AI Assistant Coding

Name: CH. Arush Nandhan

HTNO: 2303A51880

Batch-14

Date: 23-02-2026

Assignment - 11.1

solution:

Task #1:

```
class Stack:
    """
    Stack Data Structure (LIFO - Last In First Out)
    Methods:
        push(item)  - Add item to top
        pop()       - Remove and return top item
        peek()      - Return top item without removing
        is_empty()  - Check if stack is empty
    """

    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."""
        if not self.is_empty():
            return self.items.pop()
        return None

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

    def is_empty(self):
        """Return True if stack is empty."""
        return len(self.items) == 0
```

```
s = Stack()
s.push(10)
s.push(20)
print(s.peek())
print(s.pop())
print(s.is_empty())
```

Output:

```
20
20
False
```

Task #2:

```
class Queue:
    """
    Queue Data Structure (FIFO - First In First Out)
    """

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

    def enqueue(self, item):
        """Insert item at the rear."""
        self.items.append(item)

    def dequeue(self):
        """Remove and return front item."""
        if self.items:
            return self.items.pop(0)
        return None

    def peek(self):
        """Return front item without removing."""
        if self.items:
            return self.items[0]
        return None

    def size(self):
        """Return number of items."""
        return len(self.items)
```

```
q = Queue()
q.enqueue(5)
q.enqueue(15)
print(q.peek())
print(q.dequeue())
print(q.size())
```

Output:

```
5
5
1
```

Task #3:

```
class Node:
    """Node of a singly linked list."""

    def __init__(self, data):
        self.data = data
        self.next = None


class LinkedList:
    """
    Singly Linked List
    Methods:
        insert(data)
        display()
    """

    def __init__(self):
        self.head = None

    def insert(self, data):
        """Insert at end of 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):
        """Display all elements."""
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
```

```
ll = LinkedList()
ll.insert(1)
ll.insert(2)
ll.insert(3)
ll.display()
```

Output:

```
1 -> 2 -> 3 -> None
```

Task #4:

```
class BST:
    """
    Binary Search Tree with recursive insert
    and in-order traversal.
    """

    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):
        """Insert value into BST."""
        self.root = self._insert(self.root, value)

    def _insert(self, node, value):
        if node is None:
            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):
        """Perform in-order traversal."""
        self._inorder(self.root)

    def _inorder(self, node):
        if node:
            self._inorder(node.left)
            print(node.value, end=" ")
            self._inorder(node.right)
```

```
bst = BST()
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)

bst.inorder()
```

Output:

```
20 30 40 50 70
```

Task #5:

```
class HashTable:
    """
    Hash Table using chaining for collision handling.
    """

    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

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

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

    def search(self, key):
        """Search for value by key."""
        index = self._hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return None

    def delete(self, key):
        """Delete key-value pair."""
        index = self._hash(key)
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                del self.table[index][i]
                return True
        return False
```

Output:

```
ht = HashTable()

ht.insert("A", 100)
ht.insert("B", 200)

print(ht.search("A"))
ht.delete("A")
print(ht.search("A"))
```

Task #6:

```
class Graph:
    """
    Graph using adjacency list representation.
    """

    def __init__(self):
        self.graph = {}

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

    def add_edge(self, v1, v2):
        """Add edge between v1 and v2."""
        if v1 in self.graph and v2 in self.graph:
            self.graph[v1].append(v2)
            self.graph[v2].append(v1)

    def display(self):
        """Display adjacency list."""
        for vertex in self.graph:
            print(vertex, "->", self.graph[vertex])
```

```
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()
```

Output:

```
A -> ['B', 'C']
B -> ['A']
C -> ['A']
```

Task #7:

```
import heapq

class PriorityQueue:
    """
    Priority Queue using heapq (min-heap).
    """

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

    def enqueue(self, priority, item):
        """Insert item with priority."""
        heapq.heappush(self.heap, (priority, item))

    def dequeue(self):
        """Remove highest priority item."""
        if self.heap:
            return heapq.heappop(self.heap)[1]
        return None

    def display(self):
        """Display queue contents."""
        print(self.heap)
```

```
pq = PriorityQueue()

pq.enqueue(2, "Low Priority")
pq.enqueue(1, "High Priority")
pq.enqueue(3, "Very Low Priority")

print(pq.dequeue())
print(pq.dequeue())
```

Output:

```
High Priority
Low Priority
```

Task #8:

```
from collections import deque

class DequeDS:
    """
    Double-ended Queue implementation.
    """

    def __init__(self):
        self.deque = deque()

    def add_front(self, item):
        """Insert at front."""
        self.deque.appendleft(item)

    def add_rear(self, item):
        """Insert at rear."""
        self.deque.append(item)

    def remove_front(self):
        """Remove from front."""
        if self.deque:
            return self.deque.popleft()
        return None

    def remove_rear(self):
        """Remove from rear."""
        if self.deque:
            return self.deque.pop()
        return None
```

```
dq = DequeDS()

dq.add_front(10)
dq.add_rear(20)
dq.add_front(5)

print(dq.remove_front())
print(dq.remove_rear())
```

Output:

```
5
20
```

Task #9:

```
class CafeteriaQueue:
    """
    Cafeteria Order System using Queue (FIFO).
    """

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

    def place_order(self, student_name):
        """Add order to queue."""
        self.orders.append(student_name)
        print(f"Order placed by {student_name}")

    def serve_order(self):
        """Serve next student."""
        if self.orders:
            student = self.orders.pop(0)
            print(f"Serving order for {student}")
        else:
            print("No pending orders.")

    def display_orders(self):
        """Show all pending orders."""
        print("Pending Orders:", self.orders)


# Example
queue = CafeteriaQueue()
queue.place_order("Alice")
queue.place_order("Bob")
queue.serve_order()
queue.display_orders()
```

```
queue = CafeteriaQueue()

queue.place_order("Alice")
queue.place_order("Bob")
queue.place_order("Charlie")

queue.serve_order()
queue.display_orders()
```

Output:

```
Order placed by Alice
Order placed by Bob
Order placed by Charlie
Serving order for Alice
Pending Orders: ['Bob', 'Charlie']
```

Task #10:

```
class ProductSearch:
    """
    Product Search using Hash Table.
    """

    def __init__(self):
        self.products = {}

    def add_product(self, product_id, name):
        """Add product."""
        self.products[product_id] = name

    def search_product(self, product_id):
        """Search product by ID."""
        return self.products.get(product_id, "Product not found")

    def remove_product(self, product_id):
        """Remove product."""
        if product_id in self.products:
            del self.products[product_id]


# Example
store = ProductSearch()
store.add_product(101, "Laptop")
store.add_product(102, "Smartphone")

print(store.search_product(101))
store.remove_product(102)
```

```
store = ProductSearch()

store.add_product(101, "Laptop")
store.add_product(102, "Smartphone")

print(store.search_product(101))
store.remove_product(102)
print(store.search_product(102))
```

Output:

```
Laptop
Product not found
```