**AI ASSISTED CODING **

***Lab 11 – Data Structures with AI: Implementing Fundamental Structures***

Name:D.Nagamrutha

Roll no:2503A51L06

**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.

**Prompt:**

Generate a code that implements stack with push, pop, peek and is\_empty operations.

**Code Generated:**

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Observation:**

This code shows a simple stack implementation in Python using a list. It supports the basic stack operations: push (add an item), pop (remove the top item), peek (look at the top item without removing it), and is\_empty (check if the stack has no items). The sample test at the bottom demonstrates how the stack works by pushing numbers, peeking at the top, popping an element, and checking if it is empty. The code also gives a suggestion that for larger stacks, using collections. Deque would be more efficient than a list, since it handles stack-like operations faster.

**Task 2: Queue Implementation with Performance Review**• Task: Implement a Queue with enqueue(), dequeue() and is\_empty()  
methods.  
**• Instructions:**  
o First, implement using Python lists.  
o Then, ask AI to review performance and suggest a more  
efficient implementation (using collections. Deque).  
**• Expected Output:**  
o Two versions of a queue: one with lists and one optimized with  
deque, plus an AI-generated performance comparison.

**Prompt:**

Implement queue using python list with enqueue, dequeue and is\_empty methods

A screen shot of a computer program

AI-generated content may be incorrect.**Code generated:**

Suggest a more efficient implementation (using collections. Deque).

**Code Generated:**

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

**A screen shot of a computer

AI-generated content may be incorrect.Output:**

**Observation:**

The code first implements a queue using Python lists, which works but can be inefficient because removing items from the front takes extra time. The optimized version uses collections. Deque, which is faster for adding and removing elements at both ends. Overall, the deque-based queue is more efficient and better suited for larger datasets compared to the list-based implementation.

**Task 3: Singly Linked List with Traversal**• Task: Implement a Singly Linked List with operations:  
insert\_at\_end(), delete\_value(), and traverse().  
**• Instructions:**  
o Start with a simple class-based implementation (Node,  
LinkedList).  
o Use AI to generate inline comments explaining pointer updates  
(which are non-trivial).  
o Ask AI to suggest test cases to validate all operations.  
**• Expected Output:**  
o A functional linked list implementation with clear comments  
explaining the logic of insertions and deletions

**Prompt:**

Implement a singly linked list with insert\_at\_end, delete-value and traverse. Generate inline comments explain pointer updates and suggest test cases.

A screen shot of a computer program

AI-generated content may be incorrect.**Code Generated:**

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Observation:**

This code implements a singly linked list with basic operations: inserting nodes at the end, deleting a node by value, and traversing the list to display elements. It correctly handles edge cases like deleting the head, middle, or last node, as well as attempting to delete a non-existing value. The inline comments and suggested test cases make it clear and easy to understand, ensuring the implementation can be validated thoroughly.

**Task 4: Binary Search Tree (BST)**  
• Task: Implement a Binary Search Tree with methods for insert(),  
search(), and inorder\_traversal().  
**• Instructions:**  
o Provide AI with a partially written Node and BST class.

o Ask AI to complete missing methods and add docstrings.  
o Test with a list of integers and compare outputs of search() for  
present vs absent elements.  
**• Expected Output:**  
o A BST class with clean implementation, meaningful docstrings,  
and correct traversal output.

**Prompt:**

# This is one box of the tree

class Node:

def \_\_init\_\_(self, value):

self.value = value # the number we keep here

self.left = None # left child

self.right = None # right child

# This is the whole tree

class BST:

def \_\_init\_\_(self):

self.root = None # at first tree is empty

def insert(self, value):

"""

Put a number inside the tree

Steps:

- If no root yet, make this the root

- Otherwise, compare with root and go left or right

- Keep going until we find an empty spot

"""

# not written yet

pass

def search(self, value):

"""

Look for a number in the tree

Steps:

- Start at the root

- If number is smaller, go left

- If bigger, go right

- If equal, found it

"""

# not written yet

pass

def inorder(self, node=None):

"""

Visit the tree in sorted order

Steps:

- Go left side

- Visit the current node

- Go right side

"""

# not written yet

pass

Complete missing methods and add docstrings.

Test with a list of integers and compare outputs of search() for

present vs absent elements BST class with clean implementation, meaningful docstrings,

and correct traversal output.

A screen shot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.**Code Generated:**

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Observation:**

This code provides a clear and complete Binary Search Tree (BST) implementation in Python. It defines a Node class to represent each element of the tree and a BST class to manage the overall structure. The insert method ensures that values are placed in the correct position based on BST rules, keeping smaller values to the left and larger values to the right. The search method recursively checks for the presence of a given value, returning True if found and False otherwise. The inorder traversal method visits nodes in sorted order, demonstrating one of the key properties of a BST. The test section inserts multiple values to build a sample tree, performs an inorder traversal to verify the sorted order, and tests the search function with both existing and non-existing values. Overall, this code is a well-structured example of BST operations with proper recursive handling and test validation.

**Task 5: Graph Representation and BFS/DFS Traversal**• Task: Implement a Graph using an adjacency list, with traversal  
methods BFS() and DFS().  
**• Instructions:**  
o Start with an adjacency list dictionary.  
o Ask AI to generate BFS and DFS implementations with inline  
comments.  
o Compare recursive vs iterative DFS if suggested by AI.  
**• Expected Output:**  
o A graph implementation with BFS and DFS traversal methods,  
with AI-generated comments explaining traversal steps.

**Prompt:**

Implement a graph using an adjacency list with BFS,DFS methods with inline comments.

**Code Generated:**

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.**Observation:**

This code defines a graph using an adjacency list and supports traversals with BFS and DFS (both recursive and iterative). BFS uses a queue to explore nodes level by level, while DFS explores depth-first using recursion or a stack. The implementation shows how edges are added in an undirected graph and demonstrates the different traversal methods on a sample graph, highlighting BFS for breadth exploration and DFS for depth exploration.