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 code to implement a **Stack** class in Python with the following operations: push(), pop(), peek(), and is_empty(). And generate docstrings give ouput with A working Stack class with proper methods, Google-style docstrings, and inline comments for tricky parts.

TASK1.py

C: > Users > MEGHANA > OneDrive > Documents > AIAC > ASSIGNMENT-11.4 > TASK1.py > Stack

```python
class Stack:
    """A simple Stack implementation using Python list.

    Provides standard stack operations: push, pop, peek, and is_empty.

    Attributes:
        items (list): Internal container for stack elements.
    """

    def __init__(self):
        """Initializes 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)

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

        Returns:
            The item that was 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 top item from the stack without removing it.
```

TASK1.py

C: > Users > MEGHANA > OneDrive > Documents > AIAC > ASSIGNMENT-11.4 > TASK1.py > Stack

```python
class Stack:

    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():
            raise IndexError("Peek from empty stack")
        return self.items[-1]

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

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

# Example usage:
if __name__ == "__main__":
    s = Stack()
    print("Is stack empty?", s.is_empty())  # True
    s.push(10)
    s.push(20)
    print("Top item:", s.peek())  # 20
    print("Pop:", s.pop())  # 20
    print("Pop:", s.pop())  # 10
    print("Is stack empty?", s.is_empty())  # True
```

OUTPUT:

```
Is stack empty? True
Top item: 20
Pop: 20
Pop: 10
Is stack empty? True
PS C:\Users\MEGHANA\OneDrive\Documents\AIAC\ASSIGNMENT-11.4>
```

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 a python program that Implement a **Queue** with enqueue(), dequeue(), and is_empty() methods. First, implement using Python lists then to review performance and suggest a more efficient implementation (using collections.deque) there should be two verions of queue that is one with lists and one optimized with deque, plus an AI-generated performance comparison task2.py ×

C: > Users > MEGHANA > OneDrive > Documents > AIAC > ASSIGNMENT-11.4 > task2.py > ...

```python
class ListQueue:
    """Queue implementation using Python lists."""
    def __init__(self):
        self.items = []

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

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            raise IndexError("dequeue from empty queue")

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

class DequeQueue:
    """Queue implementation using collections.deque (optimized)."""
    def __init__(self):
        from collections import deque
        self.items = deque()

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

    def dequeue(self):
        if not self.is_empty():
            return self.items.popleft()
        else:
            raise IndexError("dequeue from empty queue")

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

def compare_performance():
    import time
```

task2.py ×

C: > Users > MEGHANA > OneDrive > Documents > AIAC > ASSIGNMENT-11.4 > task2.py > ...

```python
def compare_performance():
    print("Comparing performance of ListQueue vs DequeQueue for 10000 operations...\n")
    num_operations = 10000

    # ListQueue performance
    lq = ListQueue()
    start = time.time()
    for i in range(num_operations):
        lq.enqueue(i)
    for i in range(num_operations):
        lq.dequeue()
    end = time.time()
    listqueue_time = end - start
    print(f"ListQueue total time: {listqueue_time:.6f} seconds")

    # DequeQueue performance
    dq = DequeQueue()
    start = time.time()
    for i in range(num_operations):
        dq.enqueue(i)
    for i in range(num_operations):
        dq.dequeue()
    end = time.time()
    dequequeue_time = end - start
    print(f"DequeQueue total time: {dequequeue_time:.6f} seconds")

    print("\nAI-Generated Performance Comparison:")
    print("------------------------------------------------------------")
    print("ListQueue uses Python lists, and while appends are fast (O(1)),")
    print("dequeue operations are O(n) because removing from the front")
    print("requires shifting all elements. This becomes inefficient for large queues.")
    print("DequeQueue uses collections.deque, which is optimized for fast appends")
    print("and pops from both ends (O(1) for both enqueue and dequeue).")
    print("As demonstrated above, DequeQueue usually outperforms ListQueue")
    print("on large numbers of enqueue/dequeue operations.")
    print("------------------------------------------------------------")
```

task2.py ×

C: > Users > MEGHANA > OneDrive > Documents > AIAC > ASSIGNMENT-11.4 > task2.py > ...

```python
def compare_performance():
    return {"ListQueue_time": listqueue_time, "DequeQueue_time": dequequeue_time}

if __name__ == "__main__":
    compare_performance()
```

OUTPUT:

```
Comparing performance of ListQueue vs DequeQueue for 10000 operations...

ListQueue total time: 0.101871 seconds
DequeQueue total time: 0.002549 seconds

AI-Generated Performance Comparison:
--------------------------------------------------------
ListQueue uses Python lists, and while appends are fast (O(1)),
dequeue operations are O(n) because removing from the front
requires shifting all elements. This becomes inefficient for large queues.
DequeQueue uses collections.deque, which is optimized for fast appends
and pops from both ends (O(1) for both enqueue and dequeue).
As demonstrated above, DequeQueue usually outperforms ListQueue
on large numbers of enqueue/dequeue operations.
--------------------------------------------------------
```

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 a python program that Implement a **Singly Linked List** with operations: insert_at_end(), delete_value(), and traverse().generate inline comments explaining pointer updates (which are non-trivial) and suggest test cases to validate all operations

```
task2.py    task3.py ×
C: > Users > MEGHANA > OneDrive > Documents > AIAC > ASSIGNMENT-11.4 > task3.py > ...
class Node:
    """A node in a singly linked list."""
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:
    """Implementation of singly linked list with fundamental operations."""
    def __init__(self):
        self.head = None

    def insert_at_end(self, value):
        """Insert a new node with 'value' at the end of the list."""
        new_node = Node(value)
        if not self.head:
            # If the list is empty, new node becomes the head.
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            # At the end, set the last node's next to new_node.
            current.next = new_node

    def delete_value(self, value):
        """Delete the first node in the list containing the specified value."""
        current = self.head
        prev = None
        while current:
            if current.data == value:
                if prev:
                    # Bypass current node by pointing previous node's next to current's next.
                    prev.next = current.next
                else:
                    # Node to delete is the head: move head to next node.
                    self.head = current.next
                return True  # Value found and deleted.
```

task2.py   task3.py ×

C: > Users > MEGHANA > OneDrive > Documents > AIAC > ASSIGNMENT-11.4 > task3.py > ...

```python
class SinglyLinkedList:
    def delete_value(self, value):
                return True  # Value found and deleted.
            prev = current
            current = current.next
        return False  # Value not found.

    def traverse(self):
        """Return a list of all node values from head to end."""
        values = []
        current = self.head
        while current:
            values.append(current.data)
            current = current.next
        return values

# -------------------- Test Cases --------------------
# 1. Insert at end in empty and non-empty lists
# 2. Delete head, middle, tail, and non-existent values
# 3. Traverse after each modification

if __name__ == "__main__":
    ll = SinglyLinkedList()
    print("Initial traverse (should be []):", ll.traverse())
    ll.insert_at_end(10)
    ll.insert_at_end(20)
    ll.insert_at_end(30)
    print("After inserts (should be [10, 20, 30]):", ll.traverse())

    print("Delete 10 (head) (should be True):", ll.delete_value(10))
    print("Traverse after delete head (should be [20, 30]):", ll.traverse())

    print("Delete 30 (tail) (should be True):", ll.delete_value(30))
    print("Traverse after delete tail (should be [20]):", ll.traverse())

    print("Delete 20 (only node / middle) (should be True):", ll.delete_value(20))
```

```python
print("Traverse after delete last (should be []):", ll.traverse())

print("Delete 42 (non-existent) (should be False):", ll.delete_value(42))
print("Traverse after trying to delete non-existent:", ll.traverse())

# Re-insert and traverse
ll.insert_at_end(100)
ll.insert_at_end(200)
print("After more inserts (should be [100, 200]):", ll.traverse())
```

OUTPUT:

```
Initial traverse (should be []): []
After inserts (should be [10, 20, 30]): [10, 20, 30]
Delete 10 (head) (should be True): True
Traverse after delete head (should be [20, 30]): [20, 30]
Traverse after delete head (should be [20, 30]): [20, 30]
Delete 30 (tail) (should be True): True
Traverse after delete tail (should be [20]): [20]
Delete 20 (only node / middle) (should be True): True
Traverse after delete last (should be []): []
Delete 42 (non-existent) (should be False): False
Traverse after trying to delete non-existent: []
Traverse after delete head (should be [20, 30]): [20, 30]
Delete 30 (tail) (should be True): True
Traverse after delete tail (should be [20]): [20]
Delete 20 (only node / middle) (should be True): True
Traverse after delete last (should be []): []
Delete 42 (non-existent) (should be False): False
Traverse after delete head (should be [20, 30]): [20, 30]
Delete 30 (tail) (should be True): True
Traverse after delete tail (should be [20]): [20]
Delete 20 (only node / middle) (should be True): True
Traverse after delete head (should be [20, 30]): [20, 30]
Delete 30 (tail) (should be True): True
Traverse after delete tail (should be [20]): [20]
Traverse after delete head (should be [20, 30]): [20, 30]
Delete 30 (tail) (should be True): True
Traverse after delete head (should be [20, 30]): [20, 30]
Traverse after delete head (should be [20, 30]): [20, 30]
Delete 30 (tail) (should be True): True
Traverse after delete head (should be [20, 30]): [20, 30]
Delete 30 (tail) (should be True): True
Traverse after delete head (should be [20, 30]): [20, 30]
Delete 30 (tail) (should be True): True
Traverse after delete head (should be [20, 30]): [20, 30]
Delete 30 (tail) (should be True): True
Traverse after delete head (should be [20, 30]): [20, 30]
Traverse after delete head (should be [20, 30]): [20, 30]
Delete 30 (tail) (should be True): True
```

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 that Implement a **Binary Search Tree** with methods for insert(), search(), and inorder_traversal()with a partially written Node and BST class complete missing methods and add docstrings and Test with a list of integers and compare outputs of search() for present vs absent elements return output with A BST class with clean implementation, meaningful docstrings, and correct traversal output task2.py task3.py task4.py ×

C: > Users > MEGHANA > OneDrive > Documents > AIAC > ASSIGNMENT-11.4 > task4.py > ...

```python
class Node:
    """A node in the Binary Search Tree."""

    def __init__(self, data):
        """
        Initialize node with data and children.
        :param data: Value for this node.
        """
        self.data = data
        self.left = None
        self.right = None


class BST:
    """
    Binary Search Tree (BST) implementation.

    Supports element insertion, searching, and in-order traversal.
    """

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

    def insert(self, value):
        """
        Insert a value into the BST.
        :param value: Value to insert.
        """
        if self.root is None:
            self.root = Node(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        """
        Helper method to insert recursively.
```

```python
    :param node: Current node in the tree.
    :param value: Value to insert.
    """
    if value < node.data:
        if node.left is None:
            node.left = Node(value)
        else:
            self._insert_recursive(node.left, value)
    elif value > node.data:
        if node.right is None:
            node.right = Node(value)
        else:
            self._insert_recursive(node.right, value)
    # If value == node.data, do not insert again (no duplicates in BST).

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

def _search_recursive(self, node, value):
    """
    Helper method to search recursively.
    :param node: Current node.
    :param value: Value to search for.
    :return: True if found, False otherwise.
    """
    if node is None:
        return False
    if value == node.data:
        return True
```

```python
            return True
        elif value < node.data:
            return self._search_recursive(node.left, value)
        else:  # value > (method) def _search_recursive(
            return self.        node: Any,                          ue)
                                value: Any
    def inorder_traversa ) -> (Any | bool)
        """
        Perform in-order Helper method to search recursively.  ues.
        :return: List of :param node: Current node.
        """              :param value: Value to search for.
        result = []      :return: True if found, False otherwise.
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        """
        Helper to recursively traverse in-order.
        :param node: Node to traverse from.
        :param result: List to append values to.
        """
        if node is not None:
            self._inorder_recursive(node.left, result)
            result.append(node.data)
            self._inorder_recursive(node.right, result)


if __name__ == "__main__":
    # Test with a list of integers
    numbers = [7, 3, 9, 1, 5, 8, 10]
    bst = BST()
    for num in numbers:
        bst.insert(num)

    print("In-order Traversal Output (should be sorted):")
```

```python
print("In-order Traversal Output (should be sorted):")
print(bst.inorder_traversal())

# Search for present and absent elements
test_values = [5, 7, 2, 10, 12]
for val in test_values:
    found = bst.search(val)
    if found:
        print(f"Value {val} FOUND in BST.")
    else:
        print(f"Value {val} NOT FOUND in BST.")
```

OUPUT:

```
In-order Traversal Output (should be sorted):
[1, 3, 5, 7, 8, 9, 10]
Value 5 FOUND in BST.
Value 7 FOUND in BST.
Value 2 NOT FOUND in BST.
Value 10 FOUND in BST.
Value 12 NOT FOUND in BST.
PS C:\Users\MEGHANA\OneDrive\Documents\AIAC\ASSIGNMENT-11.4>
```

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 a python program that Implement a **Graph** using an adjacency list, with traversal methods BFS() and DFS()Start with an adjacency list dictionary generate BFS and DFS implementations with inline comments. The output should be A graph implementation with BFS and DFS traversal methods, with AI-generated comments explaining traversal steps.

task5.py ✕

C: > Users > MEGHANA > OneDrive > Documents > AIAC > ASSIGNMENT-11.4 > task5.py > ...

```python
class Graph:
    def __init__(self):
        # Initializes an empty adjacency list to represent the graph
        self.adj_list = {}

    def add_edge(self, src, dest):
        # Adds an edge from src to dest (undirected graph)
        if src not in self.adj_list:
            self.adj_list[src] = []
        if dest not in self.adj_list:
            self.adj_list[dest] = []
        self.adj_list[src].append(dest)
        self.adj_list[dest].append(src)

    def bfs(self, start):
        # Performs Breadth-First Search (BFS) starting from the given node
        visited = set()          # Tracks visited nodes to avoid repetition
        queue = []               # Queue for BFS
        traversal = []           # List to store the order of BFS traversal

        queue.append(start)
        visited.add(start)

        while queue:
            # Take the first node from the queue
            node = queue.pop(0)
            traversal.append(node)    # Record the node in BFS order

            # Explore all adjacent nodes (neighbors)
            for neighbor in self.adj_list.get(node, []):
                if neighbor not in visited:
                    queue.append(neighbor)  # Enqueue the unvisited neighbor
                    visited.add(neighbor)   # Mark as visited

        return traversal

    def dfs(self, start):
```

```python
        # Performs Depth-First Search (DFS) starting from the given node
        visited = set()           # Tracks visited nodes
        traversal = []            # List to store the order of DFS traversal

        def dfs_recursive(node):
            visited.add(node)         # Mark this node as visited
            traversal.append(node)    # Record the node in DFS order

            # Visit all the neighbors recursively
            for neighbor in self.adj_list.get(node, []):
                if neighbor not in visited:
                    dfs_recursive(neighbor)

        dfs_recursive(start)
        return traversal

# Example Usage:
if __name__ == '__main__':
    # Creating a graph instance
    g = Graph()
    g.add_edge('A', 'B')
    g.add_edge('A', 'C')
    g.add_edge('B', 'D')
    g.add_edge('C', 'D')
    g.add_edge('D', 'E')

    # Performing BFS and DFS traversals
    print("BFS Traversal:", g.bfs('A'))   # Output: BFS Traversal: ['A', 'B', 'C', 'D', 'E']
    print("DFS Traversal:", g.dfs('A'))   # Output: DFS Traversal: ['A', 'B', 'D', 'C', 'E']
```

OUTPUT:

```
BFS Traversal: ['A', 'B', 'C', 'D', 'E']
DFS Traversal: ['A', 'B', 'D', 'C', 'E']
BFS Traversal: ['A', 'B', 'C', 'D', 'E']
DFS Traversal: ['A', 'B', 'D', 'C', 'E']
PS C:\Users\MEGHANA\OneDrive\Documents\AIAC\ASSIGNMENT-11.4>
```
