AI ASSISTED CODING LAB ASSIGNMENT 11.1

NAME: T.SAI VARUN

HTK NO: 2403A510C0

BATCH: 05

TASK #1 – STACK IMPLEMENTATION

Prompt:

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

Code:

aiac.py > ...

```python
class Stack:
    """A simple stack implementation using Python lists."""

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

    def push(self, item):
        """Push an item onto the stack."""
        self.items.append(item)

    def pop(self):
        """Remove and return the top item from the stack."""
        if self.is_empty():
            return "Stack Underflow"
        return self.items.pop()

    def peek(self):
        """Return the top item without removing it."""
        return self.items[-1] if not self.is_empty() else None

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

# --- Testing ---
stack = Stack()
stack.push(10)
stack.push(20)
print(stack.peek())    # 20
print(stack.pop())     # 20
print(stack.is_empty())# False
```

AI ASSISTED CODING LAB ASSIGNMENT 11.1

Output:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  PORTS

PS C:\Users\saiva\OneDrive\Desktop\11> & python c:/Users/saiva/OneDrive/Desktop/11/aiac.py
20
20
False
PS C:\Users\saiva\OneDrive\Desktop\11>
```

Observation:

Works correctly with LIFO order.

TASK #2 – QUEUE IMPLEMENTATION

Prompt: **Create a FIFO Queue class.**

Code:

aiac.py > ...

```python
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        if self.size() == 0:
            return "Queue Underflow"
        return self.items.pop(0)
    def peek(self):
        return self.items[0] if self.size() > 0 else None
    def size(self):
        return len(self.items)
q = Queue()
q.enqueue('A')
q.enqueue('B')
print(q.peek())
print(q.dequeue())
print(q.size())

```

AI ASSISTED CODING LAB ASSIGNMENT 11.1

Output:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  PORTS

PS C:\Users\saiva\OneDrive\Desktop\11> & python c:/Users/saiva/OneDrive/Desktop/11/aiac.py
A
A
1
PS C:\Users\saiva\OneDrive\Desktop\11>
```

Observation: Maintains FIFO order.

TASK #3 – LINKED LIST

Prompt: Singly Linked List with insert and display.

Code:

aiac.py > ...

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def insert(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
    def display(self):
        elements = []
        temp = self.head
        while temp:
            elements.append(temp.data)
            temp = temp.next
        return elements
ll = LinkedList()
ll.insert(5)
ll.insert(10)
ll.insert(15)
print(ll.display())

```

AI ASSISTED CODING LAB ASSIGNMENT 11.1

Output:

```
PS C:\Users\saiva\OneDrive\Desktop\11> & python c:/Users/saiva/OneDrive/Desktop/11/aiac.py
[5, 10, 15]
PS C:\Users\saiva\OneDrive\Desktop\11>
```

Observation: Inserts and displays nodes correctly.

TASK #4 – BINARY SEARCH TREE

Prompt: BST with recursive insert and in-order traversal.

Code:

AI ASSISTED CODING LAB ASSIGNMENT 11.1

aiac.py > ...

```python
class BST:
    def __init__(self,value):
        self.value=value
        self.left=None
        self.right=None
    def insert(self,value):
        if value<self.value:
            if self.left:
                self.left.insert(value)
            else:
                self.left=BST(value)
        else:
            if self.right:
                self.right.insert(value)
            else:
                self.right=BST(value)
    def inorder(self):
        elements=[]
        if self.left:
            elements+=self.left.inorder()
        elements.append(self.value)
        if self.right:
            elements+=self.right.inorder()
        return elements
root=BST(10)
root.insert(5)
root.insert(15)
root.insert(2)
print(root.inorder())
```

Output:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  PORTS

PS C:\Users\saiva\OneDrive\Desktop\11> & python c:/Users/saiva/OneDrive/Desktop/11/aiac.py
[2, 5, 10, 15]
PS C:\Users\saiva\OneDrive\Desktop\11>
```

Observation: In-order traversal returns sorted order.

AI ASSISTED CODING LAB ASSIGNMENT 11.1

TASK #5 – HASH TABLE

Prompt: Hash table with chaining.

Code:

```
aiac.py > ...
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):
        index=self._hash(key)
        for i,(k,v) in enumerate(self.table[index]):
            if k==key:
                self.table[index][i]=(key,value)
                return
        self.table[index].append((key,value))
    def search(self,key):
        index=self._hash(key)
        for k,v in self.table[index]:
            if k==key:
                return v
        return None
    def delete(self,key):
        index=self._hash(key)
        self.table[index]=[(k,v) for k,v in self.table[index] if k!=key]
ht=HashTable()
ht.insert('name','Varun')
print(ht.search('name'))
ht.delete('name')
print(ht.search('name'))
```

(variable) table: list[list]

Output:

```
PS C:\Users\saiva\OneDrive\Desktop\11> & python c:/Users/saiva/OneDrive/Desktop/11/aiac.py
Varun
None
PS C:\Users\saiva\OneDrive\Desktop\11>
```

Observation: Collision handled with chaining.

TASK #6 – GRAPH REPRESENTATION

Prompt: Graph using adjacency list.

Code:

AI ASSISTED CODING LAB ASSIGNMENT 11.1

aiac.py > ...

```python
class Graph:
    def __init__(self):
        self.adj_list={}
    def add_vertex(self,vertex):
        if vertex not in self.adj_list:
            self.adj_list[vertex]=[]
    def add_edge(self,v1,v2):
        self.add_vertex(v1)
        self.add_vertex(v2)
        self.adj_list[v1].append(v2)
        self.adj_list[v2].append(v1)
    def display(self):
        for v,e in self.adj_list.items():
            print(f"{v} -> {e}")
g=Graph()
g.add_edge('A','B')
g.add_edge('A','C')
g.display()
```

Output:

```
PS C:\Users\saiva\OneDrive\Desktop\11> & python c:/Users/saiva/OneDrive/Desktop/11/aiac.py
A -> ['B', 'C']
B -> ['A']
C -> ['A']
PS C:\Users\saiva\OneDrive\Desktop\11>
```

Observation:

Adjacency list is memory efficient.

TASK #7 – PRIORITY QUEUE

Prompt: **Priority queue using heapq.**

AI ASSISTED CODING LAB ASSIGNMENT 11.1

Code:

```python
import heapq
class PriorityQueue:
    def __ini(variable) queue: list
        self.queue=[]
    def enqueue(self,priority,item):
        heapq.heappush(self.queue,(priority,item))
    def dequeue(self):
        return heapq.heappop(self.queue)[1] if self.queue else None
    def display(self):
        return [item for _,item in self.queue]
pq=PriorityQueue()
pq.enqueue(2,'B')
pq.enqueue(1,'A')
print(pq.display())
print(pq.dequeue())

```

Output:

```
PS C:\Users\saiva\OneDrive\Desktop\11> & python c:/Users/saiva/OneDrive/Desktop/11/aiac.py
['A', 'B']
A
PS C:\Users\saiva\OneDrive\Desktop\11>
```

Observation: Lower priority numbers dequeued first.

TASK #8 – DEQUE

Prompt: Double-ended queue using collections.deque.

Code:

AI ASSISTED CODING LAB ASSIGNMENT 11.1

aiac.py > ...

```python
from collections import deque
class DequeDS:
    def __init__(self):
        self.dq=deque()
    def add_front(self,item):
        self.dq.appendleft(item)
    def add_rear(self,item):
        self.dq.append(item)
    def remove_front(self):
        return self.dq.popleft() if self.dq else None
    def remove_rear(self):
        return self.dq.pop() if self.dq else None
d=DequeDS()
d.add_rear(1)
d.add_front(2)
print(d.remove_front())
print(d.remove_rear())

```

Output:

```
PS C:\Users\saiva\OneDrive\Desktop\11> & python c:/Users/saiva/OneDrive/Desktop/11/aiac.py
2
1
PS C:\Users\saiva\OneDrive\Desktop\11>
```

Observation: Supports insertion/removal from both ends.

TASK #9 – DATA STRUCTURE COMPARISON TABLE

Prompt: Generate a markdown table comparing stack, queue, linked list, BST, hash table, graph, priority queue, and deque.

CODE:

AI ASSISTED CODING LAB ASSIGNMENT 11.1

aiac.py > ...

```python
# Task #9: Data Structure Comparison Table (Markdown Output)

def data_structure_comparison():
    table = """| Data Structure | Insertion | Deletion | Search | Access | Notes |
|----------------|-----------|----------|--------|--------|-------|
| Stack          | O(1)      | O(1)     | O(n)   | O(n)   | LIFO behavior |
| Queue          | O(1)      | O(1)     | O(n)   | O(n)   | FIFO behavior |
| Linked List    | O(1)/O(n) | O(1)/O(n)| O(n)   | O(n)   | Dynamic size |
| BST            | O(log n)  | O(log n) | O(log n)| O(log n)| Sorted data |
| Hash Table     | O(1)      | O(1)     | O(1)   | N/A    | Uses chaining |
| Graph          | O(1)      | O(1)     | O(V+E) | O(V)   | Models connections |
| Priority Queue | O(log n)  | O(log n) | O(n)   | O(n)   | Uses heap |
| Deque          | O(1)      | O(1)     | O(n)   | O(n)   | Double-ended queue |"""
    return table

# --- Testing ---
print(data_structure_comparison())

```

Output:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  PORTS

PS C:\Users\saiva\OneDrive\Desktop\11> & python c:/Users/saiva/OneDrive/Desktop/11/aiac.py
| Data Structure | Insertion | Deletion | Search | Access | Notes |
|----------------|-----------|----------|--------|--------|-------|
| Stack          | O(1)      | O(1)     | O(n)   | O(n)   | LIFO behavior |
| Queue          | O(1)      | O(1)     | O(n)   | O(n)   | FIFO behavior |
| Linked List    | O(1)/O(n) | O(1)/O(n)| O(n)   | O(n)   | Dynamic size |
| BST            | O(log n)  | O(log n) | O(log n)| O(log n)| Sorted data |
| Hash Table     | O(1)      | O(1)     | O(1)   | N/A    | Uses chaining |
| Graph          | O(1)      | O(1)     | O(V+E) | O(V)   | Models connections |
| Priority Queue | O(log n)  | O(log n) | O(n)   | O(n)   | Uses heap |
| Deque          | O(1)      | O(1)     | O(n)   | O(n)   | Double-ended queue |
PS C:\Users\saiva\OneDrive\Desktop\11>
```

Observation:

1. The generated **Markdown table** provides a clear and concise comparison of common data structures.
2. It highlights the **time complexities** for insertion, deletion, search, and access, making it a quick reference guide.
3. Helps in **choosing the right data structure** based on performance requirements for different operations.
4. Also includes brief **notes** (e.g., LIFO for Stack, FIFO for Queue) to aid conceptual understanding.
5. Output format is compatible with markdown-supported tools like GitHub README files or documentation.

AI ASSISTED CODING LAB ASSIGNMENT 11.1

TASK #10 – REAL-TIME APPLICATION CHALLENGE

Prompt:

Choose the best data structure (Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, or Deque) for each of these:

1. Student attendance tracking
2. Event registration system
3. Library book borrowing
4. Bus scheduling system
5. Cafeteria order queue

Make a table with: feature → chosen data structure → short reason.
Then write Python code for **one** feature, show sample input/output, and give a short observation."

Feature Selection Table:

Feature	Data Structure	Justification
Student Attendance Tracking	Deque	Supports fast insertion/removal from both ends for logs.
Event Registration System	Hash Table	Enables quick search and removal of participants.
Library Book Borrowing	BST	Keeps books sorted by due dates or titles for efficient searches.
Bus Scheduling System	Graph	Represents bus routes and stops effectively.
Cafeteria Order Queue	Queue	Ensures FIFO serving order.

AI ASSISTED CODING LAB ASSIGNMENT 11.1

Code:

Welcome | aiac.py

aiac.py > ...

```
2
3   class CafeteriaQueue:
4       """Queue for cafeteria orders to serve in FIFO order."""
5       def __init__(self):
6           self.orders = []
7
8       def place_order(self, order):
9           """Add an order to the queue."""
10          self.orders.append(order)
11
12      def serve_order(self):
13          """Serve the next order in FIFO order."""
14          if not self.orders:
15              return "No orders to serve."
16          return self.orders.pop(0)
17
18      def display_orders(self):
19          """Display all pending orders."""
20          return self.orders
21
22  # --- Testing ---
23  cafe = CafeteriaQueue()
24  cafe.place_order("Burger")
25  cafe.place_order("Pasta")
26
27  print(cafe.display_orders())  # ['Burger', 'Pasta']
28  print(cafe.serve_order())     # Burger
29  print(cafe.display_orders())  # ['Pasta']
30
```

Output:

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS

```
PS C:\Users\saiva\OneDrive\Desktop\11> & python c:/Users/saiva/OneDrive/Desktop/11/aiac.py
['Burger', 'Pasta']
Burger
['Pasta']
PS C:\Users\saiva\OneDrive\Desktop\11>
```

Observation:

Orders are served FIFO.