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	*Expected Time to complete*
1	**Lab 11 – Data Structures with AI: Implementing Fundamental Structures** **Lab Objectives** • 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.	Week6 - Monday

- Enhance code quality with AI-generated comments and performance suggestions.

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.

prompts :.

Create a Python class named Stack with methods: push, pop, peek, and is_empty. Include docstrings and make it interactive to take user inputs for push operations

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

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.items:
            print("Stack is empty")
            return None
        return self.items.pop()

# User interaction
s = Stack()
n = int(input("How many elements to push? "))
for _ in range(n):
    s.push(int(input("Enter element: ")))

print("Top element:", s.items[-1] if s.items else "Stack empty")
print("Popped:", s.pop())
```

```
How many elements to push? 3
Enter element: 10
Enter element: 20
Enter element: 30
Top element: 30
Popped: 30
```

observations and code expalnation

- Defined Stack class with a list to store elements.
- push(item) → adds an element to the top of stack.
- pop() → removes and returns the top element; handles empty stack.
- peek() → shows top element without removing it.
- is_empty() → checks if stack is empty.
- Interactive code asks for number of elements → pushes elements from user input.

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.

prompts :

Create a Python class named Queue using list. Include methods: enqueue, dequeue, peek, and size. Make it interactive for user input.

code:

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

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.items:
            print("Queue is empty")
            return None
        return self.items.pop(0)

    def peek(self):
        return self.items[0] if self.items else None

    def size(self):
        return len(self.items)

# User interaction
q = Queue()
n = int(input("How many elements to enqueue? "))
for _ in range(n):
    q.enqueue(int(input("Enter element: ")))

print("Front element:", q.peek())
print("Dequeued:", q.dequeue())
```

```
How many elements to enqueue? 3
Enter element: 5
Enter element: 10
Enter element: 15
Front element: 5
Dequeued: 5
```

observations and code expalnation :

- Queue class stores elements in a list.
- enqueue(item) → adds element at the rear.
- dequeue() → removes element from front; prints message if empty.
- peek() → shows front element.
- size() → returns number of elements.
- User input used to enqueue multiple elements; then displays front and dequeued element.

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.

Prompts :

Create a Python Singly Linked List with Node class. Include insert and display methods. Make it interactive for user input.

Code:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def insert(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = new_node
    def display(self):
        elements = []
        temp = self.head
        while temp:
            elements.append(temp.data)
            temp = temp.next
        return elements
# User interaction
ll = LinkedList()
n = int(input("How many elements to insert? "))
for _ in range(n):
    ll.insert(int(input("Enter element: ")))
print("Linked List:", ll.display())
```

```
How many elements to insert? 3
Enter element: 10
Enter element: 20
Enter element: 30
Linked List: [10, 20, 30]
```

Code explanation and observations :

- Node class stores data and next pointer.
- LinkedList class has head pointer.
- insert(data) → adds new node at end.
- display() → prints all nodes in order.
- User inputs number of nodes → program inserts each one and displays list.

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.

Prompts :

Create a Python BST class with insert and inorder traversal methods. Make it interactive for user input.

Code :

```
class BST:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
    def insert(self, value):
        if value < self.value:
            if self.left is None:
                self.left = BST(value)
            else:
                self.left.insert(value)
        elif value > self.value:
            if self.right is None:
                self.right = BST(value)
            else:
                self.right.insert(value)
    def inorder(self):
        elements = []
        if self.left:
            elements.extend(self.left.inorder())
        elements.append(self.value)
        if self.right:
```

(method) inorder: () -> list

```python
        def inorder(self):
            elements = []
            if self.left:
                elements.extend(self.left.inorder())
            elements.append(self.value)
            if self.right:
                elements.extend(self.right.inorder())
            return elements
    # User interaction
    root_val = int(input("Enter root value: "))
    root = BST(root_val)
    n = int(input("How many elements to insert? "))
    for _ in range(n):
        root.insert(int(input("Enter element: ")))

    print("Inorder Traversal:", root.inorder())
```

(method) inorder: () -> list

```
Enter root value: 50
How many elements to insert? 6
Enter element: 30
Enter element: 70
Enter element: 20
Enter element: 40
Enter element: 60
Enter element: 80
Inorder Traversal: [20, 30, 40, 50, 60, 70, 80]
```

observations and code expalnation

- BST node has data, left, right.
- insert(data) → recursively places node in correct position.
- inorder() → returns elements in sorted order.
- Program asks for root, then number of elements → inserts each.
- Prints inorder traversal to verify BST structure.

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.

Prompts :

Create a Python Hash Table class with insert, search, and delete methods. Handle collisions using chaining. Include docstrings and user input.

Code:

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

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

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

    def search(self, key):
        idx = self._hash(key)
        for pair in self.table[idx]:
            if pair[0] == key:
                return pair[1]
        return None

    def delete(self, key):
        idx = self._hash(key)
        for pair in self.table[idx]:
            if pair[0] == key:
                self.table[idx].remove(pair)
                return True
        return False
```

```
# User interaction
ht = HashTable()
n = int(input("How many key-value pairs to insert? "))
for _ in range(n):
    k = input("Enter key: ")
    v = input("Enter value: ")
    ht.insert(k, v)

search_key = input("Enter key to search: ")
print("Search Result:", ht.search(search_key))
```

```
How many key-value pairs to insert? 3
Enter key: 1
Enter value: A
Enter key: 2
Enter value: B
Enter key: 11
Enter value: C
Enter key to search: 11
Search Result: C
```

Observation and code explanation

- Uses a list of lists (buckets) for collision chaining.
- insert(key, value) → hashes key → appends to bucket.
- search(key) → looks for key in bucket → returns value or None.
- delete(key) → removes key-value pair if exists.
- User enters number of key-value pairs → program inserts → searches and prints result.

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 :

Create a Python Graph class using adjacency list. Include methods: add_vertex, add_edge, display. Make it interactive.

Code

```
class Graph:
    def __init__(self):
        self.adj_list = {}
    def add_vertex(self, vertex):
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []
    def add_edge(self, v1, v2):
        if v1 in self.adj_list and v2 in self.adj_list:
            self.adj_list[v1].append(v2)
            self.adj_list[v2].append(v1)
    def display(self):
        return self.adj_list
```

```python
    # User interaction
    g = Graph()
    n = int(input("How many vertices? "))
    for _ in range(n):
        v = input("Enter vertex: ")
        g.add_vertex(v)
    m = int(input("How many edges? "))
    for _ in range(m):
        v1 = input("Enter vertex 1: ")
        v2 = input("Enter vertex 2: ")
        g.add_edge(v1, v2)
    print("Graph:", g.display())
```

```
How many vertices? 3
Enter vertex: A
Enter vertex: B
Enter vertex: C
How many edges? 2
Enter vertex 1: A
Enter vertex 2: B
Enter vertex 1: A
Enter vertex 2: C
Graph: {'A': ['B', 'C'], 'B': ['A'], 'C': ['A']}
```

Code explanation and Observation :

- Dictionary stores adjacency list.
- add_vertex(v) → adds vertex if not exist.
- add_edge(v1, v2) → adds edge (undirected) between two vertices.
- display() → prints adjacency list.
- Program asks number of vertices and edges → user enters each → displays graph.

Task Description #7 – Priority Queue

Task: Use AI to implement a priority queue using Python's heapq module.

Sample Input Code:

```python
class PriorityQueue:
    pass
```

Expected Output:

- Implementation with enqueue (priority), dequeue (highest priority), and display methods.

Prompt :

Create a Python Priority Queue class using heapq. Include enqueue (priority), dequeue (highest priority), and display methods. Interactive user input.

Code :

```
import heapq
class PriorityQueue:
    def __init__(self):
        self.queue = []
    def enqueue(self, priority, item):
        heapq.heappush(self.queue, (priority, item))
    def dequeue(self):
        if not self.queue:
            print("Queue is empty")
            return None
        return heapq.heappop(self.queue)[1]
    def display(self):
        return [item for priority, item in self.queue]
# User interaction
pq = PriorityQueue()
n = int(input("How many elements to enqueue? "))
for _ in range(n):
    item = input("Enter item: ")
    priority = int(input("Enter priority: "))
    pq.enqueue(priority, item)
print("Priority Queue:", pq.display())
print("Dequeued:", pq.dequeue())
```

```
How many elements to enqueue? 3
Enter item: task1
Enter priority: 2
Enter item: task2
Enter priority: 1
Enter item: task3
Enter priority: 3
Priority Queue: ['task2', 'task1', 'task3']
Dequeued: task2
```

Observation and code explanation:

- Uses heapq for priority management.
- enqueue(priority, item) → pushes tuple (priority, item).
- dequeue() → pops element with smallest priority value.
- display() → shows queue.
- User enters items with priority → program enqueues → dequeues highest priority → displays 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 :

Create a Python Deque class using collections.deque. Include methods: insert_front, insert_rear, remove_front, remove_rear, display. Interactive input.

Code :

```
class DequeDS:
    def __init__(self):
        self.deque = deque()

    def insert_front(self, item):
        self.deque.appendleft(item)

    def insert_rear(self, item):
        self.deque.append(item)

    def remove_front(self):
        if not self.deque:
            print("Deque is empty")
            return None
        return self.deque.popleft()

    def remove_rear(self):
        if not self.deque:
            print("Deque is empty")
            return None
        return self.deque.pop()

    def display(self):
        return list(self.deque)

# User interaction
d = DequeDS()
n = int(input("How many elements to insert at rear? "))
for _ in range(n):
    d.insert_rear(input("Enter element: "))

print("Deque:", d.display())
print("Removed from front:", d.remove_front())
```

```
# User interaction
d = DequeDS()
n = int(input("How many elements to insert at rear? "))
for _ in range(n):
    d.insert_rear(input("Enter element: "))

print("Deque:", d.display())
print("Removed from front:", d.remove_front())
```

```
How many elements to insert at rear? 3
Enter element: 10
Enter element: 20
Enter element: 30
Deque: ['10', '20', '30']
Removed from front: 10
```

Observation and code explation

- collections.deque allows fast insertion/removal at both ends.
- insert_front(item) → adds to front.
- insert_rear(item) → adds to rear.
- remove_front() / remove_rear() → removes from respective end.
- display() → prints deque.
- User inputs number of elements → inserts → removes → displays 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, Deque with their time complexities for insertion, deletion, search, access. Using pandas

Code :

```
import pandas as pd

# Create a dictionary with the data
data = {
    "Data Structure": ["Stack", "Queue", "Linked List", "BST", "Hash Table", "Graph", "Priority Queue", "Deque"],
    "Insertion": ["O(1)", "O(1)", "O(1)", "O(log n)", "O(1)", "O(1)", "O(log n)", "O(1)"],
    "Deletion": ["O(1)", "O(1)", "O(1)", "O(log n)", "O(1)", "O(1)", "O(log n)", "O(1)"],
    "Search": ["O(n)", "O(n)", "O(n)", "O(log n)", "O(1)", "O(V+E)", "O(n)", "O(n)"],
    "Access": ["O(1)", "O(1)", "O(n)", "O(n)", "O(1)", "O(V+E)", "O(n)", "O(1)"]
}

# Create the DataFrame
df = pd.DataFrame(data)

# Display the DataFrame
df
```

	Data Structure	Insertion	Deletion	Search	Access
0	Stack	O(1)	O(1)	O(n)	O(1)
1	Queue	O(1)	O(1)	O(n)	O(1)
2	Linked List	O(1)	O(1)	O(n)	O(n)
3	BST	O(log n)	O(log n)	O(log n)	O(n)
4	Hash Table	O(1)	O(1)	O(1)	O(1)
5	Graph	O(1)	O(1)	O(V+E)	O(V+E)
6	Priority Queue	O(log n)	O(log n)	O(n)	O(n)
7	Deque	O(1)	O(1)	O(n)	O(1)

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.

 Deliverables (For All Tasks)

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.

Prompt :

Create a Python menu-driven Queue for cafeteria orders. Include methods: place_order, serve_order, next_order, queue_size. Use user input to interact.

```python
class CafeteriaOrderQueue:
    """
    A user-defined Queue implementation for cafeteria orders.
    Implements FIFO (First-In-First-Out) behavior.
    """
    def __init__(self, capacity=100):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = 0
        self.rear = -1
        self.size = 0

    def place_order(self, student_name: str):
        """
        Add a student's order to the queue.
        """
        if self.size == self.capacity:
            print("❌ Queue is full! Cannot place more orders.")
            return
        self.rear = (self.rear + 1) % self.capacity
        self.queue[self.rear] = student_name
        self.size += 1
        print(f"✅ Order placed for {student_name}")

    def serve_order(self):
        """
        Serve the next student in the queue.
        """
        if self.size == 0:
            print("❌ No orders to serve.")
            return
        order = self.queue[self.front]
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        print(f"🍽️ Served: {order}")

    def next_order(self):
        """
        Peek at the next order without removing it.
        """
        if self.size == 0:
            print("❌ No orders in queue.")
        else:
            print(f"👉 Next order: {self.queue[self.front]}")

    def queue_size(self):
        """
        Display number of orders in the queue.
        """
        print(f"📊 Orders in queue: {self.size}")


# ---------------- Main Program with User Input ----------------
if __name__ == "__main__":
    cafeteria = CafeteriaOrderQueue(capacity=10)

    while True:
        print("\n=== Cafeteria Order Queue System ===")
        print("1. Place Order")
        print("2. Serve Order")
        print("3. Show Next Order")
        print("4. Show Queue Size")
        print("5. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            name = input("Enter student name: ")
            cafeteria.place_order(name)
        elif choice == "2":
            cafeteria.serve_order()
        elif choice == "3":
            cafeteria.next_order()
        elif choice == "4":
            cafeteria.queue_size()
        elif choice == "5":
            print(" Exiting Cafeteria System. Goodbye!")
            break
        else:
```

```
Enter your choice: 1
Enter student name: bob
✅ Order placed for bob

=== Cafeteria Order Queue System ===
1. Place Order
2. Serve Order
3. Show Next Order
4. Show Queue Size
5. Exit
Enter your choice: 2
🍽️ Served: alice

=== Cafeteria Order Queue System ===
1. Place Order
2. Serve Order
3. Show Next Order
4. Show Queue Size
5. Exit
Enter your choice: 1
Enter student name: dsa
✅ Order placed for dsa

=== Cafeteria Order Queue System ===
1. Place Order
2. Serve Order
3. Show Next Order
4. Show Queue Size
5. Exit
Enter your choice: 3
👉 Next order: bob

=== Cafeteria Order Queue System ===
1. Place Order
2. Serve Order
3. Show Next Order
4. Show Queue Size
5. Exit
Enter your choice: 2
🍽️ Served: bob

=== Cafeteria Order Queue System ===
1. Place Order
2. Serve Order
3. Show Next Order
4. Show Queue Size
5. Exit
Enter your choice: 4
📊 Orders in queue: 1

=== Cafeteria Order Queue System ===
1. Place Order
2. Serve Order
3. Show Next Order
4. Show Queue Size
5. Exit
Enter your choice: 5
 Exiting Cafeteria System. Goodbye!
```

Observations and Code Explanation

- CafeteriaOrderQueue uses circular queue array.
- place_order(student_name) → adds student; handles full queue.
- serve_order() → serves front student; handles empty queue.
- next_order() → shows next order without removing.
- queue_size() → displays number of orders.
- Menu allows user to place, serve, peek, check size, exit interactively.