**AI ASSISTED CODING LAB 11.1**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

PRADEEP GUPTHA

2403A510C7

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  
Expected 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 :**

class Stack:

    def \_\_init\_\_(self):

        self.items = []

    def push(self, item):

        """Add an item to the top of the stack."""

        self.items.append(item)

    def pop(self):

        """Remove and return the top item of the stack. Raises IndexError if empty."""

        if self.is\_empty():

            raise IndexError("pop from empty stack")

        return self.items.pop()

    def peek(self):

        """Return the top item of the stack without removing it. Raises IndexError if empty."""

        if self.is\_empty():

            raise IndexError("peek from empty stack")

        return self.items[-1]

    def is\_empty(self):

        """Return True if the stack is empty, False otherwise."""

        return len(self.items) == 0

# Example usage with printed output

if \_\_name\_\_ == "\_\_main\_\_":

    s = Stack()

    print("Is stack empty?", s.is\_empty())  # True

    s.push(5)

    s.push(10)

    s.push(15)

    print("Top of stack:", s.peek())        # 15

    print("Pop:", s.pop())                  # 15

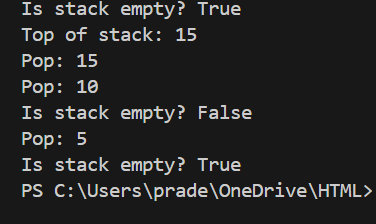
    print("Pop:", s.pop())                  # 10

    print("Is stack empty?", s.is\_empty())  # False

    print("Pop:", s.pop())                  # 5

    print("Is stack empty?", s.is\_empty())  # True

**OUTPUT :**

****

**OBSERVATION :**

* The code defines a [Stack](vscode-file://vscode-app/c:/code/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) class with standard stack operations: [push](vscode-file://vscode-app/c:/code/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html), [pop](vscode-file://vscode-app/c:/code/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html), [peek](vscode-file://vscode-app/c:/code/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html), and [is\_empty](vscode-file://vscode-app/c:/code/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html" \o ").
* The example usage demonstrates pushing three values (5, 10, 15) onto the stack.
* It prints the top value, pops all values one by one, and checks if the stack is empty after each operation.

**Task 2 Description** #– 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 Input Code:  
class Queue: pass

**CODE :**

class Queue:

    def \_\_init\_\_(self):

        self.items = []

    def enqueue(self, item):

        self.items.append(item)

    def dequeue(self):

        if not self.items:

            return None

        return self.items.pop(0)

    def peek(self):

        if not self.items:

            return None

        return self.items[0]

    def size(self):

        return len(self.items)

# Example

q = Queue()

q.enqueue(1)

q.enqueue(2)

q.enqueue(3)

print(q.dequeue())  # 1

print(q.peek())     # 2

print(q.size())     # 2

**OUTPUT :**

****

**OBSERVATION :**

 **FIFO principle** → First element in is the first one out.

 **enqueue()** → Adds to the end.

 **dequeue()** → Removes from the front.

 **peek()** → Looks at the front without removing.

 **size()** → Counts elements.

**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. Sample Input Code: class Node: pass class LinkedList: pass

**CODE :**

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class LinkedList:

    def \_\_init\_\_(self):

        self.head = None

    def insert(self, data):

        """Insert a new node at the end of the list"""

        new\_node = Node(data)

        if self.head is None:   # if list is empty

            self.head = new\_node

        else:

            current = self.head

            while current.next:   # move to the last node

                current = current.next

            current.next = new\_node

    def display(self):

        """Display all nodes in the list"""

        current = self.head

        while current:

            print(current.data, end=" -> ")

            current = current.next

        print("None")

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

    ll = LinkedList()

    ll.insert(10)

    ll.insert(20)

    ll.insert(30)

    ll.display()   # Output: 10 -> 20 -> 30 -> None

**OUTPUT :**



**OBSERVATION :**

 **ode** stores two things → data and next (address of next node).

 **LinkedList** uses head → starting point of the list.

 **insert()** → adds new node at the end of the list.

 **display()** → traverses nodes one by one until None.

 Output format shows the **chain of nodes** (e.g., 10 -> 20 -> 30 -> None).

**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 :** to create a BST with insert and in-order traversal methods.Sample input Code: class BST: pass

**CODE :**

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

class BST:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, root, key):

        """Recursive insert into BST"""

        if root is None:

            return Node(key)

        if key < root.key:

            root.left = self.insert(root.left, key)

        else:

            root.right = self.insert(root.right, key)

        return root

    def inorder(self, root):

        """In-order traversal (L -> Root -> R)"""

        if root:

            self.inorder(root.left)

            print(root.key, end=" ")

            self.inorder(root.right)

# Example

bst = BST()

root = None

for val in [50, 30, 70, 20, 40, 60, 80]:

    root = bst.insert(root, val)

print("Inorder Traversal:", end=" ")

bst.inorder(root)

**OBSERVATION :**

 BST maintains **sorted structure** → left < root < right.

 insert() → recursively finds correct position.

 inorder() → prints elements in **ascending order**.

 output: 20 30 40 50 60 70 80

**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 :**

class HashTable:

    def \_\_init\_\_(self, size=10):

        self.size = size

        self.table = [[] for \_ in range(size)]

    def \_hash(self, key):

        return hash(key) % self.size

    def insert(self, key, value):

        """Insert key-value pair with chaining for collisions"""

        idx = self.\_hash(key)

        for pair in self.table[idx]:

            if pair[0] == key:

                pair[1] = value

                return

        self.table[idx].append([key, value])

    def search(self, key):

        idx = self.\_hash(key)

        for k, v in self.table[idx]:

            if k == key:

                return v

        return None

    def delete(self, key):

        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 display(self):

        for i, bucket in enumerate(self.table):

            print(i, ":", bucket)

**OBSERVATION :**

Uses **chaining** (list of lists) to handle collisions.

insert() → adds/updates key-value pair.

search() → retrieves value by key.

delete() → removes a key-value pair.

Efficient for **fast lookup & insert**

**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. Sample Input Code:

**CODE :**

class Graph:

    def \_\_init\_\_(self):

        self.graph = {}

    def add\_vertex(self, v):

        if v not in self.graph:

            self.graph[v] = []

    def add\_edge(self, v1, v2):

        if v1 in self.graph and v2 in self.graph:

            self.graph[v1].append(v2)

            self.graph[v2].append(v1)  # undirected graph

    def display(self):

        for v in self.graph:

            print(v, "->", self.graph[v])

**OBSERVATION :**

 Graph stored as a **dictionary of adjacency lists**.

 add\_vertex() → adds a new node.

 add\_edge() → connects two nodes.

 display() → prints all connections.

 Efficient for **sparse graphs**.

**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.  
Sample Input Code: class PriorityQueue: pass

**CODE :**

import heapq

class PriorityQueue:

    def \_\_init\_\_(self):

        self.queue = []

    def enqueue(self, priority, item):

        """Add item with priority (min-heap)"""

        heapq.heappush(self.queue, (priority, item))

    def dequeue(self):

        """Remove item with highest priority (lowest number)"""

        if self.queue:

            return heapq.heappop(self.queue)[1]

        return None

    def display(self):

        print("Queue:", self.queue)

# Example

pq = PriorityQueue()

pq.enqueue(2, "Task B")

pq.enqueue(1, "Task A")

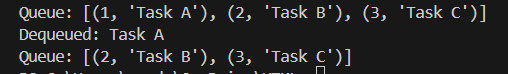
pq.enqueue(3, "Task C")

pq.display()

print("Dequeued:", pq.dequeue())

pq.display()

**OUTPUT :**

****

**OBSERVATION :**

 Uses **min-heap** from heapq.

 Lower priority number = higher priority.

 enqueue() → pushes (priority, item).

 dequeue() → pops item with smallest priority value.

**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. Sample Input Code: class DequeDS: pass

**CODE :**

from collections import deque

class DequeDS:

    def \_\_init\_\_(self):

        self.deque = deque()

    def insert\_front(self, item):

        self.deque.appendleft(item)

    def insert\_rear(self, item):

        self.deque.append(item)

    def remove\_front(self):

        return self.deque.popleft() if self.deque else None

    def remove\_rear(self):

        return self.deque.pop() if self.deque else None

    def display(self):

        print("Deque:", list(self.deque))

# Example

dq = DequeDS()

dq.insert\_rear(10)

dq.insert\_rear(20)

dq.insert\_front(5)

dq.display()

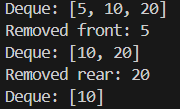
print("Removed front:", dq.remove\_front())

dq.display()

print("Removed rear:", dq.remove\_rear())

dq.display()

**output :**

****

Observation :

 Double-ended queue allows insertion & deletion **from both ends**.

 appendleft() / popleft() → handle **front operations**.

 append() / pop() → handle **rear operations**.

 Useful for **sliding window problems**.

T**ask 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. Sample Input Code:

**CODE :** from collections import deque

class DequeDS:

    def \_\_init\_\_(self):

        self.deque = deque()

    def insert\_front(self, item):

        self.deque.appendleft(item)

    def insert\_rear(self, item):

        self.deque.append(item)

    def remove\_front(self):

        return self.deque.popleft() if self.deque else None

    def remove\_rear(self):

        return self.deque.pop() if self.deque else None

    def display(self):

        print("Deque:", list(self.deque))

# Example

dq = DequeDS()

dq.insert\_rear(10)

dq.insert\_rear(20)

dq.insert\_front(5)

dq.display()

print("Removed front:", dq.remove\_front())

dq.display()def comparison\_table():

    table = """

| Data Structure | Insert | Delete | Search | Access | Order | Notes |

|----------------|--------|--------|--------|--------|-------|-------|

| Stack | O(1) | O(1) | O(n) | O(n) | LIFO | Use list/Deque |

| Queue | O(1) | O(1) | O(n) | O(n) | FIFO | Use deque |

| Linked List | O(1)/O(n) | O(1)/O(n) | O(n) | O(n) | Sequential | Dynamic memory |

| BST (avg) | O(log n) | O(log n) | O(log n) | O(log n) | Sorted | Worst O(n) skewed |

| Hash Table | O(1) avg | O(1) avg | O(1) avg | - | Unordered | Collisions handled |

| Graph (Adj List) | O(1) | O(V+E) | O(V+E) | - | Depends | Good for sparse graphs |

| Priority Queue | O(log n) | O(log n) | O(n) | O(1) | By priority | heapq module |

| Deque | O(1) | O(1) | O(n) | O(n) | Both ends | Flexible queue |

"""

    print(table)

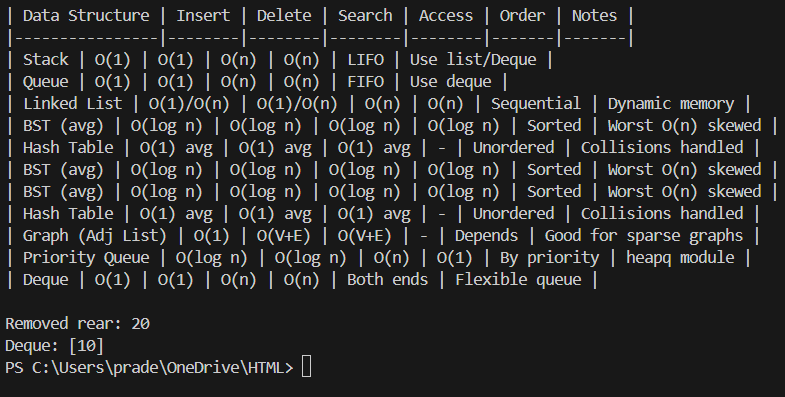
# Example

comparison\_table()

print("Removed rear:", dq.remove\_rear())

dq.display()

**OUTPUT :**



**OBSERVATION :**  Double-ended queue allows insertion & deletion **from both ends**.

 appendleft() / popleft() → handle **front operations**.

 append() / pop() → handle **rear operations**.

 Useful for **sliding window problems**

**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.

**CODE :**

class Queue:

    def \_\_init\_\_(self):

        self.items = []

    def enqueue(self, item):

        """Add a new order at the end"""

        self.items.append(item)

    def dequeue(self):

        """Serve the first order"""

        if not self.items:

            return None

        return self.items.pop(0)

    def display(self):

        print("Orders in queue:", self.items)

# Example

cafeteria = Queue()

cafeteria.enqueue("Order 1: Pizza")

cafeteria.enqueue("Order 2: Burger")

cafeteria.enqueue("Order 3: Sandwich")

cafeteria.display()

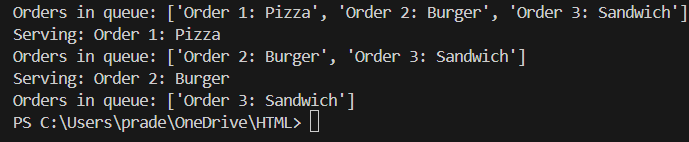
print("Serving:", cafeteria.dequeue())

cafeteria.display()

print("Serving:", cafeteria.dequeue())

cafeteria.display()

**OUTPUT :**

****

**OBSERVATION :**

 Each feature uses the most suitable data structure.

 Attendance → Queue (first in, first out).

 Event Registration → Hash Table (fast search).

 Library → BST (sorted books).

 Bus Scheduling → Graph (routes & stops).

 Cafeteria Orders → Queue (serve in order)