Assignment – 11.5

Name: Shashank.B
Batch: 21
Ht.no: 2303A51431

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 to AI

"Generate a Python Stack class with push, pop, peek, and is_empty methods with comments and docstrings."

```python
'''Generate a Python Stack class with push, pop, peek, and is_empty methods with
comments and docstrings'''
class Stack:
    """A simple implementation of a stack data structure."""

    def __init__(self):
        """Initialize an empty stack."""
        self.items = []

    def push(self, item):
        """Add an item to the top of the stack.

        Args:
            item: The item to be added to the stack.
        """
        self.items.append(item)

    def pop(self):
        """Remove and return the item at the top of the stack.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.items.pop()

    def peek(self):
        """Return the item at the top of the stack without removing it.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]

    def is_empty(self):
        """Check if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self.items) == 0
#sample  input and output
stack = Stack()
stack.push(1)
stack.push(2)
print(stack.peek())  # Output: 2
print(stack.pop())   # Output: 2
```

Observation

The Stack data structure works in LIFO order. AI-generated code improved clarity using docstrings and error handling.

```
PS C:\Users\2303A\OneDrive\Desktop\python course>
top/python course/aiassistant.py"
2
2
False
1
True
World
World
Hello
```

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

"Generate a Queue class using Python list with enqueue, dequeue, peek, and size methods."

```python
'''Generate a Queue class using Python list with enqueue, dequeue, peek, and size methods'''
class Queue:
    def __init__(self):
        self.items = []

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

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            raise IndexError("Dequeue from an empty queue")

    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
            raise IndexError("Peek from an empty queue")

    def size(self):
        return len(self.items)

    def is_empty(self):
        return len(self.items) == 0
# Example usage:
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
print(queue.peek())  # Output: 1
print(queue.dequeue())  # Output: 1
print(queue.size())  # Output: 1
```

Observation

Queue follows FIFO principle. AI helped in designing efficient queue operations.

```
A/OneDrive/Desktop/python course/aiassistant.py"
1
1
1
```

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."

```python
'''Generate a singly linked list with insert and display methods'''
class Node:
    '''Node class for singly linked list'''
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    '''Linked list class with insert and display methods'''
    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
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
    def display(self):
        '''Display the linked list'''
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
# Example usage
linked_list = LinkedList()
linked_list.insert(10)
linked_list.insert(20)
linked_list.insert(30)
```

Observation

Linked list stores nodes dynamically. AI-generated structure simplified pointer handling.

```
A/OneDrive/Desktop/python course/aiassistant.py
10 -> 20 -> 30 -> None
```

Task Description #4 – 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

“Generate a Hash Table class with insert, search, delete using chaining.”

```python
'''Generate a Hash Table class with insert, search, delete using chaining'''
class HashTable:
    def __init__(self):
        self.size = 10
        self.table = [[] for _ in range(self.size)]

    def hash_function(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        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):
        index = self.hash_function(key)
        for k, v in self.table[index]:
            if k == key:
                return v  # Return the value associated with the key
        return None  # Key not found

    def delete(self, key):
        index = self.hash_function(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]  # Remove the key-value pair
                return True
        return False  # Key not found
# Example usage
hash_table = HashTable()
hash_table.insert("name", "Kushal")
hash_table.insert("age", 30)
print(hash_table.search("name"))  # Output: Kushal
print(hash_table.search("age"))   # Output: 30
hash_table.delete("name")
print(hash_table.search("name"))  # Output: None
```

Observation

Hash table enables fast O(1) lookup. AI suggested chaining to handle collisions.

```
PS C:\Users\2303A\OneDrive\Desktop\pytho
top/python course/aiassistant.py"
Kushal
30
None
```

Task Description #5 – 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

"Generate a Graph class using adjacency list with add_vertex, add_edge, display methods."

```python
'''Generate a Graph class using adjacency list with add_vertex, add_edge, display methods'''
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertex1)

    def display(self):
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {edges}")
# Example usage
g = Graph()
g.add_vertex("A")
g.add_vertex("B")
g.add_vertex("C")
g.add_edge("A", "B")
g.add_edge("A", "C")
g.display()
```

Observation

Graph adjacency list efficiently stores network connections. AI helped in structuring graph logic.

```
PS C:\Users\2303A\OneDrive\Desktop\python course>
top/python course/aiassistant.py"
A: ['B', 'C']
B: ['A']
C: ['A']
```

Task Description #6: Smart Hospital Management System – Data

Structure Selection

A hospital wants to develop a Smart Hospital Management System that handles:

1. Patient Check-In System – Patients are registered and treated in order of arrival.
2. Emergency Case Handling – Critical patients must be treated first. 3. Medical Records Storage – Fast retrieval of patient details using ID.
4. Doctor Appointment Scheduling – Appointments sorted by time.
5. Hospital Room Navigation – Represent connections between wards and rooms.

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.

Feature	Data Structure	Justification
Patient Check-In	Queue	FIFO order for patient arrival
Emergency Handling	Priority Queue	Critical patients treated first
Medical Records	Hash Table	Fast ID-based retrieval
Appointment Scheduling	BST	Sorted by time
Room Navigation	Graph	Represents connected rooms

Task Description #7: Smart City Traffic Control System

A city plans a Smart Traffic Management System that includes:

1. Traffic Signal Queue – Vehicles waiting at signals.
2. Emergency Vehicle Priority Handling – Ambulances and fire trucks prioritized.
3. Vehicle Registration Lookup – Instant access to vehicle details. 4. Road Network Mapping – Roads and intersections connected logically.
5. Parking Slot Availability – Track available and occupied slots.

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.

Feature	**Data Structure**	**Justification**
Traffic Signal Queue	Queue	Vehicles wait in order
Emergency Priority	Priority Queue	Ambulances first
Vehicle Lookup	Hash Table	Instant access
Road Network	Graph	Intersections and roads
Parking Slots	Deque	Add/remove from both ends

Task Description #8: 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.

Feature	**Data Structure**	**Justification**
Shopping Cart	Linked List	Dynamic add/remove
Order Processing	Queue	FIFO order
Top Products	Priority Queue	Ranked by sales
Product Search	Hash Table	Fast lookup
Delivery Routes	Graph	Location connections