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 - Thursday	**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 - Thursday

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

Task 1: Implementing a Stack (LIFO)

- **Task**: Use AI to help implement a **Stack** class in Python with the following operations: push(), pop(), peek(), and is_empty().
- **Instructions**:
  - Ask AI to generate code skeleton with docstrings.
  - Test stack operations using sample data.
  - Request AI to suggest optimizations or alternative implementations (e.g., using collections.deque).
- **Expected Output**:
  - A working Stack class with proper methods, Google-style docstrings, and inline comments for tricky parts.

PROMPT : write a python program implement a **Stack** class with proper methods, Google-style docstrings, and inline comments for tricky parts.

CODE:

```
class Stack:
    """A simple Stack implementation in Python.

    Provides standard stack operations such as push, pop, peek, is_empty, and siz
    """

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

    def push(self, item):
        """Push an item onto the stack.

        Args:
            item: The item to be added to the stack.
        """
        self._items.append(item)
        print(f"Pushed {item!r}: {self._items}")

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

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            print("Error: pop from empty stack")
            raise IndexError('pop from empty stack')
        popped_item = self._items.pop()
        print(f"Popped {popped_item!r}: {self._items}")
```

```
    return popped_item

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

    Returns:
        The item at the top of the stack.

    Raises:
        IndexError: If the stack is empty.
    """
    if self.is_empty():
        print("Error: peek from empty stack")
        raise IndexError('peek from empty stack')
    print(f"Peeked at {self._items[-1]!r}: {self._items}")
    return self._items[-1]

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

    Returns:
        True if the stack is empty, False otherwise.
    """
    empty = len(self._items) == 0
    print(f"Is empty? {empty}")
    return empty

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

    Returns:
        The size of the stack as an integer.
    """
        sz = len(self._items)
        print(f"Size: {sz}")
        return sz
if __name__ == "__main__":
    s = Stack()
    s.is_empty()    # True
    s.push(10)
    s.push(20)
    s.peek()        # 20
    s.push(30)
    s.size()        # 3
    s.pop()
    s.pop()
    s.pop()
    s.is_empty()    # True
    # Next line will raise IndexError and print an error
    try:
        s.pop()
    except IndexError:
        pass
```

OUTPUT:

```
Is empty? True
Pushed 10: [10]
Pushed 20: [10, 20]
Is empty? False
Peeked at 20: [10, 20]
Pushed 30: [10, 20, 30]
Size: 3
Is empty? False
Popped 30: [10, 20]
Is empty? False
Popped 20: [10]
Is empty? False
Popped 10: []
Is empty? True
Is empty? True
Error: pop from empty stack
PS C:\Users\DEEKSHA\OneDrive\Desktop\AIAC\Lab-11>
```

Task 2: Queue Implementation with Performance Review

- **Task**: Implement a **Queue** with enqueue(), dequeue(), and is_empty() methods.
- **Instructions**:
  - First, implement using Python lists.
  - Then, ask AI to review performance and suggest a more efficient implementation (using collections.deque).
- **Expected Output**:
  - Two versions of a queue: one with lists and one optimized with deque, plus an AI-generated performance comparison.

PROMPT: write the python code Implement a **Queue** with Two versions of a queue: one with lists and one optimized with deque, plus an AI-generated performance comparison.

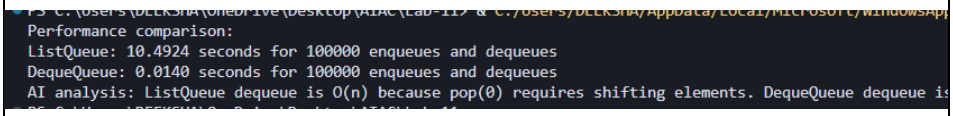
CODE:

```python
class ListQueue:
    """A simple queue implementation using Python lists."""
    def __init__(self):
        self.queue = []

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

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.queue.pop(0)

    def is_empty(self):
        return len(self.queue) == 0

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

from collections import deque
import time

class DequeQueue:
    """An optimized queue implementation using collections.deque."""
    def __init__(self):
        self.queue = deque()

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

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.queue.popleft()
```

```python
    def is_empty(self):
        return len(self.queue) == 0

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

def performance_comparison():
    N = 100000

    # Timing ListQueue
    lq = ListQueue()
    start_time = time.time()
    for i in range(N):
        lq.enqueue(i)
    for i in range(N):
        lq.dequeue()
    list_time = time.time() - start_time

    # Timing DequeQueue
    dq = DequeQueue()
    start_time = time.time()
    for i in range(N):
        dq.enqueue(i)
    for i in range(N):
        dq.dequeue()
    deque_time = time.time() - start_time

    print("Performance comparison:")
    print(f"ListQueue: {list_time:.4f} seconds for {N} enqueues and dequeues")
    print(f"DequeQueue: {deque_time:.4f} seconds for {N} enqueues and dequeues")
    print("AI analysis: ListQueue dequeue is O(n) because pop(0) requires shifting elements. DequeQueue dequeue is O(1). Therefore, DequeQueue is signif
```

```python
# Example execution of performance
performance_comparison()
```

OUTPUT:



Task 3: Singly Linked List with Traversal

- **Task**: Implement a **Singly Linked List** with operations: insert_at_end(), delete_value(), and traverse().
- **Instructions**:
  - Start with a simple class-based implementation (Node, LinkedList).
  - Use AI to generate inline comments explaining pointer updates (which are non-trivial).
  - Ask AI to suggest test cases to validate all operations.
- **Expected Output**:
  - A functional linked list implementation with clear comments explaining the logic of insertions and deletions.

PROMPT: write the python program Implement a **Singly Linked List** with clear comments explaining the logic of insertions and deletions.

CODE:

```python
class Node:
    """
    Represents a node in a singly linked list.
    Each node contains some data and a reference to the next node.
    """
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:
    """
    Implements a basic singly linked list with methods to:
    - Insert at the end
    - Delete a node by value
    - Traverse the 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 list.
        """
        new_node = Node(data)
        # If the list is empty, set the new node as the head
        if self.head is None:
            self.head = new_node
            print(f"Inserted {data} as the head.")
            return
        # Otherwise, traverse to the end and append
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
```

```
        print(f"Inserted {data} at the end.")

    def delete_value(self, value):
        """
        Deletes the first node with the specified value.
        If the value does not exist, nothing happens.
        """
        current = self.head
        prev = None
        # Traverse the list to find the node to delete
        while current:
            if current.data == value:
                if prev:
                    # Bypass the current node
                    prev.next = current.next
                else:
                    # If deleting the head node
                    self.head = current.next
                print(f"Deleted node with value {value}.")
                return
            prev = current
            current = current.next
        print(f"Value {value} not found in the list.")

    def traverse(self):
        """
        Prints all elements in the list.
        """
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" -> ".join(elements) if elements else "List is empty.")

# Example usage:
if __name__ == "__main__":
    sll = SinglyLinkedList()
    sll.traverse()          # Should show "List is empty."
    sll.insert_at_end(10)
    sll.insert_at_end(20)
    sll.insert_at_end(30)
    sll.traverse()          # 10 -> 20 -> 30
    sll.delete_value(20)
    sll.traverse()          # 10 -> 30
    sll.delete_value(100)   # Value not found
    sll.delete_value(10)
    sll.traverse()          # 30
    sll.delete_value(30)
    sll.traverse()          # List is empty.
```

OUTPUT:

```
List is empty.
Inserted 10 as the head.
Inserted 20 at the end.
Inserted 30 at the end.
10 -> 20 -> 30
Deleted node with value 20.
10 -> 30
Value 100 not found in the list.
Deleted node with value 10.
30
Deleted node with value 30.
List is empty.
```

Task 4: Binary Search Tree (BST)

- **Task**: Implement a **Binary Search Tree** with methods for insert(), search(), and inorder_traversal().
- **Instructions**:
  - Provide AI with a partially written Node and BST class.
  - Ask AI to complete missing methods and add docstrings.
  - Test with a list of integers and compare outputs of search() for present vs absent elements.
- **Expected Output**:
  - A BST class with clean implementation, meaningful docstrings, and correct traversal output.

PROMPT: write a python program Implement a **Binary Search Tree with** meaningful docstrings, and correct traversal output.

CODE:

```python
class TreeNode:
    """
    Represents a node in the Binary Search Tree.
    Each node contains some data and pointers to its left and right children.
    """
    def __init__(self, value):
        """
        Initialize a tree node.
        Args:
            value: The value to store in the node.
        """
        self.value = value
        self.left = None
        self.right = None

class BinarySearchTree:
    """
    Implements a Binary Search Tree with insert, search, and inorder_traversal methods.
    """

    def __init__(self):
        """Initializes an empty BST."""
        self.root = None

    def insert(self, value):
        """
        Insert a value into the BST.
        Args:
            value: The value to be inserted.
        """
        if self.root is None:
            self.root = TreeNode(value)
            print(f"Inserted {value} as root")
        else:
            self._insert_recursive(self.root, value)
```

```
    Helper for recursive insertion.
    Args:
        node: Current node to compare.
        value: Value to insert.
    """
    if value < node.value:
        if node.left is None:
            node.left = TreeNode(value)
            print(f"Inserted {value} to left of {node.value}")
        else:
            self._insert_recursive(node.left, value)
    elif value > node.value:
        if node.right is None:
            node.right = TreeNode(value)
            print(f"Inserted {value} to right of {node.value}")
        else:
            self._insert_recursive(node.right, value)
    else:
        print(f"Value {value} already exists. No duplicates allowed in BST.")

def search(self, value):
    """
    Search for a value in the BST.
    Args:
        value: The value to search for.
    Returns:
        True if found, False otherwise.
    """
    return self._search_recursive(self.root, value)

def _search_recursive(self, node, value):
    """Helper for recursive search."""
    if node is None:
```

```
            print(f"{value} not found in BST.")
            return False
        if value == node.value:
            print(f"Found {value} in BST.")
            return True
        elif value < node.value:
            return self._search_recursive(node.left, value)
        else:
            return self._search_recursive(node.right, value)

    def inorder_traversal(self):
        """
        Perform inorder traversal of the BST.
        Returns:
            List of values in inorder sequence.
        """
        result = []
        self._inorder_recursive(self.root, result)
        print("Inorder traversal:", " ".join(str(x) for x in result))
        return result

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

```
if __name__ == "__main__":
    """
    Example usage of BinarySearchTree with insert, search, and inorder_traversal:
    """

    bst = BinarySearchTree()

    # Insert values
    print("Inserting values into BST:")
    bst.insert(50)
    bst.insert(30)
    bst.insert(70)
    bst.insert(20)
    bst.insert(40)
    bst.insert(60)
    bst.insert(80)

    print("\nPerforming inorder traversal:")
    bst.inorder_traversal()      # Should print sorted order: 20 30 40 50 60 70 80

    print("\nSearching for values:")
    bst.search(60)               # Found
    bst.search(25)               # Not found
    bst.search(50)               # Found (root)
```

OUTPUT:

```
Inserting values into BST:
Inserted 50 as root
Inserted 30 to left of 50
Inserted 70 to right of 50
Inserted 20 to left of 30
Inserted 40 to right of 30
Inserted 60 to left of 70
Inserted 80 to right of 70

Performing inorder traversal:
Inorder traversal: 20 30 40 50 60 70 80

Searching for values:
Found 60 in BST.
25 not found in BST.
Found 50 in BST.
```

Task 5: Graph Representation and BFS/DFS Traversal

- **Task**: Implement a **Graph** using an adjacency list, with traversal methods BFS() and DFS().
- **Instructions**:
  - Start with an adjacency list dictionary.
  - Ask AI to generate BFS and DFS implementations with inline comments.
  - Compare recursive vs iterative DFS if suggested by AI.
- **Expected Output**:
  - A graph implementation with BFS and DFS traversal methods, with AI-generated comments explaining traversal steps.

PROMPT : write the python program Implement a **Graph** using an adjacency list with BFS and DFS traversal methods, with AI-generated comments explaining traversal steps.

OUTPUT:

```python
class Graph:
    """
    Graph implementation using an adjacency list.
    Supports adding edges and performing BFS and DFS traversals.
    """
    def __init__(self):
        # Initialize the adjacency list as a dictionary
        self.adj_list = {}

    def add_edge(self, u, v):
        """
        Adds an undirected edge between node u and node v.
        """
        if u not in self.adj_list:
            self.adj_list[u] = []
        if v not in self.adj_list:
            self.adj_list[v] = []
        # Add both edges since the graph is undirected
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)

    def bfs(self, start):
        """
        Breadth-First Search traversal from the starting node.
        Prints the order of traversal with explanatory comments.
        """
        from collections import deque

        visited = set()                 # Set to keep track of visited nodes
        queue = deque([start])          # Use a queue for BFS

        print(f"Starting BFS from node {start}")
        while queue:
            node = queue.popleft()      # Dequeue a node for exploration
            if node not in visited:
                print(f"Visiting: {node}")
                visited.add(node)
                # Enqueue all unvisited neighbors
                for neighbor in self.adj_list.get(node, []):
                    if neighbor not in visited:
```

```
                    print(f"  Queueing neighbor {neighbor} of {node}")
                    queue.append(neighbor)
        print("BFS traversal complete.")

    def dfs(self, start):
        """
        Depth-First Search traversal from the starting node.
        Prints the order of traversal with explanatory comments.
        """
        visited = set()

        print(f"Starting DFS from node {start}")
        def dfs_recursive(node):
            print(f"Visiting: {node}")
            visited.add(node)
            # Explore each neighbor recursively if not visited
            for neighbor in self.adj_list.get(node, []):
                if neighbor not in visited:
                    print(f"  Going deeper to neighbor {neighbor} of {node}")
                    dfs_recursive(neighbor)
        dfs_recursive(start)
        print("DFS traversal complete.")

# Example usage:
if __name__ == "__main__":
    g = Graph()
    # Add edges to the graph
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 3)
    g.add_edge(1, 4)
    g.add_edge(2, 5)
    g.add_edge(2, 6)

    print("Adjacency List Representation:")
    for node, neighbors in g.adj_list.items():
        print(f"{node}: {neighbors}")

    print("\nBFS Traversal Example:")
    g.bfs(0)

    print("\nDFS Traversal Example:")
    g.dfs(0)
```

OUTPUT:

```
Adjacency List Representation:
0: [1, 2]
1: [0, 3, 4]
2: [0, 5, 6]
3: [1]
4: [1]
5: [2]
6: [2]

BFS Traversal Example:
Starting BFS from node 0
Visiting: 0
  Queueing neighbor 1 of 0
  Queueing neighbor 2 of 0
Visiting: 1
  Queueing neighbor 3 of 1
  Queueing neighbor 4 of 1
Visiting: 2
  Queueing neighbor 5 of 2
  Queueing neighbor 6 of 2
Visiting: 3
Visiting: 4
Visiting: 5
Visiting: 6
BFS traversal complete.

DFS Traversal Example:
Starting DFS from node 0
Visiting: 0
  Going deeper to neighbor 1 of 0
Visiting: 1
  Going deeper to neighbor 3 of 1
Visiting: 3
  Going deeper to neighbor 4 of 1
Visiting: 4
  Going deeper to neighbor 2 of 0
Visiting: 2
  Going deeper to neighbor 5 of 2
Visiting: 5
  Going deeper to neighbor 6 of 2
Visiting: 6
DFS traversal complete.
```