AI ASSISTED CODING
ASSIGNMENT - 11.1
ROLL NO. 2303A510J0

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 clean Python implementation of a `Stack` class with `push`, `pop`, `peek`, and `is_empty` methods. Use a list for storage. Include proper docstrings for each method and handle edge cases (e.g., raise an exception when popping or peeking from an empty stack). Keep the code well-formatted and professional.

Code:

```python
class Stack:
    """A simple Stack implementation using a list as the underlying
storage."""

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

    def push(self, item):
        """
        Add an item to the top of the stack.

        Args:
            item: The item to be added to the stack.
        """
        self._items.append(item)

    def pop(self):
        """
        Remove and return the item from the top of the stack.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """
        Return the item at the top of the stack without removing it.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
```

```python
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]

    def is_empty(self):
        """
        Check if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self._items) == 0

    def __len__(self):
        """Return the number of items in the stack."""
        return len(self._items)

    def __repr__(self):
        """Return a string representation of the stack."""
        return f"Stack({self._items})"


# Sample usage demonstrating all Stack methods
if __name__ == "__main__":
    stack = Stack()

    # Push items
    stack.push(10)
    stack.push(20)
    stack.push(30)
    print(f"Stack after pushes: {stack}")
    print(f"Stack length: {len(stack)}")

    # Peek at top
    print(f"Peek: {stack.peek()}")

    # Pop items
    print(f"Pop: {stack.pop()}")
    print(f"Pop: {stack.pop()}")
```

```python
    print(f"Stack after pops: {stack}")

    # Check if empty
    print(f"Is empty: {stack.is_empty()}")

    # Pop remaining item
    stack.pop()
    print(f"Is empty: {stack.is_empty()}")

    # Attempt to pop from empty stack
    try:
        stack.pop()
    except IndexError as e:
        print(f"Error: {e}")
```

Output:

```
Stack after pushes: Stack([10, 20, 30])
Stack length: 3
Peek: 30
Pop: 30
Pop: 20
Stack after pops: Stack([10])
Is empty: False
Is empty: True
Error: pop from empty stack
```

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

Prompt:

Generate a clean Python implementation of a `Queue` class using a list to maintain FIFO order. Include the methods `enqueue`, `dequeue`, `peek`, and `size`. Add proper docstrings for each method and handle edge cases appropriately (e.g., raise an exception when dequeuing from an empty queue). Keep the code well-formatted and professional, and include a short sample usage demonstrating all methods.

Code:

```python
class Queue:
    """A simple FIFO queue implementation using a list."""

    def __init__(self):
        """Initialize an empty queue."""
        self._items = []

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

        Args:
            item: The item to add to the queue.
```

```python
        """
        self._items.append(item)

    def dequeue(self):
        """
        Remove and return the item at the front of the queue.

        Returns:
            The item at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Cannot dequeue from an empty queue")
        return self._items.pop(0)

    def peek(self):
        """
        Return the item at the front of the queue without removing it.

        Returns:
            The item at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Cannot peek at an empty queue")
        return self._items[0]

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

        Returns:
            The number of items in the queue.
        """
        return len(self._items)
```

```python
    def is_empty(self):
        """
        Check if the queue is empty.

        Returns:
            True if the queue is empty, False otherwise.
        """
        return len(self._items) == 0


# Sample usage
if __name__ == "__main__":
    q = Queue()

    # Enqueue items
    q.enqueue(10)
    q.enqueue(20)
    q.enqueue(30)
    print(f"Queue size: {q.size()}")  # Output: 3

    # Peek at front
    print(f"Front item: {q.peek()}")  # Output: 10

    # Dequeue items
    print(f"Dequeued: {q.dequeue()}")  # Output: 10
    print(f"Dequeued: {q.dequeue()}")  # Output: 20
    print(f"Queue size: {q.size()}")  # Output: 1

    # Dequeue remaining item
    print(f"Dequeued: {q.dequeue()}")  # Output: 30

    # Handle empty queue
    try:
        q.dequeue()
    except IndexError as e:
        print(f"Error: {e}")  # Output: Error: Cannot dequeue from an
empty queue
```

Output:

```
Queue size: 3
Front item: 10
Dequeued: 10
Dequeued: 20
Queue size: 1
Dequeued: 30
Error: Cannot dequeue from an empty queue
```

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 clean Python implementation of a Singly Linked List using separate `Node` and `LinkedList` classes. Include `insert` and `display` methods. Add proper docstrings for all classes and methods, keep the code well-structured and readable, and ensure the implementation works correctly. Include a short sample usage demonstrating insertion and display operations.

Code:

```
"""
Singly Linked List Implementation
A clean implementation of a singly linked list with Node and LinkedList
classes.
"""


class Node:
    """
    Represents a single node in the linked list.

    Attributes:
        data: The value stored in the node.
        next: Reference to the next node in the list.
    """

    def __init__(self, data):
        """
        Initialize a node with given data.

        Args:
            data: The value to store in the node.
        """
        self.data = data
        self.next = None


class LinkedList:
    """
    Represents a singly linked list.
```

```
    Attributes:
        head: Reference to the first node in the list.
    """

    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None

    def insert(self, data):
        """
        Insert a new node with the given data at the end of the list.

        Args:
            data: The value to insert.
        """
        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 insert_at_beginning(self, data):
        """
        Insert a new node with the given data at the beginning of the
list.

        Args:
            data: The value to insert.
        """
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def display(self):
```

```
        """
        Display all elements in the linked list.
        Prints elements in format: element1 -> element2 -> element3 ->
None
        """
        elements = []
        current = self.head

        while current:
            elements.append(str(current.data))
            current = current.next

        print(" -> ".join(elements) + " -> None")


# Sample usage
if __name__ == "__main__":
    # Create a new linked list
    ll = LinkedList()

    # Insert elements at the end
    ll.insert(10)
    ll.insert(20)
    ll.insert(30)
    ll.insert(40)

    # Display the list
    print("Linked List after insertions at end:")
    ll.display()

    # Insert at the beginning
    ll.insert_at_beginning(5)

    print("\nLinked List after insertion at beginning:")
    ll.display()
```

Output:

```
Linked List after insertions at end:
10 -> 20 -> 30 -> 40 -> None

Linked List after insertion at beginning:
5 -> 10 -> 20 -> 30 -> 40 -> 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:
Generate a clean Python implementation of a Binary Search Tree (BST) with recursive `insert` and `inorder_traversal` methods. Include proper docstrings for all classes and methods. Keep the code well-structured, readable, and professional. Ensure the implementation works

correctly and include a short sample usage demonstrating insertion and in-order traversal.

Code:

```
"""
Binary Search Tree (BST) implementation with recursive insert and inorder
traversal.
"""


class Node:
    """Represents a single node in the Binary Search Tree."""

    def __init__(self, value):
        """
        Initialize a node with a given value.

        Args:
            value: The value to store in the node.
        """
        self.value = value
        self.left = None
        self.right = None


class BinarySearchTree:
    """A Binary Search Tree implementation with recursive operations."""

    def __init__(self):
        """Initialize an empty Binary Search Tree."""
        self.root = None

    def insert(self, value):
        """
        Insert a value into the BST.

        Args:
            value: The value to insert.
```

```python
        """
        if self.root is None:
            self.root = Node(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        """
        Recursively insert a value into the BST.

        Args:
            node: The current node being examined.
            value: The value to insert.
        """
        if value < node.value:
            if node.left is None:
                node.left = Node(value)
            else:
                self._insert_recursive(node.left, value)
        else:
            if node.right is None:
                node.right = Node(value)
            else:
                self._insert_recursive(node.right, value)

    def inorder_traversal(self):
        """
        Perform an in-order traversal of the BST (Left, Root, Right).

        Returns:
            A list of values in sorted order.
        """
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        """
        Recursively traverse the BST in in-order sequence.
```

```
        Args:
            node: The current node being visited.
            result: The list to accumulate values.
        """
        if node is not None:
            self._inorder_recursive(node.left, result)
            result.append(node.value)
            self._inorder_recursive(node.right, result)


# Sample usage
if __name__ == "__main__":
    bst = BinarySearchTree()

    # Insert values
    values = [50, 30, 70, 20, 40, 60, 80]
    for value in values:
        bst.insert(value)

    # Perform in-order traversal
    print("In-order traversal:", bst.inorder_traversal())
```

Output:

```
In-order traversal: [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:
Generate a clean Python implementation of a `HashTable` class with `insert`, `search`, and `delete` methods. Use collision handling through chaining (e.g., linked lists or lists within buckets). Include proper docstrings and clear comments explaining the logic. Ensure the code is well-structured, handles edge cases appropriately, and include a short sample usage demonstrating all methods.

Code:
```python
class Node:
    """A node in the linked list for hash table chaining."""
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None


class HashTable:
    """
    A hash table implementation using chaining for collision handling.

    Attributes:
        size (int): The number of buckets in the hash table.
        buckets (list): List of linked lists to store key-value pairs.
    """

    def __init__(self, size=10):
        """
        Initialize the hash table with a given size.
```

```
        Args:
            size (int): Number of buckets (default: 10).
        """
        self.size = size
        self.buckets = [None] * size

    def _hash(self, key):
        """
        Generate a hash value for the given key.

        Args:
            key: The key to hash.

        Returns:
            int: Hash index within the bucket range.
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """
        Insert or update a key-value pair in the hash table.

        Args:
            key: The key to insert.
            value: The value associated with the key.
        """
        index = self._hash(key)
        node = self.buckets[index]

        # Traverse the chain to check if key already exists
        while node:
            if node.key == key:
                node.value = value  # Update existing key
                return
            node = node.next

        # Key not found, insert at the beginning of the chain
        new_node = Node(key, value)
        new_node.next = self.buckets[index]
```

```python
        self.buckets[index] = new_node

    def search(self, key):
        """
        Search for a value by key in the hash table.

        Args:
            key: The key to search for.

        Returns:
            The value if found, None otherwise.
        """
        index = self._hash(key)
        node = self.buckets[index]

        # Traverse the chain to find the key
        while node:
            if node.key == key:
                return node.value
            node = node.next

        return None

    def delete(self, key):
        """
        Delete a key-value pair from the hash table.

        Args:
            key: The key to delete.

        Returns:
            bool: True if deleted, False if key not found.
        """
        index = self._hash(key)
        node = self.buckets[index]
        prev = None

        # Traverse the chain to find the key
        while node:
            if node.key == key:
```

```python
                if prev:
                    prev.next = node.next  # Remove from middle or end
                else:
                    self.buckets[index] = node.next  # Remove from
beginning
                return True
            prev = node
            node = node.next

        return False

    def display(self):
        """Display all key-value pairs in the hash table."""
        for i, node in enumerate(self.buckets):
            print(f"Bucket {i}: ", end="")
            while node:
                print(f"({node.key}: {node.value})", end=" -> ")
                node = node.next
            print("None")


# Sample usage
if __name__ == "__main__":
    ht = HashTable(size=5)

    # Insert operations
    ht.insert("name", "Alice")
    ht.insert("age", 25)
    ht.insert("city", "New York")
    ht.insert("job", "Engineer")

    print("Hash Table after insertions:")
    ht.display()

    # Search operations
    print(f"\nSearch 'name': {ht.search('name')}")
    print(f"Search 'age': {ht.search('age')}")
    print(f"Search 'unknown': {ht.search('unknown')}")

    # Update operation
```

```
    ht.insert("age", 26)
    print(f"\nSearch 'age' after update: {ht.search('age')}")

    # Delete operations
    print(f"\nDeleting 'city': {ht.delete('city')}")
    print(f"Deleting 'unknown': {ht.delete('unknown')}")

    print("\nHash Table after deletions:")
    ht.display()
```

Output:

```
Hash Table after insertions:
Bucket 0: (job: Engineer) -> (city: New York) ->
None
Bucket 1: (name: Alice) -> None
Bucket 2: None
Bucket 3: (age: 25) -> None
Bucket 4: None

Search 'name': Alice
Search 'age': 25
Search 'unknown': None

Search 'age' after update: 26

Deleting 'city': True
Deleting 'unknown': False

Hash Table after deletions:
Bucket 0: (job: Engineer) -> None
```

```
Bucket 1: (name: Alice) -> None
Bucket 2: None
Bucket 3: (age: 26) -> None
Bucket 4: None
```

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:

Generate a clean Python implementation of a `Graph` class using an adjacency list representation. Include methods to add vertices, add edges, and display connections. Add proper docstrings for all methods, keep the code well-structured and readable, and handle basic edge cases appropriately. Include a short sample usage demonstrating all methods.

Code:

```
class Graph:
    """A graph implementation using adjacency list representation."""
```

```python
    def __init__(self):
        """Initialize an empty graph."""
        self.adjacency_list = {}

    def add_vertex(self, vertex):
        """
        Add a vertex to the graph.

        Args:
            vertex: The vertex to add (hashable type)
        """
        if vertex not in self.adjacency_list:
            self.adjacency_list[vertex] = []

    def add_edge(self, vertex1, vertex2, directed=False):
        """
        Add an edge between two vertices.

        Args:
            vertex1: The first vertex
            vertex2: The second vertex
            directed: If True, edge is directed (vertex1 -> vertex2).
                      If False, edge is undirected (bidirectional).
        """
        # Add vertices if they don't exist
        self.add_vertex(vertex1)
        self.add_vertex(vertex2)

        # Add edge
        if vertex2 not in self.adjacency_list[vertex1]:
            self.adjacency_list[vertex1].append(vertex2)

        # Add reverse edge if undirected
        if not directed and vertex1 not in self.adjacency_list[vertex2]:
            self.adjacency_list[vertex2].append(vertex1)

    def display(self):
        """Display all vertices and their connections."""
        if not self.adjacency_list:
            print("Graph is empty")
```

```python
            return

        for vertex, neighbors in self.adjacency_list.items():
            print(f"{vertex}: {neighbors}")

    def get_neighbors(self, vertex):
        """
        Get all neighbors of a vertex.

        Args:
            vertex: The vertex to query

        Returns:
            List of neighbors, or None if vertex doesn't exist
        """
        return self.adjacency_list.get(vertex, None)


# Sample usage
if __name__ == "__main__":
    # Create a graph
    graph = Graph()

    # Add vertices
    graph.add_vertex("A")
    graph.add_vertex("B")
    graph.add_vertex("C")

    # Add edges (undirected)
    graph.add_edge("A", "B")
    graph.add_edge("B", "C")
    graph.add_edge("A", "C")

    # Display the graph
    print("Graph connections:")
    graph.display()

    # Get neighbors
    print(f"\nNeighbors of A: {graph.get_neighbors('A')}")
    print(f"Neighbors of B: {graph.get_neighbors('B')}")
```

```python
# Add directed edge
graph.add_edge("C", "A", directed=True)
print("\nAfter adding directed edge C->A:")
graph.display()
```

Output:

```
Graph connections:
A: ['B', 'C']
B: ['A', 'C']
C: ['B', 'A']

Neighbors of A: ['B', 'C']
Neighbors of B: ['A', 'C']

After adding directed edge C->A:
A: ['B', 'C']
B: ['A', 'C']
C: ['B', 'A']
```

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:
Generate a clean Python implementation of a `PriorityQueue` class using the `heapq` module. Include `enqueue` (with priority), `dequeue` (returning the highest priority element), and `display` methods. Add proper docstrings for all methods, keep the code well-structured and readable, and handle edge cases appropriately. Include a short sample usage demonstrating all methods.

Code:
```
import heapq

class PriorityQueue:
    """
    A priority queue implementation using Python's heapq module.
    Lower priority values are dequeued first (min-heap).
    """

    def __init__(self):
        """Initialize an empty priority queue."""
        self.heap = []

    def enqueue(self, item, priority):
        """
        Add an item to the queue with a given priority.
```

```python
        Args:
            item: The data to store in the queue.
            priority: The priority value (lower values = higher priority).
        """
        heapq.heappush(self.heap, (priority, item))

    def dequeue(self):
        """
        Remove and return the highest priority element.

        Returns:
            The item with the lowest priority value.

        Raises:
            IndexError: If the queue is empty.
        """
        if not self.heap:
            raise IndexError("Cannot dequeue from an empty queue")
        priority, item = heapq.heappop(self.heap)
        return item

    def display(self):
        """Display all items in the queue with their priorities."""
        if not self.heap:
            print("Priority Queue is empty")
            return
        print("Priority Queue (priority, item):")
        for priority, item in sorted(self.heap):
            print(f"  Priority {priority}: {item}")

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

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


# Sample usage
```

```python
if __name__ == "__main__":
    pq = PriorityQueue()

    # Enqueue items with priorities
    pq.enqueue("Task A", 3)
    pq.enqueue("Task B", 1)
    pq.enqueue("Task C", 2)
    pq.enqueue("Task D", 1)

    print("Initial Queue:")
    pq.display()

    # Dequeue items
    print(f"\nDequeued: {pq.dequeue()}")
    print(f"Dequeued: {pq.dequeue()}")

    print(f"\nQueue after dequeuing:")
    pq.display()

    print(f"\nQueue size: {pq.size()}")
    print(f"Is empty: {pq.is_empty()}")
```

Output:

```
Initial Queue:
Priority Queue (priority, item):
  Priority 1: Task B
  Priority 1: Task D
  Priority 2: Task C
  Priority 3: Task A

Dequeued: Task B
Dequeued: Task D

Queue after dequeuing:
```

```
Priority Queue (priority, item):
  Priority 2: Task C
  Priority 3: Task A

Queue size: 2
Is empty: False
```

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:
Generate a clean Python implementation of a `DequeDS` class using `collections.deque`. Include methods to insert and remove elements from both ends. Add proper docstrings for all methods, keep the code well-structured and readable, and handle edge cases appropriately. Include a short sample usage demonstrating all operations.

Code:

```python
from collections import deque

"""
Deque Data Structure Implementation using collections.deque
A double-ended queue that allows insertion and removal from both ends.
"""



class DequeDS:
    """A wrapper class around collections.deque with standard
operations."""

    def __init__(self):
        """Initialize an empty deque."""
        self._items = deque()

    def append_right(self, item):
        """
        Add an element to the right (rear) end of the deque.

        Args:
            item: The element to add.
        """
        self._items.append(item)

    def append_left(self, item):
        """
        Add an element to the left (front) end of the deque.

        Args:
            item: The element to add.
        """
        self._items.appendleft(item)

    def remove_right(self):
        """
        Remove and return an element from the right (rear) end.

        Returns:
```

```
            The removed element.

        Raises:
            IndexError: If the deque is empty.
        """
        if self.is_empty():
            raise IndexError("Cannot remove from an empty deque")
        return self._items.pop()

    def remove_left(self):
        """
        Remove and return an element from the left (front) end.

        Returns:
            The removed element.

        Raises:
            IndexError: If the deque is empty.
        """
        if self.is_empty():
            raise IndexError("Cannot remove from an empty deque")
        return self._items.popleft()

    def is_empty(self):
        """
        Check if the deque is empty.

        Returns:
            True if empty, False otherwise.
        """
        return len(self._items) == 0

    def size(self):
        """
        Get the number of elements in the deque.

        Returns:
            The size of the deque.
        """
        return len(self._items)
```

```
    def peek_left(self):
        """
        View the leftmost element without removing it.

        Returns:
            The leftmost element or None if empty.
        """
        return self._items[0] if not self.is_empty() else None

    def peek_right(self):
        """
        View the rightmost element without removing it.

        Returns:
            The rightmost element or None if empty.
        """
        return self._items[-1] if not self.is_empty() else None

    def display(self):
        """Display all elements in the deque from left to right."""
        print(f"Deque: {list(self._items)}")


# Sample usage
if __name__ == "__main__":
    dq = DequeDS()

    print("--- Adding elements ---")
    dq.append_right(10)
    dq.append_right(20)
    dq.append_left(5)
    dq.append_left(1)
    dq.display()
    print(f"Size: {dq.size()}\n")

    print("--- Peeking elements ---")
    print(f"Left: {dq.peek_left()}, Right: {dq.peek_right()}\n")

    print("--- Removing from left ---")
```

```python
    print(f"Removed: {dq.remove_left()}")
    dq.display()

    print("\n--- Removing from right ---")
    print(f"Removed: {dq.remove_right()}")
    dq.display()

    print(f"\nFinal size: {dq.size()}")
    print(f"Is empty: {dq.is_empty()}")
```

Output:

```
--- Adding elements ---
Deque: [1, 5, 10, 20]
Size: 4

--- Peeking elements ---
Left: 1, Right: 20

--- Removing from left ---
Removed: 1
Deque: [5, 10, 20]

--- Removing from right ---
Removed: 20
Deque: [5, 10]

Final size: 2
Is empty: False
```

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.

Feature	Chosen Data Structure	Justification
Student Attendance Tracking	Deque	A Deque allows efficient insertion and removal from both ends, making it suitable for recording student entry and exit logs in real time. It preserves order while allowing flexible operations for

		managing daily attendance records.
Event Registration System	Hash Table	A Hash Table provides fast insertion, search, and deletion in average constant time. It is ideal for managing participant registrations where quick lookup and removal based on student ID or name is required.
Library Book Borrowing	Binary Search Tree (BST)	A BST maintains elements in sorted order and allows efficient searching, insertion, and deletion. It is suitable for organizing books by ID or due date and managing borrowing records systematically.
Bus Scheduling System	Graph	A Graph effectively represents bus stops as vertices and routes as edges. It models connectivity between stops and supports traversal operations for route management.

Cafeteria Order Queue	Queue	A Queue follows the FIFO principle, ensuring students are served in the order they arrive. It efficiently manages order processing using enqueue and dequeue operations.

Prompt:
Generate a clean and well-structured Python implementation of a Bus Scheduling System using a `Graph` data structure with an adjacency list representation. Model bus stops as vertices and routes as edges. Include methods to add bus stops, add routes between stops, and display all routes. Use proper class design, include clear comments and docstrings for all methods, and ensure the program is readable and professional. Handle basic edge cases appropriately. Include a short sample usage demonstrating the addition of stops and routes and displaying the connections.

Code:
```
"""
Bus Scheduling System using Graph Data Structure

This module implements a bus scheduling system where bus stops are
represented
```

```
as vertices and routes between stops are represented as edges in a graph.
"""


class BusSchedulingSystem:
    """
    A graph-based bus scheduling system using adjacency list
representation.

    Attributes:
        stops (dict): Dictionary mapping stop names to their adjacency
lists.
    """

    def __init__(self):
        """Initialize an empty bus scheduling system."""
        self.stops = {}

    def add_stop(self, stop_name):
        """
        Add a new bus stop to the system.

        Args:
            stop_name (str): The name of the bus stop to add.

        Returns:
            bool: True if stop was added, False if it already exists.
        """
        if stop_name in self.stops:
            print(f"Stop '{stop_name}' already exists.")
            return False

        self.stops[stop_name] = []
        print(f"Stop '{stop_name}' added successfully.")
        return True

    def add_route(self, from_stop, to_stop, distance=None):
        """
        Add a bidirectional route between two bus stops.
```

```
        Args:
            from_stop (str): The starting bus stop.
            to_stop (str): The destination bus stop.
            distance (float, optional): Distance between stops in km.

        Returns:
            bool: True if route was added, False if stops don't exist.
        """
        if from_stop not in self.stops or to_stop not in self.stops:
            print("Error: One or both stops do not exist.")
            return False

        if from_stop == to_stop:
            print("Error: Cannot add route from a stop to itself.")
            return False

        route_info = {"stop": to_stop, "distance": distance}

        # Check if route already exists
        if any(r["stop"] == to_stop for r in self.stops[from_stop]):
            print(f"Route from '{from_stop}' to '{to_stop}' already
exists.")
            return False

        self.stops[from_stop].append(route_info)
        self.stops[to_stop].append({"stop": from_stop, "distance":
distance})

        print(f"Route added: '{from_stop}' <-> '{to_stop}'" +
              (f" ({distance} km)" if distance else ""))
        return True

    def display_routes(self):
        """Display all routes in the bus scheduling system."""
        if not self.stops:
            print("No stops in the system.")
            return

        print("\n" + "="*50)
        print("BUS SCHEDULING SYSTEM - ALL ROUTES")
```

```
        print("="*50)

        for stop, routes in sorted(self.stops.items()):
            print(f"\nFrom '{stop}':")
            if routes:
                for route in routes:
                    distance_str = f" - {route['distance']} km" if
route['distance'] else ""
                    print(f"  → {route['stop']}{distance_str}")
            else:
                print("  (No routes)")
        print("="*50 + "\n")

    def get_stop_connections(self, stop_name):
        """
        Get all direct connections from a specific stop.

        Args:
            stop_name (str): The name of the bus stop.

        Returns:
            list: List of connected stops, or None if stop doesn't exist.
        """
        if stop_name not in self.stops:
            print(f"Stop '{stop_name}' does not exist.")
            return None

        return [route["stop"] for route in self.stops[stop_name]]


# Sample Usage
if __name__ == "__main__":
    # Create a bus scheduling system
    system = BusSchedulingSystem()

    # Add bus stops
    print("Adding bus stops...\n")
    system.add_stop("Central Station")
    system.add_stop("Airport")
    system.add_stop("Bus Terminal")
```

```python
    system.add_stop("Downtown")
    system.add_stop("University")

    # Add routes between stops
    print("\nAdding routes...\n")
    system.add_route("Central Station", "Airport", 15.5)
    system.add_route("Central Station", "Downtown", 5.0)
    system.add_route("Airport", "Bus Terminal", 8.3)
    system.add_route("Downtown", "University", 3.2)
    system.add_route("Bus Terminal", "University", 12.1)

    # Display all routes
    system.display_routes()

    # Get specific stop connections
    print("Connections from 'Central Station':")
    connections = system.get_stop_connections("Central Station")
    if connections:
        for conn in connections:
            print(f"  → {conn}")
```

Output:

```
Adding bus stops...

Stop 'Central Station' added successfully.
Stop 'Airport' added successfully.
Stop 'Bus Terminal' added successfully.
Stop 'Downtown' added successfully.
Stop 'University' added successfully.

Adding routes...

Route added: 'Central Station' <-> 'Airport' (15.5 km)
```

```
Route added: 'Central Station' <-> 'Downtown' (5.0 km)
Route added: 'Airport' <-> 'Bus Terminal' (8.3 km)
Route added: 'Downtown' <-> 'University' (3.2 km)
Route added: 'Bus Terminal' <-> 'University' (12.1 km)

==================================================
BUS SCHEDULING SYSTEM - ALL ROUTES
==================================================

From 'Airport':
  → Central Station - 15.5 km
  → Bus Terminal - 8.3 km

From 'Bus Terminal':
  → Airport - 8.3 km
  → University - 12.1 km

From 'Central Station':
  → Airport - 15.5 km
  → Downtown - 5.0 km

From 'Downtown':
  → Central Station - 5.0 km
  → University - 3.2 km

From 'University':
  → Downtown - 3.2 km
```

```
  → Bus Terminal - 12.1 km
==================================================

Connections from 'Central Station':
  → Airport
  → Downtown
```

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.

Feature	Chosen Data Structure	Justification

Shopping Cart Management	Linked List	A Linked List allows dynamic addition and removal of products without requiring resizing like arrays. Since shopping carts frequently change as users add or remove items, a linked list provides efficient insert and delete operations. It is suitable for managing a flexible and dynamic collection of products.
Order Processing System	Queue	A Queue follows the First-In-First-Out (FIFO) principle, ensuring that orders are processed in the exact order they are placed. This maintains fairness and consistency in order handling. It efficiently supports enqueue and dequeue operations for order management.
Top-Selling Products Tracker	Priority Queue	A Priority Queue allows products to be ranked and retrieved based on their sales count. The product with the highest sales can be accessed efficiently. This makes it ideal for dynamically maintaining and displaying top-selling products.

Product Search Engine	**Hash Table**	**A Hash Table provides fast average-case lookup, insertion, and deletion operations. Since products are searched using unique product IDs, a hash table ensures efficient and direct access. This significantly improves search performance in large inventories.**
Delivery Route Planning	**Graph**	**A Graph effectively models warehouses and delivery locations as vertices and the routes between them as edges. It allows representation of connections and supports traversal or pathfinding algorithms. This makes it ideal for route planning and logistics management.**

Prompt:
Generate a clean and well-structured Python implementation of a Delivery Route Planning system using a `Graph` data structure with an adjacency list representation. Model warehouses and delivery locations as vertices and routes between them as edges. Include methods to add locations, add routes, and display all connections. Use proper class

design with clear comments and docstrings for all methods. Ensure the program is readable, professional, and handles basic edge cases appropriately. Include a short sample usage demonstrating the addition of locations, routes, and displaying the delivery network.

Code:

```
"""
Delivery Route Planning System using Graph Data Structure.

This module implements a graph-based delivery route planner that manages
warehouses and delivery locations as vertices and routes as edges.
"""


class DeliveryNetwork:
    """
    A graph-based system for managing delivery routes and locations.

    Uses an adjacency list representation to store the network of
    warehouses and delivery locations with their connections.
    """

    def __init__(self):
        """Initialize an empty delivery network."""
        self.graph = {}

    def add_location(self, location_name):
        """
        Add a new location (warehouse or delivery point) to the network.

        Args:
            location_name (str): Name of the location to add.

        Returns:
            bool: True if location added, False if already exists.
        """
```

```python
        if location_name in self.graph:
            print(f"Location '{location_name}' already exists.")
            return False

        self.graph[location_name] = []
        print(f"Location '{location_name}' added successfully.")
        return True

    def add_route(self, source, destination, distance=None):
        """
        Add a route between two locations.

        Args:
            source (str): Starting location.
            destination (str): Ending location.
            distance (float, optional): Distance between locations.

        Returns:
            bool: True if route added, False if locations don't exist.
        """
        if source not in self.graph or destination not in self.graph:
            print(f"Error: One or both locations don't exist.")
            return False

        if source == destination:
            print(f"Error: Cannot add route from location to itself.")
            return False

        route_info = (destination, distance) if distance else destination

        if route_info not in self.graph[source]:
            self.graph[source].append(route_info)
            print(f"Route added: {source} -> {destination}")
            return True

        print(f"Route already exists: {source} -> {destination}")
        return False

    def display_network(self):
        """Display all locations and their connections in the network."""
```

```
        if not self.graph:
            print("The delivery network is empty.")
            return

        print("\n" + "=" * 60)
        print("DELIVERY NETWORK STRUCTURE")
        print("=" * 60)

        for location, routes in self.graph.items():
            if not routes:
                print(f"{location}: No routes")
            else:
                connections = []
                for route in routes:
                    if isinstance(route, tuple):
                        destination, distance = route
                        connections.append(f"{destination} ({distance}
km)")
                    else:
                        connections.append(route)
                print(f"{location}: -> {', '.join(connections)}")

        print("=" * 60 + "\n")

    def get_routes_from(self, location):
        """
        Get all routes from a specific location.

        Args:
            location (str): The source location.

        Returns:
            list: Routes from the location, or empty list if not found.
        """
        if location not in self.graph:
            print(f"Location '{location}' not found.")
            return []

        return self.graph[location]
```

```python
# Sample Usage
if __name__ == "__main__":
    # Initialize the delivery network
    network = DeliveryNetwork()

    # Add locations (warehouses and delivery points)
    network.add_location("Main Warehouse")
    network.add_location("Downtown Hub")
    network.add_location("Airport Terminal")
    network.add_location("Neighborhood Store A")
    network.add_location("Neighborhood Store B")

    # Add routes with distances
    network.add_route("Main Warehouse", "Downtown Hub", 15.5)
    network.add_route("Main Warehouse", "Airport Terminal", 25.0)
    network.add_route("Downtown Hub", "Neighborhood Store A", 8.3)
    network.add_route("Downtown Hub", "Neighborhood Store B", 12.1)
    network.add_route("Airport Terminal", "Neighborhood Store B", 20.0)

    # Display the delivery network
    network.display_network()

    # Retrieve and display routes from specific location
    print("Routes from 'Main Warehouse':")
    routes = network.get_routes_from("Main Warehouse")
    for route in routes:
        if isinstance(route, tuple):
            print(f"  -> {route[0]} ({route[1]} km)")
        else:
            print(f"  -> {route}")
```

Output:

```
Location 'Main Warehouse' added successfully.
Location 'Downtown Hub' added successfully.
Location 'Airport Terminal' added successfully.
Location 'Neighborhood Store A' added successfully.
Location 'Neighborhood Store B' added successfully.
Route added: Main Warehouse -> Downtown Hub
Route added: Main Warehouse -> Airport Terminal
Route added: Downtown Hub -> Neighborhood Store A
Route added: Downtown Hub -> Neighborhood Store B
Route added: Airport Terminal -> Neighborhood Store B

=========================================================
===
DELIVERY NETWORK STRUCTURE
=========================================================
===
Main Warehouse: -> Downtown Hub (15.5 km), Airport
Terminal (25.0 km)
Downtown Hub: -> Neighborhood Store A (8.3 km),
Neighborhood Store B (12.1 km)
Airport Terminal: -> Neighborhood Store B (20.0 km)
Neighborhood Store A: No routes
Neighborhood Store B: No routes
=========================================================
===
```

```
Routes from 'Main Warehouse':
  -> Downtown Hub (15.5 km)
  -> Airport Terminal (25.0 km)
```