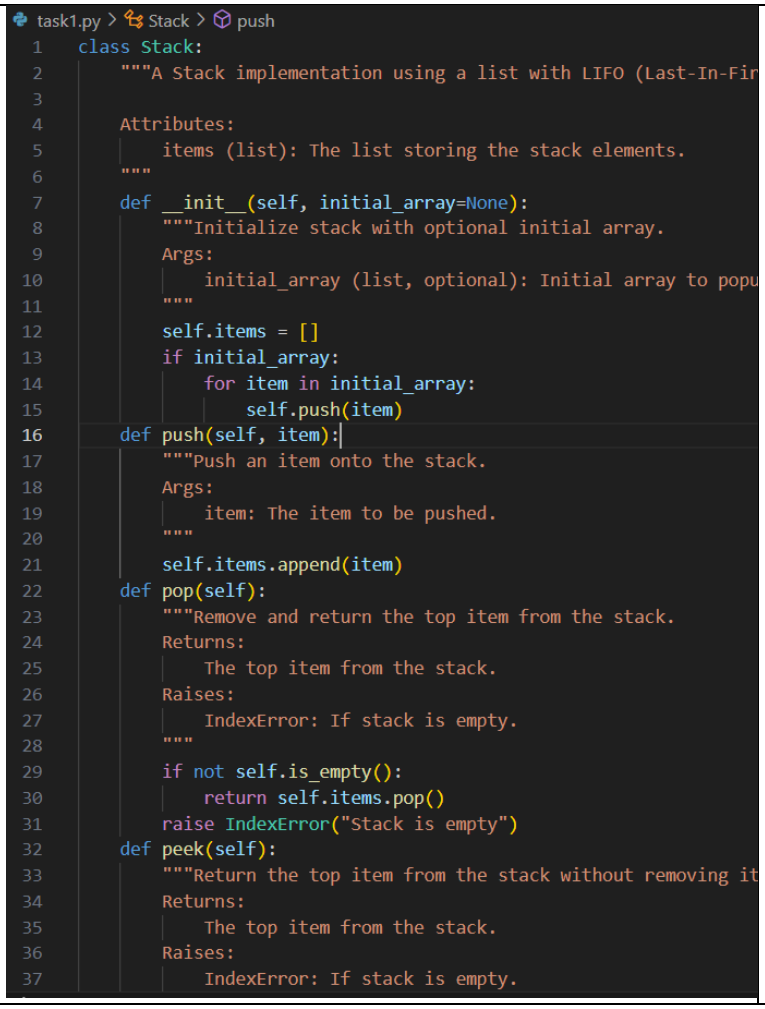
SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:**2025-2026
Course Coordinator Name		Venkataramana Veeramsetty	
Instructor(s) Name		Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar 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**: o Ask AI to generate code skeleton with docstrings. o Test stack operations using sample data. o Request AI to suggest optimizations or alternative implementations (e.g., using collections.deque). • **Expected Output**: o A working Stack class with proper methods, Google-style docstrings, and inline comments for tricky parts. o **PROMPT: implement a python program a stack(LIFO) with following operations push(),pop(),peek() and is_empty() take a array and perform operations according to it.provide docstrings and inlinecomments for the code** o **CODE:**	


task1.py > Stack > push
class Stack:
"""A Stack implementation using a list with LIFO (Last-In-Fir
Attributes:
items (list): The list storing the stack elements.
"""
def __init__(self, initial_array=None):
"""Initialize stack with optional initial array.
Args:
initial_array (list, optional): Initial array to popu
"""
self.items = []
if initial_array:
for item in initial_array:
self.push(item)
def push(self, item):
"""Push an item onto the stack.
Args:
item: The item to be pushed.
"""
self.items.append(item)
def pop(self):
"""Remove and return the top item from the stack.
Returns:
The top item from the stack.
Raises:
IndexError: If stack is empty.
"""
if not self.is_empty():
return self.items.pop()
raise IndexError("Stack is empty")
def peek(self):
"""Return the top item from the stack without removing it
Returns:
The top item from the stack.
Raises:
IndexError: If stack is empty.

task1.py > Stack > is_empty

```
class Stack:
    def peek(self):
            IndexError: If stack is empty."""
        if not self.is_empty():
            return self.items[-1]
        raise IndexError("Stack is empty")
    def is_empty(self):
        """Check if the stack is empty.
        Returns:
            bool: True if stack is empty, False otherwise."""
        return len(self.items) == 0
    def __str__(self):
        """Return string representation of the stack.
        Returns:
            str: String representation of stack contents.  """
        return str(self.items)
# Test the implementation
if __name__ == "__main__":
    # Initial array
    initial_array = [3, 5, 8, 9, 6, 1]
    # Create stack with initial array
    stack = Stack(initial_array)
    # Test operations
    print("Initial stack:", stack)
    # Push a new element
    stack.push(7)
    print("After pushing 7:", stack)
    # Pop an element
    popped_item = stack.pop()
    print(f"Popped item: {popped_item}")
    print("After popping:", stack)
    # Peek at top element
    top_item = stack.peek()
    print(f"Top item (peek): {top_item}")
    # Check if empty
    print("Is stack empty?", stack.is_empty())
```

- **OUTPUT:**

```
PS C:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 11.4> & "C:/Program Files/Python313/python.exe" "c:/Users/Vyshn/OneDrive/D
1.py"
Initial stack: [3, 5, 8, 9, 6, 1]
After pushing 7: [3, 5, 8, 9, 6, 1, 7]
Popped item: 7
After popping: [3, 5, 8, 9, 6, 1]
Top item (peek): 1
Is stack empty? False
PS C:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 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.
  - CODE:

Welcome | task1.py | task2.py

task2.py > performance_test

```python
from collections import deque
import time
# Queue implementation using lists
class ListQueue:
    def __init__(self):
        self.queue = []
    def enqueue(self, item):
        self.queue.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        raise IndexError("dequeue from empty queue")
    def is_empty(self):
        return len(self.queue) == 0
# Queue implementation using collections.deque
class DequeQueue:
    def __init__(self):
        self.queue = deque()
    def enqueue(self, item):
        self.queue.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()
        raise IndexError("dequeue from empty queue")
    def is_empty(self):
        return len(self.queue) == 0
# Performance comparison
def performance_test():
    list_queue = ListQueue()
    deque_queue = DequeQueue()
    # Enqueue items from console
    while True:
        item = input("Enter an item to enqueue (or type 'exit'
        if item.lower() == 'exit':
            break
        list_queue.enqueue(item)
        deque_queue.enqueue(item)
```

```python
        list_queue.enqueue(item)
        deque_queue.enqueue(item)
    # Dequeue test
    print("Dequeuing items from both que
    while not list_queue.is_empty():
        list_queue.dequeue()
    while not deque_queue.is_empty():
        deque_queue.dequeue()
if __name__ == "__main__":
    performance_test()
```

- OUTPUT:

```
                                                & "C:/Program Files/Python313/p
/Vyshn/OneDrive/Desktop/AIAC/lab 11.4/task2.py"
List Queue Enqueue Time: 0.000609 seconds
Deque Queue Enqueue Time: 0.000643 seconds
List Queue Dequeue Time: 0.131247 seconds
Deque Queue Dequeue Time: 0.001681 seconds
PS C:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 11.4> & "C:/Program Files/Python313/p
/Vyshn/OneDrive/Desktop/AIAC/lab 11.4/task2.py"
Enter an item to enqueue (or type 'exit' to stop): stop
Enter an item to enqueue (or type 'exit' to stop): exit
Dequeuing items from both queues...
PS C:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 11.4>
```

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.
  - CODE:

-

```python
class Node:
    def __init__(self, data):
        self.data = data  # Store the data
        self.next = None  # Initialize the next pointer to None
class LinkedList:
    def __init__(self):
        self.head = None  # Initialize the head of the list to None
    def insert_at_end(self, data):
        new_node = Node(data)  # Create a new node
        if not self.head:  # If the list is empty
            self.head = new_node  # Set the new node as the head
            return
        last_node = self.head  # Start from the head
        while last_node.next:  # Traverse to the last node
            last_node = last_node.next  # Move to the next node
        last_node.next = new_node  # Update the last node's next pointer to
    def delete_value(self, value):
        current_node = self.head  # Start from the head
        if current_node and current_node.data == value:  # Check if the hea
            self.head = current_node.next  # Update head to the next node
            return
        prev_node = None  # Initialize previous node
        while current_node and current_node.data != value:  # Traverse the
            prev_node = current_node  # Keep track of the previous node
            current_node = current_node.next  # Move to the next node
        if current_node:  # If the value was found
            prev_node.next = current_node.next  # Bypass the current node
    def traverse(self):
        current_node = self.head  # Start from the head
        while current_node:  # Traverse until the end of the list
            print(current_node.data, end=" -> ")  # Print the current node'
            current_node = current_node.next  # Move to the next node
        print("None")  # Indicate the end of the list
# Test cases with console input
if __name__ == "__main__":
    ll = LinkedList()
    while True:
```

(parameter) self: Self@LinkedList

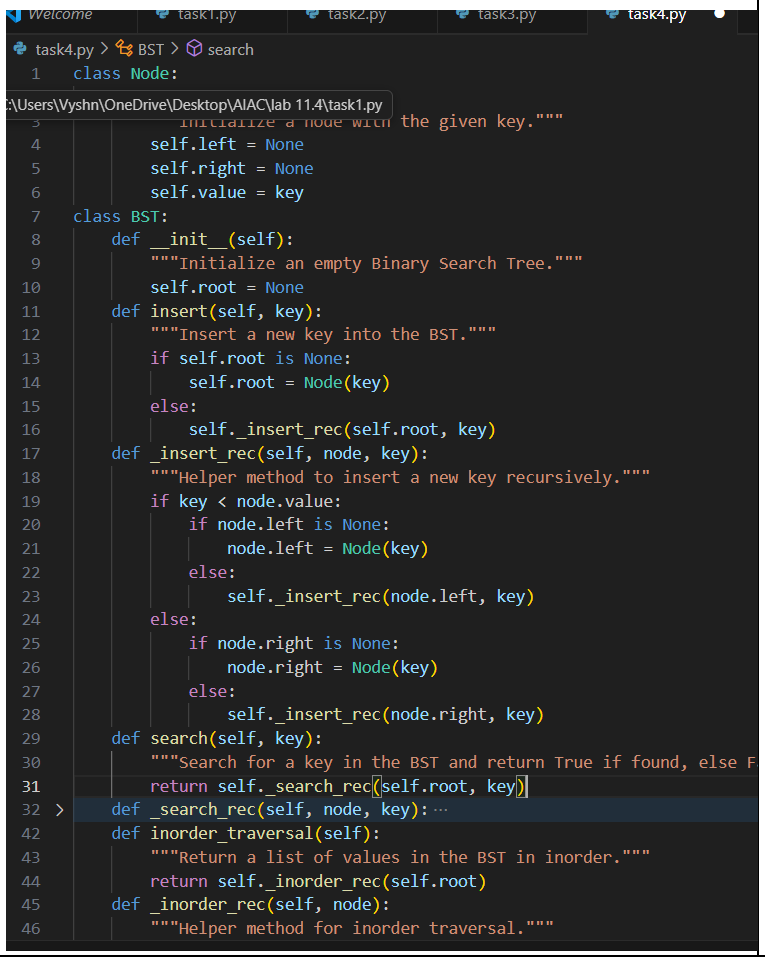
```python
if __name__ == "__main__":
    ll = LinkedList()
    while True:
        action = input("Enter 'insert' to add a value, 'delete' to remove a val
        if action == 'insert':
            value = int(input("Enter a value to insert: "))
            ll.insert_at_end(value)
        elif action == 'delete':
            value = int(input("Enter a value to delete: "))
            ll.delete_value(value)
        elif action == 'traverse':
            ll.traverse()  # Display the current list
        elif action == 'exit':
            break
        else:
            print("Invalid action. Please try again.")
```

- OUTPUT:

```
PS C:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 11.4>
PS C:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 11.4> & "C:/Program Files/Python313/py
/Vyshn/OneDrive/Desktop/AIAC/lab 11.4/task3.py"
Enter 'insert' to add a value, 'delete' to remove a value, or 'traverse' to displa
xit' to quit): insert
Enter a value to insert: 5
Enter 'insert' to add a value, 'delete' to remove a value, or 'traverse' to displa
xit' to quit): insert
Enter a value to insert: 8
Enter 'insert' to add a value, 'delete' to remove a value, or 'traverse' to displa
xit' to quit): traverse
5 -> 8 -> None
Enter 'insert' to add a value, 'delete' to remove a value, or 'traverse' to displa
xit' to quit): exit
```

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.
  - CODE:
  -



```
Welcome  task1.py  task2.py  task3.py  task4.py
task4.py > BST > search
1   class Node:
C:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 11.4\task1.py
                Initialize a node with the given key."""
4           self.left = None
5           self.right = None
6           self.value = key
7   class BST:
8       def __init__(self):
9           """Initialize an empty Binary Search Tree."""
10          self.root = None
11      def insert(self, key):
12          """Insert a new key into the BST."""
13          if self.root is None:
14              self.root = Node(key)
15          else:
16              self._insert_rec(self.root, key)
17      def _insert_rec(self, node, key):
18          """Helper method to insert a new key recursively."""
19          if key < node.value:
20              if node.left is None:
21                  node.left = Node(key)
22              else:
23                  self._insert_rec(node.left, key)
24          else:
25              if node.right is None:
26                  node.right = Node(key)
27              else:
28                  self._insert_rec(node.right, key)
29      def search(self, key):
30          """Search for a key in the BST and return True if found, else F
31          return self._search_rec(self.root, key)
32  >   def _search_rec(self, node, key): ···
42      def inorder_traversal(self):
43          """Return a list of values in the BST in inorder."""
44          return self._inorder_rec(self.root)
45      def _inorder_rec(self, node):
46          """Helper method for inorder traversal."""
```

```
        """Helper method for inorder traversal."""
        return (self._inorder_rec(node.left) if node.left else []) +
                [node.value] + \
                (self._inorder_rec(node.right) if node.right else [])
# Testing the BST implementation
if __name__ == "__main__":
    bst = BST()
    numbers = [7, 3, 9, 1, 5, 8, 10]
    for number in numbers:
        bst.insert(number)
    print("Inorder Traversal:", bst.inorder_traversal())
    print("Search for 5:", bst.search(5))  # Should return True
    print("Search for 6:", bst.search(6))  # Should return False
```

OUTPUT:

```
PS C:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 11.4>  c:; cd 'c:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 1
1.4'; & 'c:\Program Files\Python313\python.exe' 'c:\Users\Vyshn\.vscode\extensions\ms-python.debugpy-
2025.14.1-win32-x64\bundled\libs\debugpy\launcher' '51108' '--' 'c:\Users\Vyshn\OneDrive\Desktop\AIAC
\lab 11.4\task4.py'
Inorder Traversal: [1, 3, 5, 7, 8, 9, 10]
Search for 5: True
Search for 6: False
```

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.
  - CODE:

Welcome | task1.py | task2.py | task3.py | task4.py | task5.py

task5.py > Graph > __init__

```python
class Graph:
    def __init__(self):
        # Initialize an empty adjacency list
        self.adjacency_list = {}
    def add_edge(self, u, v):
        # Add an edge from vertex u to vertex v
        if u not in self.adjacency_list:
            self.adjacency_list[u] = []
        if v not in self.adjacency_list:
            self.adjacency_list[v] = []
        self.adjacency_list[u].append(v)
        self.adjacency_list[v].append(u)  # For undirected graph
    def bfs(self, start):
        # Perform Breadth-First Search (BFS) starting from the given vertex
        visited = set()  # Keep track of visited nodes
        queue = [start]  # Initialize the queue with the start node
        bfs_order = []  # List to store the order of traversal
        while queue:
            vertex = queue.pop(0)  # Dequeue a vertex
            if vertex not in visited:
                visited.add(vertex)  # Mark it as visited
                bfs_order.append(vertex)  # Add to the BFS order
                # Enqueue all unvisited neighbors
                queue.extend(neighbor for neighbor in self.adjacency_list[vertex] if ne
        return bfs_order
    def dfs_recursive(self, vertex, visited=None):
        # Perform Depth-First Search (DFS) recursively
        if visited is None:
            visited = set()  # Initialize visited set
        visited.add(vertex)  # Mark the vertex as visited
        dfs_order = [vertex]  # List to store the order of traversal
        # Recur for all the neighbors
        for neighbor in self.adjacency_list[vertex]:
            if neighbor not in visited:
                dfs_order.extend(self.dfs_recursive(neighbor, visited))
        return dfs_order
    def dfs_iterative(self, start):
```

Ln 4, C

```python
        return dfs_order
    def dfs_iterative(self, start):
        # Perform Depth-First Search (DFS) iteratively
        visited = set()  # Keep track of visited nodes
        stack = [start]  # Initialize the stack with the start node
        dfs_order = []  # List to store the order of traversal
        while stack:
            vertex = stack.pop()  # Pop a vertex from the stack
            if vertex not in visited:
                visited.add(vertex)  # Mark it as visited
                dfs_order.append(vertex)  # Add to the DFS order
                # Push all unvisited neighbors onto the stack
                stack.extend(neighbor for neighbor in self.adjacency_list[vertex]
        return dfs_order
# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(1, 2)
    g.add_edge(1, 3)
    g.add_edge(2, 4)
    g.add_edge(3, 5)
    print("BFS:", g.bfs(1))  # Output: BFS traversal starting from vertex 1
    print("DFS (Recursive):", g.dfs_recursive(1))  # Output: DFS traversal startin
    print("DFS (Iterative):", g.dfs_iterative(1))  # Output: DFS traversal startin
```

- OUTPUT:

```
PS C:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 11.4>  c:; cd 'c:\Users\Vyshn\OneDrive\Desktop\AIAC
1.4'; & 'c:\Program Files\Python313\python.exe' 'c:\Users\Vyshn\.vscode\extensions\ms-python.de
2025.14.1-win32-x64\bundled\libs\debugpy\launcher' '60003' '--' 'c:\Users\Vyshn\OneDrive\Deskto
\lab 11.4\task5.py'
BFS: [1, 2, 3, 4, 5]
DFS (Recursive): [1, 2, 4, 3, 5]
DFS (Iterative): [1, 3, 5, 2, 4]
PS C:\Users\Vyshn\OneDrive\Desktop\AIAC\lab 11.4>
```