LAB – 11.1

AI-Assisted Coding

Name – Rounak Raj

Hall-Ticket – 2303A54043

Lab 11 – Data Structures with AI: Implementing Fundamental

Structures

Lab Objectives

• Use AI to assist in designing and implementing fundamental data structures in Python.

• Learn how to prompt AI for structure creation, optimization, and

documentation.

• Improve understanding of Lists, Stacks, Queues, Linked Lists,

Trees, Graphs, and Hash Tables.

• Enhance code quality with AI-generated comments and

performance suggestions.

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.

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.

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.

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.

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.

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.

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.

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.

Task Description #9 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.

Task Description #10: 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.

SOLUTION:

Part 1: Fundamental Data Structures

Task 1 – Stack Implementation

```
class Stack:
    """A LIFO (Last-In, First-Out) stack implementation using a Python list."""

    def __init__(self):
        self.stack = []

    def push(self, item):
```

```
        """Add an item to the top of the stack."""
        self.stack.append(item)

    def pop(self):
        """Remove and return the top item. Raises IndexError if empty."""
        if self.is_empty():
            return "Stack is empty"
        return self.stack.pop()

    def peek(self):
        """Return the top item without removing it."""
        if self.is_empty():
            return None
        return self.stack[-1]

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

Task 2 – Queue Implementation

```
class Queue:
    """A FIFO (First-In, First-Out) queue implementation using a Python list."""

    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        """Add an item to the end of the queue."""
        self.queue.append(item)
```

```
    def dequeue(self):
        """Remove and return the front item. Raises error if empty."""
        if self.size() == 0:
            return "Queue is empty"
        return self.queue.pop(0)

    def peek(self):
        """View the first item in the queue."""
        return self.queue[0] if not self.is_empty() else None

    def size(self):
        """Return the number of items in the queue."""
        return len(self.queue)

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

Task 3 – Singly Linked List

```
class Node:
    """Represents a single node in a linked list."""
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    """A basic Singly Linked List implementation."""
    def __init__(self):
        self.head = None

    def insert(self, data):
```

```python
        """Insert a new node at the end of the list."""
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def display(self):
        """Print all elements in the list."""
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" -> ".join(elements))
```

Task 4 – Binary Search Tree (BST)

```python
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


class BST:
    """Binary Search Tree with recursive insert and in-order traversal."""
    def __init__(self):
        self.root = None
```

```
    def insert(self, key):
        """Public method to insert a key."""
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        """Helper to find the correct insertion spot."""
        if key < node.val:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert_recursive(node.left, key)
        else:
            if node.right is None:
                node.right = Node(key)
            else:
                self._insert_recursive(node.right, key)

    def in_order_traversal(self, node):
        """Traverse the tree in sorted order."""
        if node:
            self.in_order_traversal(node.left)
            print(node.val, end=" ")
            self.in_order_traversal(node.right)
```

Task 5 – Hash Table (Chaining)

```
class HashTable:
```

```
"""Hash Table implementation using chaining for collision handling."""
def __init__(self, size=10):
    self.size = size
    self.table = [[] for _ in range(self.size)]

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

def insert(self, key, value):
    """Insert or update a key-value pair."""
    index = self._hash(key)
    for item in self.table[index]:
        if item[0] == key:
            item[1] = value
            return
    self.table[index].append([key, value])

def search(self, key):
    """Retrieve value by key. Returns None if not found."""
    index = self._hash(key)
    for item in self.table[index]:
        if item[0] == key:
            return item[1]
    return None

def delete(self, key):
    """Remove a key-value pair from the table."""
    index = self._hash(key)
    for i, item in enumerate(self.table[index]):
```

```
            if item[0] == key:
                del self.table[index][i]
                return True
        return False
```

Task 6 – Graph Representation

```
class Graph:
    """Graph implementation using an Adjacency List (dictionary)."""
    def __init__(self):
        self.adj_list = {}

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

    def add_edge(self, v1, v2):
        """Add an undirected edge between two vertices."""
        if v1 in self.adj_list and v2 in self.adj_list:
            self.adj_list[v1].append(v2)
            self.adj_list[v2].append(v1)

    def display(self):
        """Show all connections in the graph."""
        for vertex, neighbors in self.adj_list.items():
            print(f"{vertex}: {', '.join(map(str, neighbors))}")
```

Task 7 – Priority Queue

```
import heapq

class PriorityQueue:
```

```
    """Priority Queue implementation using Python's heapq module."""
    def __init__(self):
        self.elements = []

    def enqueue(self, item, priority):
        """Add an item with a priority (lower number = higher priority)."""
        heapq.heappush(self.elements, (priority, item))

    def dequeue(self):
        """Remove and return the item with the highest priority."""
        if self.elements:
            return heapq.heappop(self.elements)[1]
        return "Queue is empty"

    def display(self):
        """Display the current heap (internal structure)."""
        print(self.elements)
```

Task 8 – Deque

```
from collections import deque

class DequeDS:
    """Double-ended queue using collections.deque."""
    def __init__(self):
        self.items = deque()

    def add_front(self, item):
        """Insert item at the beginning."""
        self.items.appendleft(item)
```

```
    def add_rear(self, item):
        """Insert item at the end."""
        self.items.append(item)

    def remove_front(self):
        """Remove and return item from the front."""
        return self.items.popleft() if self.items else None

    def remove_rear(self):
        """Remove and return item from the rear."""
        return self.items.pop() if self.items else None
```

Task 9: Campus Resource Management System

Feature	**Data Structure**	**Justification**
Attendance Tracking	**Hash Table**	Provides $O(1)$ average time complexity for logging and looking up student IDs, making entry/exit recording nearly instantaneous.
Event Registration	**BST**	Allows for efficient searching and provides sorted lists of participants for check-in sheets or reports.
Library Borrowing	**Hash Table**	Quickly maps a unique Book ID (ISBN) to its current status and due date for fast transactions.
Bus Scheduling	**Graph**	Vertices represent stops and edges represent routes; essential for mapping connections and calculating travel paths.

Feature	Data Structure	Justification
Cafeteria Order	**Queue**	Follows the First-In, First-Out (FIFO) principle, ensuring students are served in the exact order they arrived.

Selected Feature Implementation: Cafeteria Order Queue

```
class CafeteriaQueue:
    """Handles student orders in the order they are placed."""
    def __init__(self):
        self.orders = []

    def place_order(self, student_name, order_detail):
        """Add a new order to the end of the queue."""
        self.orders.append({"name": student_name, "order": order_detail})
        print(f"Order added for {student_name}.")

    def serve_next(self):
        """Serve the student at the front of the line."""
        if not self.orders:
            print("No pending orders.")
            return
        current = self.orders.pop(0)
        print(f"Serving {current['name']}: {current['order']}")

# Example Usage
campus_cafe = CafeteriaQueue()
campus_cafe.place_order("Alice", "Coffee")
campus_cafe.place_order("Bob", "Sandwich")
```

campus_cafe.serve_next() # Serves Alice

Task 10: Smart E-Commerce Platform

Feature	**Data Structure**	**Justification**
Shopping Cart	**Linked List**	Ideal for dynamic additions and removals of products while browsing without needing a fixed size.
Order Processing	**Queue**	Ensures orders are processed sequentially (FIFO) to maintain fairness and operational flow.
Top-Selling Tracker	**Priority Queue**	A Max-Heap can efficiently keep the product with the highest sales count at the top for real-time ranking.
Search Engine	**Hash Table**	Uses Product IDs as keys to provide \$O(1)\$ lookup time, allowing customers to find products instantly.
Route Planning	**Graph**	Models the network of warehouses and delivery points to calculate the most efficient delivery paths.

Selected Feature Implementation: Product Search Engine

```
class ProductSearch:
    """Fast lookup system for e-commerce products using Hash Table logic."""
    def __init__(self):
        self.inventory = {}

    def add_product(self, product_id, name, price):
        """Store product details indexed by their unique ID."""
        self.inventory[product_id] = {"name": name, "price": price}
```

```python
    def find_product(self, product_id):
        """Retrieve product details instantly using its ID."""
        product = self.inventory.get(product_id)
        if product:
            return f"Found: {product['name']} - ${product['price']}"
        return "Product not found."

# Example Usage
store = ProductSearch()
store.add_product("P101", "Smartphone", 699)
store.add_product("P102", "Laptop", 1200)
print(store.find_product("P101"))
```