Lab 11 – Data Structures with AI: FULL REPORT

Name:J.ABHIRAM

Roll No:2403A51342

Batch:06

Task 1 – Stack

Prompt Used:

AI Prompt: Generate a Stack class with push, pop, peek, is_empty and docstrings.

Code:

```python
class Stack:
    def __init__(self):
        self.items = []  # initialize an empty list for the stack

    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"""
        if not self.is_empty():
            return self.items.pop()
        return "Stack is empty"

    def peek(self):
        """Return the top item without removing it"""
        if not self.is_empty():
            return self.items[-1]
        return "Stack is empty"

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


# Example usage:
s = Stack()
s.push(10)
s.push(20)
print(s.peek())    # 20
print(s.pop())     # 20
print(s.is_empty())  # False
print(s.pop())     # 10
print(s.is_empty())  # True

```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai asss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai asss/task1.py"
20
20
False
10
True
```

Observation:

Observation: Stack operates on LIFO principle, so last pushed is first removed.

Task 2 – Queue

Prompt Used:

AI Prompt: Implement a Queue using list for FIFO operations.

Code:

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

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

    def dequeue(self):
        """Remove and return the item from the front of the queue"""
        if not self.is_empty():
            return self.items.pop(0)
        return "Queue is empty"

    def peek(self):
        """Return the front item without removing it"""
        if not self.is_empty():
            return self.items[0]
        return "Queue is empty"

    def is_empty(self):
        """Check if the queue is empty"""
        return len(self.items) == 0
    def size(self):
        """Return the number of items in the queue"""
        return len(self.items)
q = Queue()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)

print(q.peek())     # 10 (front of queue)
print(q.dequeue())  # 10 (removed from front)
print(q.size())     # 2
print(q.is_empty()) # False
```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai asss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai asss/task2.py"
10
10
2
False
```

Observation:

Observation: Queue follows FIFO, first entered element is removed first.

Task 3 – Linked List

Prompt Used:

AI Prompt: Create a Singly Linked List with insert and display methods.

Code:

task3.py > ...

```python
class Node:
    def __init__(self, data):
        self.data = data      # store data
        self.next = None      # pointer to the next node


class SinglyLinkedList:
    def __init__(self):
        self.head = None      # start with an empty list

    def insert(self, data):
        """Insert a new node at the end of the linked list"""
        new_node = Node(data)
        if self.head is None:     # if list is empty
            self.head = new_node
        else:
            temp = self.head
            while temp.next:      # traverse until last node
                temp = temp.next
            temp.next = new_node  # link new node at the end

    def display(self):
        """Display all nodes in the linked list"""
        if self.head is None:
            print("List is empty")
        else:
            temp = self.head
            while temp:
                print(temp.data, end=" -> ")
                temp = temp.next
            print("None")   # indicate end of list
ll = SinglyLinkedList()
ll.insert(10)
ll.insert(20)
ll.insert(30)

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

Test Cases:

ll = LinkedList()
ll.insert(10)

```
ll.insert(20)
assert ll.display() == [20, 10]
```

Observation:

Observation: Linked list nodes dynamically hold references instead of contiguous memory.

Task 4 – Binary Search Tree

Prompt Used:

AI Prompt: Implement BST with insert and inorder traversal.

Code:

```
class BST:
    class Node:
        def __init__(self, value):
            self.value = value
            self.left = None
            self.right = None

    def __init__(self):
        self.root = None

    def insert(self, value):
        self.root = self._insert(self.root, value)

    def _insert(self, node, value):
        if not node:
            return self.Node(value)
        if value < node.value:
            node.left = self._insert(node.left, value)
        else:
            node.right = self._insert(node.right, value)
        return node

    def inorder(self):
        result = []
        self._inorder(self.root, result)
        return result

    def _inorder(self, node, result):
        if node:
            self._inorder(node.left, result)
            result.append(node.value)
            self._inorder(node.right, result)
```

Test Cases:

```
b = BST()
b.insert(10)
b.insert(5)
b.insert(15)
assert b.inorder() == [5, 10, 15]
```

Observation:

Observation: In-order traversal of BST always returns sorted data.

Task 5 – Hash Table

Prompt Used:

AI Prompt: Implement hash table with chaining.

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):
        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):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
    def delete(self, key):
        index = self._hash(key)
        self.table[index] = [kv for kv in self.table[index] if kv[0] != key]
```

Test Cases:

```
h = HashTable()
h.insert("id", 101)
assert h.search("id") == 101
h.delete("id")
assert h.search("id") == None
```

Observation:

Observation: Hash table uses indexing for fast lookup, chaining avoids collisions.

Task 6 – Graph

Prompt Used:

AI Prompt: Implement graph using adjacency list.

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):
        self.graph[v1].append(v2)
    def display(self):
        return self.graph
```

Test Cases:

```
g = Graph()
g.add_vertex('A')
g.add_vertex('B')
g.add_edge('A','B')
assert g.display() == {'A':['B'],'B':[]}
```

Observation:

Observation: Graph is best when representing networks and relationships.

Task 7 – Priority Queue

Prompt Used:

AI Prompt: Implement priority queue using heapq.

Code:

```
import heapq
class PriorityQueue:
    def __init__(self):
        self.pq = []
    def enqueue(self, priority, value):
        heapq.heappush(self.pq, (priority, value))
    def dequeue(self):
        return heapq.heappop(self.pq) if self.pq else None
```

```
    def display(self):
        return list(self.pq)
```

Test Cases:

```
pq = PriorityQueue()
pq.enqueue(2,"B")
pq.enqueue(1,"A")
assert pq.dequeue()[1] == "A"
```

Observation:

Observation: Lowest priority number is served first when using min-heap.

Task 8 – Deque

Prompt Used:

AI Prompt: Implement deque with insert/remove from both ends.

Code:

```
from collections import deque
class DequeDS:
    def __init__(self):
        self.d = deque()
    def add_front(self, value):
        self.d.appendleft(value)
    def add_rear(self, value):
        self.d.append(value)
    def remove_front(self):
        return self.d.popleft()
    def remove_rear(self):
        return self.d.pop()
```

Test Cases:

```
d = DequeDS()
d.add_front(10)
d.add_rear(20)
assert d.remove_front() == 10
```

Observation:

Observation: Deque supports O(1) insert/remove from both ends.