

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		Assignment Type: Lab	Academic Year:2025-2026
Course Coordinator Name		Venkataramana Veeramsetty	
Instructor(s) Name		Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)	
Course Code	24CS002PC215	Course Title	AI Assisted Coding
Year/Sem	II/I	Regulation	R24
Date and Day of Assignment	Week6 - Monday	Time(s)	
Duration	2 Hours	Applicable to Batches	
AssignmentNumber:11.1(Present assignment number)/24(Total number of assignments)			
Q.No.	Question		
1	Lab 11 – Data Structures with AI: Implementing Fundamental Structures Lab Objectives <ul style="list-style-type: none"> • Use AI to assist in designing and implementing fundamental data structures in Python. • Learn how to prompt AI for structure creation, optimization, and documentation. • Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, Graphs, and Hash Tables. 	Expected Time to complete Week6 - Monday	

	<ul style="list-style-type: none"> Enhance code quality with AI-generated comments and performance suggestions. 	
	<p>Task Description #1 – Stack Implementation</p> <p>Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.</p> <p>Sample Input Code:</p> <pre>class Stack: pass</pre> <p>Expected Output:</p> <ul style="list-style-type: none"> A functional stack implementation with all required methods and docstrings. Prompt: Generate a Python class Stack that implements push, pop, peek, and is_empty methods using a list internally. Include docstrings for each method, and raise appropriate exceptions for invalid operations (like popping from empty stack). Also generate 3 assert test cases to validate stack operations and print success messages if they pass. <pre>▶ class Stack: def __init__(self): """Initializes an empty stack.""" self._items = [] def push(self, item): """Adds an item to the top of the stack.""" self._items.append(item) def pop(self): """Removes and returns the item from the top of the stack. Raises an IndexError if the stack is empty. """ if not self.is_empty(): return self._items.pop() else: raise IndexError("pop from empty stack") def peek(self): """Returns the item at the top of the stack without removing it. Raises an IndexError if the stack is empty. """ if not self.is_empty(): return self._items[-1] else: raise IndexError("peek from empty stack") def is_empty(self): """Returns True if the stack is empty, False otherwise.""" return len(self._items) == 0 def __len__(self): """Returns the number of items in the stack.""" return len(self._items) def __str__(self): """Returns a string representation of the stack.""" return str(self._items)</pre>	

```

▶ # Test Case 1: Push and Peek
stack1 = Stack()
stack1.push(1)
stack1.push(2)
assert stack1.peek() == 2, "Test Case 1 Failed: Push and Peek"
print("Test Case 1 Passed: Push and Peek")

# Test Case 2: Push and Pop
stack2 = Stack()
stack2.push(10)
stack2.push(20)
assert stack2.pop() == 20, "Test Case 2 Failed: Push and Pop"
assert stack2.pop() == 10, "Test Case 2 Failed: Push and Pop"
assert stack2.is_empty(), "Test Case 2 Failed: Push and Pop"
print("Test Case 2 Passed: Push and Pop")

# Test Case 3: Pop from Empty Stack (expecting an error)
stack3 = Stack()
try:
    stack3.pop()
    assert False, "Test Case 3 Failed: Pop from Empty Stack (No exception raised)"
except IndexError:
    assert True, "Test Case 3 Failed: Pop from Empty Stack (Wrong exception raised)"
print("Test Case 3 Passed: Pop from Empty Stack")

```

→ Test Case 1 Passed: Push and Peek
 Test Case 2 Passed: Push and Pop
 Test Case 3 Passed: 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 Python class Queue that implements enqueue, dequeue, peek, and size methods using a list internally.
 Make sure it follows FIFO behavior.
 Add docstrings for all methods, handle edge cases (like dequeuing from empty queue), and create 3 assert test cases to verify functionality.

```

▶ class Queue:
    def __init__(self):
        """Initializes an empty queue."""
        self._items = []
    def enqueue(self, item):
        """Adds an item to the rear of the queue."""
        self._items.append(item)
    def dequeue(self):
        """Removes and returns the item from the front of the queue.
        Raises an IndexError if the queue is empty."""
        if not self.is_empty():
            return self._items.pop(0) # Remove from the beginning for FIFO
        else:
            raise IndexError("dequeue from empty queue")
    def peek(self):
        """Returns the item at the front of the queue without removing it.
        Raises an IndexError if the queue is empty."""
        if not self.is_empty():
            return self._items[0]
        else:
            raise IndexError("peek from empty queue")
    def is_empty(self):
        """Returns True if the queue is empty, False otherwise."""
        return len(self._items) == 0
    def size(self):
        """Returns the number of items in the queue."""
        return len(self._items)
    def __len__(self):
        """Returns the number of items in the queue."""
        return len(self._items)
    def __str__(self):
        """Returns a string representation of the queue."""
        return str(self._items)

# Test Case 1: Enqueue and Dequeue
queue1 = Queue()
queue1.enqueue(100)
queue1.enqueue(200)
assert queue1.dequeue() == 100, "Test Case 1 Failed: Enqueue and Dequeue (First item)"
assert queue1.dequeue() == 200, "Test Case 1 Failed: Enqueue and Dequeue (Second item)"
assert queue1.is_empty(), "Test Case 1 Failed: Enqueue and Dequeue (Queue not empty)"
print("Test Case 1 Passed: Enqueue and Dequeue")

# Test Case 2: Enqueue, Peek, and Size
queue2 = Queue()
queue2.enqueue('a')
queue2.enqueue('b')
assert queue2.peek() == 'a', "Test Case 2 Failed: Peek"
assert queue2.size() == 2, "Test Case 2 Failed: Size"
assert queue2.dequeue() == 'a', "Test Case 2 Failed: Dequeue after peek"
assert queue2.size() == 1, "Test Case 2 Failed: Size after dequeue"
print("Test Case 2 Passed: Enqueue, Peek, and Size")

# Test Case 3: Dequeue from Empty Queue (expecting an error)
queue3 = Queue()
try:
    queue3.dequeue()
    assert False, "Test Case 3 Failed: Dequeue from Empty Queue (No exception raised)"
except IndexError:
    assert True, "Test Case 3 Failed: Dequeue from Empty Queue (Wrong exception raised)"
print("Test Case 3 Passed: Dequeue from Empty Queue")

```

Test Case 1 Passed: Enqueue and Dequeue
Test Case 2 Passed: Enqueue, Peek, and Size
Test Case 3 Passed: Dequeue from 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 Python implementation of a singly linked list with a Node class and a LinkedList class.
 Implement methods for inserting nodes at the end and displaying the list.
 Add docstrings and provide 3 assert test cases to validate insertion and display output.

```
class Node:  

    """Represents a node in a singly linked list."""  

    def __init__(self, data=None):  

        self.data = data  

        self.next = None  

class LinkedList:  

    """Represents a singly linked list."""  

    def __init__(self):  

        self.head = None  

    def insert_at_end(self, data):  

        """Inserts a new node with the given data at the end of the linked list."""  

        new_node = Node(data)  

        if self.head is None:  

            self.head = new_node  

        return  

        last_node = self.head  

        while last_node.next:  

            last_node = last_node.next  

        last_node.next = new_node  

    def display(self):  

        """Displays the data of each node in the linked list."""  

        elements = []  

        current_node = self.head  

        while current_node:  

            elements.append(current_node.data)  

            current_node = current_node.next  

        return elements  

    def __str__(self):  

        """Returns a string representation of the linked list."""  

        return str(self.display())  

  

# Test Case 1: Insert at end and display with multiple nodes  

linked_list1 = LinkedList()  

linked_list1.insert_at_end(1)  

linked_list1.insert_at_end(2)  

linked_list1.insert_at_end(3)  

assert linked_list1.display() == [1, 2, 3], "Test Case 1 Failed: Insert at end and display (multiple nodes)"  

print("Test Case 1 Passed: Insert at end and display with multiple nodes")  

  

# Test Case 2: Insert at end and display with a single node  

linked_list2 = LinkedList()  

linked_list2.insert_at_end(100)  

assert linked_list2.display() == [100], "Test Case 2 Failed: Insert at end and display (single node)"  

print("Test Case 2 Passed: Insert at end and display with a single node")  

  

# Test Case 3: Display an empty list  

linked_list3 = LinkedList()  

assert linked_list3.display() == [], "Test Case 3 Failed: Display empty list"  

print("Test Case 3 Passed: Display an empty list")  

  

Test Case 1 Passed: Insert at end and display with multiple nodes  

Test Case 2 Passed: Insert at end and display with a single node  

Test Case 3 Passed: Display an empty list
```

Task Description #4 – Binary Search Tree (BST)

	<p>Task: Use AI to create a BST with insert and in-order traversal methods.</p> <p>Sample Input Code:</p> <pre>class BST: pass</pre> <p>Expected Output:</p> <ul style="list-style-type: none"> • BST implementation with recursive insert and traversal methods. • Prompt: Generate a Python class BST that supports recursive insertion and in-order traversal. <p>Use a Node class for tree nodes.</p> <p>Include docstrings and write 3 assert test cases to confirm that insertion creates a correct BST and traversal gives sorted output.</p> <pre>class Node: """Represents a node in a Binary Search Tree.""" def __init__(self, key): self.key = key self.left = None self.right = None class BST: """Represents a Binary Search Tree.""" def __init__(self): self.root = None def insert(self, key): """Inserts a new key into the BST.""" self.root = self._insert_recursive(self.root, key) def _insert_recursive(self, root, key): """Recursive helper function for inserting a key.""" if root is None: return Node(key) if key < root.key: root.left = self._insert_recursive(root.left, key) else: root.right = self._insert_recursive(root.right, key) return root def inorder_traversal(self): """Performs an in-order traversal of the BST.""" result = [] self._inorder_recursive(self.root, result) return result def _inorder_recursive(self, root, result): """Recursive helper function for in-order traversal.""" if root: self._inorder_recursive(root.left, result) result.append(root.key) self._inorder_recursive(root.right, result) def __str__(self): """Returns a string representation of the in-order traversal.""" return str(self.inorder_traversal())</pre>	
--	--	--

	<pre> # Test Case 1: Insert multiple nodes and check in-order traversal bst1 = BST() bst1.insert(50) bst1.insert(30) bst1.insert(70) bst1.insert(20) bst1.insert(40) bst1.insert(60) bst1.insert(80) assert bst1.inorder_traversal() == [20, 30, 40, 50, 60, 70, 80], "Test Case 1 Failed: Insertion and In-order Traversal" print("Test Case 1 Passed: Insert multiple nodes and check in-order traversal") # Test Case 2: Insert a single node and check in-order traversal bst2 = BST() bst2.insert(100) assert bst2.inorder_traversal() == [100], "Test Case 2 Failed: Insert single node and check in-order traversal" print("Test Case 2 Passed: Insert a single node and check in-order traversal") # Test Case 3: Test with duplicate values (should be inserted in the right subtree in this implementation) bst3 = BST() bst3.insert(5) bst3.insert(3) bst3.insert(5) bst3.insert(7) assert bst3.inorder_traversal() == [3, 5, 5, 7], "Test Case 3 Failed: Insertion with duplicate values" print("Test Case 3 Passed: Test with duplicate values") </pre> <p>Test Case 1 Passed: Insert multiple nodes and check in-order traversal Test Case 2 Passed: Insert a single node and check in-order traversal Test Case 3 Passed: Test with duplicate values</p>	
	<p>Task Description #5 – Hash Table</p> <p>Task: Use AI to implement a hash table with basic insert, search, and delete methods.</p> <p>Sample Input Code:</p> <pre>class HashTable: pass</pre> <p>Expected Output:</p> <ul style="list-style-type: none"> • Collision handling using chaining, with well-commented methods. • Prompt: Generate a Python class HashTable that supports insert, search, and delete operations with collision handling using chaining (lists). <p>Add docstrings for clarity.</p> <p>Provide 3 assert test cases covering insertion, search, and deletion (including key not found case).</p>	

```

class HashTable:
    """Represents a Hash Table with chaining for collision handling."""
    def __init__(self, size):
        """Initializes a hash table with a given size."""
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def _hash_function(self, key):
        """Calculates the hash value for a given key."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Inserts a key-value pair into the hash table.
        If the key already exists, updates the value.
        """
        index = self._hash_function(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value) # Update value if key exists
                return
        self.table[index].append((key, value)) # Append if key doesn't exist

    def search(self, key):
        """Searches for a key in the hash table and returns its value.
        Returns None if the key is not found.
        """
        index = self._hash_function(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None # Key not found

    def delete(self, key):
        """Deletes a key-value pair from the hash table based on the key.
        Returns True if the key was found and deleted, False otherwise.
        """
        index = self._hash_function(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                return True
        return False # Key not found

    def __str__(self):
        """Returns a string representation of the hash table."""
        return str(self.table)

```

	<pre> # Test Case 1: Insertion and Search hash_table1 = HashTable(10) hash_table1.insert("apple", 1) hash_table1.insert("banana", 2) hash_table1.insert("cherry", 3) assert hash_table1.search("apple") == 1, "Test Case 1 Failed: Insertion and Search (apple)" assert hash_table1.search("banana") == 2, "Test Case 1 Failed: Insertion and Search (banana)" assert hash_table1.search("cherry") == 3, "Test Case 1 Failed: Insertion and Search (cherry)" assert hash_table1.search("grape") is None, "Test Case 1 Failed: Search (key not found)" print("Test Case 1 Passed: Insertion and Search") # Test Case 2: Insertion with Collision and Deletion hash_table2 = HashTable(5) # Smaller size to increase chance of collision hash_table2.insert("key1", "value1") hash_table2.insert("key6", "value6") # Likely collision with key1 assuming simple hash assert hash_table2.search("key1") == "value1", "Test Case 2 Failed: Search after collision (key1)" assert hash_table2.search("key6") == "value6", "Test Case 2 Failed: Search after collision (key6)" assert hash_table2.delete("key1") is True, "Test Case 2 Failed: Deletion (key1)" assert hash_table2.search("key1") is None, "Test Case 2 Failed: Search after deletion (key1)" assert hash_table2.search("key6") == "value6", "Test Case 2 Failed: Search for other key after deletion" print("Test Case 2 Passed: Insertion with Collision and Deletion") # Test Case 3: Deletion of Non-existent Key hash_table3 = HashTable(10) hash_table3.insert("orange", 5) assert hash_table3.delete("grape") is False, "Test Case 3 Failed: Deletion of non-existent key" assert hash_table3.search("orange") == 5, "Test Case 3 Failed: Existing key deleted incorrectly" print("Test Case 3 Passed: Deletion of Non-existent Key") </pre> <p>Test Case 1 Passed: Insertion and Search Test Case 2 Passed: Insertion with Collision and Deletion Test Case 3 Passed: Deletion of Non-existent Key</p>
	<p>Task Description #6 – Graph Representation</p> <p>Task: Use AI to implement a graph using an adjacency list.</p> <p>Sample Input Code:</p> <pre>class Graph: pass</pre> <p>Expected Output:</p> <ul style="list-style-type: none"> • Graph with methods to add vertices, add edges, and display connections. • Prompt: Generate a Python class Graph that represents a graph using an adjacency list. Implement methods to add vertices, add edges (undirected), and display connections. Include docstrings and write 3 assert test cases to verify graph connectivity.

	<pre> class Graph: """Represents a graph using an adjacency list.""" def __init__(self): """Initializes an empty graph with an adjacency list.""" self.graph = [] def add_vertex(self, vertex): """Adds a vertex to the graph if it doesn't already exist.""" if vertex not in self.graph: self.graph[vertex] = [] def add_edge(self, vertex1, vertex2): """Adds an undirected edge between two vertices. Adds vertices if they don't exist. """ self.add_vertex(vertex1) self.add_vertex(vertex2) # Add edge in both directions for an undirected graph if vertex2 not in self.graph[vertex1]: self.graph[vertex1].append(vertex2) if vertex1 not in self.graph[vertex2]: self.graph[vertex2].append(vertex1) def display_connections(self): """Displays the adjacency list representation of the graph.""" for vertex, neighbors in self.graph.items(): print(f'{vertex}: {neighbors}') def get_vertices(self): """Returns a list of all vertices in the graph.""" return list(self.graph.keys()) def get_neighbors(self, vertex): """Returns a list of neighbors for a given vertex. Returns an empty list if the vertex does not exist. """ return self.graph.get(vertex, []) def __str__(self): """Returns a string representation of the graph.""" return str(self.graph) # Test Case 1: Add vertices and edges, check connectivity graph1 = Graph() graph1.add_vertex("A") graph1.add_vertex("B") graph1.add_vertex("C") graph1.add_edge("A", "B") graph1.add_edge("B", "C") assert "A" in graph1.get_vertices(), "Test Case 1 Failed: Add vertex A" assert "B" in graph1.get_vertices(), "Test Case 1 Failed: Add vertex B" assert "C" in graph1.get_vertices(), "Test Case 1 Failed: Add vertex C" assert "D" not in graph1.get_vertices(), "Test Case 1 Failed: Non-existent vertex D" assert "B" in graph1.get_neighbors("A"), "Test Case 1 Failed: Edge A-B" assert "A" in graph1.get_neighbors("B"), "Test Case 1 Failed: Edge B-A" assert "C" in graph1.get_neighbors("B"), "Test Case 1 Failed: Edge B-C" assert "B" in graph1.get_neighbors("C"), "Test Case 1 Failed: Edge C-B" assert "A" not in graph1.get_neighbors("C"), "Test Case 1 Failed: No edge A-C" assert graph1.get_neighbors("D") == [], "Test Case 1 Failed: Neighbors of non-existent vertex" print("Test Case 1 Passed: Add vertices and edges, check connectivity") # Test Case 2: Add edge to non-existent vertices graph2 = Graph() graph2.add_edge("X", "Y") assert "X" in graph2.get_vertices(), "Test Case 2 Failed: Add vertex X via edge" assert "Y" in graph2.get_vertices(), "Test Case 2 Failed: Add vertex Y via edge" assert "Y" in graph2.get_neighbors("X"), "Test Case 2 Failed: Edge X-Y added" print("Test Case 2 Passed: Add edge to non-existent vertices") # Test Case 3: Adding duplicate edge graph3 = Graph() graph3.add_edge("P", "Q") graph3.add_edge("P", "Q") # Add duplicate edge assert len(graph3.get_neighbors("P")) == 1, "Test Case 3 Failed: Duplicate edge added to P" assert len(graph3.get_neighbors("Q")) == 1, "Test Case 3 Failed: Duplicate edge added to Q" print("Test Case 3 Passed: Adding duplicate edge") </pre> <p>Test Case 1 Passed: Add vertices and edges, check connectivity Test Case 2 Passed: Add edge to non-existent vertices Test Case 3 Passed: Adding duplicate edge</p>	
	<p>Task Description #7 – Priority Queue</p> <p>Task: Use AI to implement a priority queue using Python's heapq module.</p> <p>Sample Input Code:</p> <pre> class PriorityQueue: pass </pre>	

Expected Output:

- Implementation with enqueue (priority), dequeue (highest priority), and display methods.
- **Prompt:** Generate a Python class PriorityQueue using the heapq module.
Implement methods to enqueue elements with priority, dequeue the element with highest priority, and display the queue.
Add docstrings and write 3 assert test cases to verify enqueue-dequeue behavior.

```
import heapq
class PriorityQueue:
    """Represents a Priority Queue using the heapq module."""
    def __init__(self):
        """Initializes an empty priority queue."""
        self._queue = []

    def enqueue(self, priority, item):
        """Adds an item with a given priority to the priority queue.
        Lower priority values are considered higher priority.
        """
        heapq.heappush(self._queue, (priority, item))

    def dequeue(self):
        """Removes and returns the item with the highest priority.
        Raises an IndexError if the priority queue is empty.
        """
        if not self.is_empty():
            return heapq.heappop(self._queue)[1] # Return the item, not the priority
        else:
            raise IndexError("dequeue from empty priority queue")

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

    def __len__(self):
        """Returns the number of items in the priority queue."""
        return len(self._queue)

    def __str__(self):
        """Returns a string representation of the priority queue.
        # Note: The internal representation of the heap is not sorted by priority
        # This string representation shows the internal heap structure
        """
        return str(self._queue)

# Test Case 1: Enqueue and Dequeue in order of priority
pq1 = PriorityQueue()
pq1.enqueue(2, "Task B")
pq1.enqueue(1, "Task A")
pq1.enqueue(3, "Task C")
assert pq1.dequeue() == "Task A", "Test Case 1 Failed: Dequeue order (Task A)"
assert pq1.dequeue() == "Task B", "Test Case 1 Failed: Dequeue order (Task B)"
assert pq1.dequeue() == "Task C", "Test Case 1 Failed: Dequeue order (Task C)"
assert pq1.is_empty(), "Test Case 1 Failed: Priority queue not empty after dequeues"
print("Test Case 1 Passed: Enqueue and Dequeue in order of priority")

# Test Case 2: Enqueue with same priority
pq2 = PriorityQueue()
pq2.enqueue(5, "Task X")
pq2.enqueue(5, "Task Y")
# The order for same priority items is not guaranteed, but both should be dequeued
item1 = pq2.dequeue()
item2 = pq2.dequeue()
assert (item1 == "Task X" and item2 == "Task Y") or (item1 == "Task Y" and item2 == "Task X"), "Test Case 2 Failed: Enqueue with same priority"
assert pq2.is_empty(), "Test Case 2 Failed: Priority queue not empty after dequeues"
print("Test Case 2 Passed: Enqueue with same priority")

# Test Case 3: Dequeue from Empty Priority Queue (expecting an error)
pq3 = PriorityQueue()
try:
    pq3.dequeue()
    assert False, "Test Case 3 Failed: Dequeue from Empty Priority Queue (No exception raised)"
except IndexError:
    assert True, "Test Case 3 Failed: Dequeue from Empty Priority Queue (Wrong exception raised)"
print("Test Case 3 Passed: Dequeue from Empty Priority Queue")

Test Case 1 Passed: Enqueue and Dequeue in order of priority
Test Case 2 Passed: Enqueue with same priority
Test Case 3 Passed: Dequeue from Empty Priority Queue
```

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 Python class DequeDS using collections.deque. Implement methods to insert and remove elements from both ends, and display the deque. Include docstrings and write 3 assert test cases to confirm correct behavior.

```
from collections import deque
class DequeDS:
    """Represents a Double-Ended Queue (Deque) using collections.deque."""
    def __init__(self):
        """Initializes an empty deque."""
        self._deque = deque()
    def insert_front(self, item):
        """Adds an item to the front of the deque."""
        self._deque.appendleft(item)
    def insert_rear(self, item):
        """Adds an item to the rear of the deque."""
        self._deque.append(item)
    def remove_front(self):
        """Removes and returns the item from the front of the deque.
        Raises an IndexError if the deque is empty.
        """
        if not self.is_empty():
            return self._deque.popleft()
        else:
            raise IndexError("remove from empty deque")
    def remove_rear(self):
        """Removes and returns the item from the rear of the deque.
        Raises an IndexError if the deque is empty.
        """
        if not self.is_empty():
            return self._deque.pop()
        else:
            raise IndexError("remove from empty deque")
    def is_empty(self):
        """Returns True if the deque is empty, False otherwise."""
        return len(self._deque) == 0
    def size(self):
        """Returns the number of items in the deque."""
        return len(self._deque)
```

```
def display(self):
    """Displays the elements in the deque."""
    return list(self._deque)
def __len__(self):
    """Returns the number of items in the deque."""
    return len(self._deque)
def __str__(self):
    """Returns a string representation of the deque."""
    return str(list(self._deque))
```

```

deque1 = DequeDS()
deque1.insert_front(1)
deque1.insert_rear(2)
deque1.insert_front(0)
deque1.insert_rear(3)
assert deque1.remove_front() == 0, "Test Case 1 Failed: Remove from front (1)"
assert deque1.remove_rear() == 3, "Test Case 1 Failed: Remove from rear (1)"
assert deque1.remove_front() == 1, "Test Case 1 Failed: Remove from front (2)"
assert deque1.remove_rear() == 2, "Test Case 1 Failed: Remove from rear (2)"
assert deque1.is_empty(), "Test Case 1 Failed: Deque not empty after removals"
print("Test Case 1 Passed: Insert at front and rear, remove from front and rear")

# Test Case 2: Check size and display
deque2 = DequeDS()
deque2.insert_front('a')
deque2.insert_rear('b')
deque2.insert_front('c')
assert deque2.size() == 3, "Test Case 2 Failed: size"
assert deque2.display() == ['c', 'a', 'b'], "Test Case 2 Failed: Display"
print("Test Case 2 Passed: Check size and display")

# Test Case 3: Remove from Empty Deque (expecting an error)
dequeue3 = DequeDS()
try:
    dequeue3.remove_rear()
    assert False, "Test Case 3 Failed: Remove from Empty Deque (No exception raised)"
except IndexError:
    assert True, "Test Case 3 Failed: Remove from Empty Deque (Correct exception for front)"

try:
    dequeue3.remove_front()
    assert False, "Test Case 3 Failed: Remove from Empty Deque (No exception raised)"
except IndexError:
    assert True, "Test Case 3 Failed: Remove from Empty Deque (Correct exception for rear)"
print("Test Case 3 Passed: Remove from Empty Deque")

```

Test Case 1 Passed: Insert at front and rear, remove from front and rear
 Test Case 2 Passed: Check size and display
 Test Case 3 Passed: Remove from Empty Deque

Task 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 markdown table comparing Stack, Queue, Linked List, BST, Hash Table, Graph, Priority Queue, and Deque.
 Include columns for operations (insert, delete, search, traversal) and their time complexities (average/worst case).
 Make it clean and easy to read.

		Data Structure	Operation	Time Complexity (Average)	Time Complexity (Worst Case)	Notes
Stack	Push	O(1)	O(1)	Using Python list (amortized)		
	Pop	O(1)	O(1)	Using Python list (amortized)		
	Peek	O(1)	O(1)			
	is_empty	O(1)	O(1)			
Queue	Enqueue	O(1)	O(1)	Using Python list (amortized append)		
	Dequeue	O(n)	O(n)	Using Python list (pop from start)		
	Peek	O(1)	O(1)			
	Size	O(1)	O(1)			
Linked List	Insert at End	O(n)	O(n)	Depends on traversal to find end		
	Delete	O(n)	O(n)	Depends on traversal to find node		
	Search	O(n)	O(n)	Linear search		
	Traversal	O(n)	O(n)	Visiting each node		
BST	Insert	O(log n)	O(n)	Worst case is skewed tree		
	Search	O(log n)	O(n)	Worst case is skewed tree		
	Deletion	O(log n)	O(n)	Worst case is skewed tree		
	Traversal	O(n)	O(n)	Visiting each node		
Hash Table	Insert	O(1)	O(n)	Worst case with poor hash function/collisions		
	Search	O(1)	O(n)	Worst case with poor hash function/collisions		
	Delete	O(1)	O(n)	Worst case with poor hash function/collisions		
	Add Vertex	O(1)	O(1)	Using dictionary		
Graph	Add Edge	O(1)	O(1)	Using dictionary and list append		
	Get Neighbors	O(degree)	O(n)	Degree of the vertex		
	Traversal (BFS/DFS)	O(V + E)	O(V + E)	V = vertices, E = edges		
	Enqueue	O(log n)	O(log n)	Using heapq (heappify is O(n))		
Priority Queue	Dequeue	O(log n)	O(log n)	Using heapq		
	Peek	O(1)	O(1)	Using heapq		
	Insert Front	O(1)	O(1)	Using collections.deque		
	Insert Rear	O(1)	O(1)	Using collections.deque		
Deque	Remove Front	O(1)	O(1)	Using collections.deque		
	Remove Rear	O(1)	O(1)	Using collections.deque		

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:
 - Stack
 - Queue
 - Priority Queue
 - Linked List
 - Binary Search Tree (BST)
 - Graph
 - Hash Table
 - 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.
- **Prompt:** Create a table mapping each of these features – Student Attendance Tracking, Event Registration System, Library Book Borrowing, Bus Scheduling System, and Cafeteria Order Queue – to the most appropriate data structure (choose from stack, queue, priority queue, linked list, BST, graph, hash table, deque).

Give 2–3 sentence justification for each choice.

Then generate Python code to implement **one feature** (choose the most interesting one, e.g., Cafeteria Order Queue using Queue).

Include docstrings, and provide 3 assert test cases

```
print("""
| Feature | Most Appropriate Data Structure |
|-----|-----|
| Student Attendance Tracking | Hash Table |
| Event Registration System | BST or Hash Table |
| Library Book Borrowing | Hash Table or Linked List |
| Bus Scheduling System | Queue or Priority Queue |
| Cafeteria Order Queue | Queue |
""")
```

Feature	Most Appropriate Data Structure
Student Attendance Tracking	Hash Table
Event Registration System	BST or Hash Table
Library Book Borrowing	Hash Table or Linked List
Bus Scheduling System	Queue or Priority Queue
Cafeteria Order Queue	Queue

	<pre> class CafeteriaOrderQueue: """ Represents a cafeteria order system using a Queue data structure. Orders are processed in a First-In, First-Out (FIFO) manner. """ def __init__(self): """Initializes an empty cafeteria order queue.""" self._orders = Queue() def add_order(self, order): """ Adds a new order to the rear of the queue. Args: order: The order details to be added. """ self._orders.enqueue(order) print(f"Order '{order}' added to the queue.") def process_order(self): """ Removes and returns the order from the front of the queue (processes the next order). Returns: The processed order. Raises: IndexError: If the order queue is empty. """ try: order = self._orders.dequeue() print(f"Processing order: '{order}'") return order except IndexError: raise IndexError("No orders to process. The queue is empty.") def view_next_order(self): """ Returns the order at the front of the queue without removing it (previews the next order). Returns: The next order to be processed. Raises: IndexError: If the order queue is empty. """ try: return self._orders.peek() except IndexError: raise IndexError("No orders in the queue.") def is_empty(self): """ Checks if the order queue is empty. Returns: True if the queue is empty, False otherwise. """ return self._orders.is_empty() def __len__(self): """Returns the number of orders in the queue.""" return len(self._orders) def __str__(self): """Returns a string representation of the order queue.""" return str(self._orders) </pre>	
--	--	--

```

# Test Case 1: Add orders and process them in FIFO order
cafe_queue1 = CafeteriaOrderQueue()
cafe_queue1.add_order("Burger and Fries")
cafe_queue1.add_order("Pizza Slice")
cafe_queue1.add_order("Salad")

assert not cafe_queue1.is_empty(), "Test Case 1 Failed: Queue should not be empty after adding orders"
assert len(cafe_queue1) == 3, "Test Case 1 Failed: Queue size should be 3"
assert cafe_queue1.view_next_order() == "Burger and Fries", "Test Case 1 Failed: View next order (1)"
assert cafe_queue1.process_order() == "Burger and Fries", "Test Case 1 Failed: Process order (1)"
assert len(cafe_queue1) == 2, "Test Case 1 Failed: Queue size should be 2 after processing"
assert cafe_queue1.process_order() == "Pizza Slice", "Test Case 1 Failed: Process order (2)"
assert cafe_queue1.process_order() == "Salad", "Test Case 1 Failed: Process order (3)"
assert cafe_queue1.is_empty(), "Test Case 1 Failed: Queue should be empty after processing all orders"
print("Test Case 1 Passed: Add orders and process them in FIFO order")

# Test Case 2: View next order on an empty queue (expecting an error)
cafe_queue2 = CafeteriaOrderQueue()
try:
    cafe_queue2.view_next_order()
    assert False, "Test Case 2 Failed: View next order on empty queue (No exception raised)"
except IndexError:
    assert True, "Test Case 2 Failed: View next order on empty queue (Correct exception)"
print("Test Case 2 Passed: View next order on an empty queue")

# Test Case 3: Process order from an empty queue (expecting an error)
cafe_queue3 = CafeteriaOrderQueue()
try:
    cafe_queue3.process_order()
    assert False, "Test Case 3 Failed: Process order from empty queue (No exception raised)"
except IndexError:
    assert True, "Test Case 3 Failed: Process order from empty queue (Correct exception)"
print("Test Case 3 Passed: Process order from an empty queue")

Order 'Burger and Fries' added to the queue.
Order 'Pizza Slice' added to the queue.
Order 'Salad' added to the queue.
Processing order: 'Burger and Fries'.
Processing order: 'Pizza Slice'.
Processing order: 'Salad'.
Test Case 1 Passed: Add orders and process them in FIFO order
Test Case 2 Passed: View next order on an empty queue
Test Case 3 Passed: Process order from an empty queue

```

```

# Test Case 1: Add orders and process them in FIFO order
cafe_queue1 = CafeteriaOrderQueue()
cafe_queue1.add_order("Burger and Fries")
cafe_queue1.add_order("Pizza Slice")
cafe_queue1.add_order("Salad")

assert not cafe_queue1.is_empty(), "Test Case 1 Failed: Queue should not be empty after adding orders"
assert len(cafe_queue1) == 3, "Test Case 1 Failed: Queue size should be 3"
assert cafe_queue1.view_next_order() == "Burger and Fries", "Test Case 1 Failed: View next order (1)"
assert cafe_queue1.process_order() == "Burger and Fries", "Test Case 1 Failed: Process order (1)"
assert len(cafe_queue1) == 2, "Test Case 1 Failed: Queue size should be 2 after processing"
assert cafe_queue1.process_order() == "Pizza Slice", "Test Case 1 Failed: Process order (2)"
assert cafe_queue1.process_order() == "Salad", "Test Case 1 Failed: Process order (3)"
assert cafe_queue1.is_empty(), "Test Case 1 Failed: Queue should be empty after processing all orders"
print("Test Case 1 Passed: Add orders and process them in FIFO order")

# Test Case 2: View next order on an empty queue (expecting an error)
cafe_queue2 = CafeteriaOrderQueue()
try:
    cafe_queue2.view_next_order()
    assert False, "Test Case 2 Failed: View next order on empty queue (No exception raised)"
except IndexError:
    assert True, "Test Case 2 Failed: View next order on empty queue (Correct exception)"
print("Test Case 2 Passed: View next order on an empty queue")

# Test Case 3: Process order from an empty queue (expecting an error)
cafe_queue3 = CafeteriaOrderQueue()
try:
    cafe_queue3.process_order()
    assert False, "Test Case 3 Failed: Process order from empty queue (No exception raised)"
except IndexError:
    assert True, "Test Case 3 Failed: Process order from empty queue (Correct exception)"
print("Test Case 3 Passed: Process order from an empty queue")

Order 'Burger and Fries' added to the queue.
Order 'Pizza Slice' added to the queue.
Order 'Salad' added to the queue.
Processing order: 'Burger and Fries'.
Processing order: 'Pizza Slice'.
Processing order: 'Salad'.
Test Case 1 Passed: Add orders and process them in FIFO order
Test Case 2 Passed: View next order on an empty queue
Test Case 3 Passed: Process order from an empty queue

```

	<p><input checked="" type="checkbox"/> Deliverables (For All Tasks)</p> <ol style="list-style-type: none">1. AI-generated prompts for code and test case generation.2. At least 3 assert test cases for each task.3. AI-generated initial code and execution screenshots.4. Analysis of whether code passes all tests.5. Improved final version with inline comments and explanation.6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.	
--	---	--