {order}")

# Usage example
cafeteria = CafeteriaOrderQueue()
cafeteria.add_order("Alice", "Veg Sandwich")
cafeteria.add_order("Bob", "Chicken Burger")
cafeteria.add_order("Charlie", "Pasta")
cafeteria.display_orders()
cafeteria.serve_order()
cafeteria.display_orders()
```

Output :

```
:/Program Files/Python313/python.exe" c:/Users/Reshm/Desktop/AIAC/11:/Program Fil:/Program Files/Python313/python.exe" c:/Users/Reshm/De
sktop/An.exe" c:/Users/Reshm/Desktop/AIAC/11.1.py
n.exe" c:/Users/Reshm/Desktop/AIAC/11.1.py
n.exe" c:/Users/Reshm/Desktop/AIAC/11.1.py
n.exe" c:/Users/Reshm/Desktop/AIAC/11.1.py
n.exe" c:/Users/Reshm/Desktop/AIAC/11.1.py
n.exe" c:/Users/Reshm/Desktop/AIAC/11.1.py
Pending orders:
1. Alice: Veg Sandwich
2. Bob: Chicken Burger
3. Charlie: Pasta
```

```
Serving Alice: Veg Sandwich
Pending orders:
1. Bob: Chicken Burger
2. Charlie: Pasta
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The table clearly maps each campus management feature to the most suitable data structure, with concise justifications for each choice based on operational needs (e.g., fast search, sequential access, route modeling, FIFO service).

The Python program implements a cafeteria order queue using a simple list-based queue, supporting order addition, serving (removal in FIFO order), and displaying pending orders.

The code includes docstrings and comments, making it easy to understand and maintain.

The usage example demonstrates adding, serving, and displaying orders, reflecting real-world cafeteria operations.