AI\_Assisted\_Coding  
 Assignment:11.4

HTNO:2403A51286

Q1:

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:

Implement a Python class for a Stack (Last-In-First-Out) with methods:

- push(item)

- pop()

- peek()

- is\_empty()

Include:

- Google-style docstrings for each method

- Inline comments for tricky parts

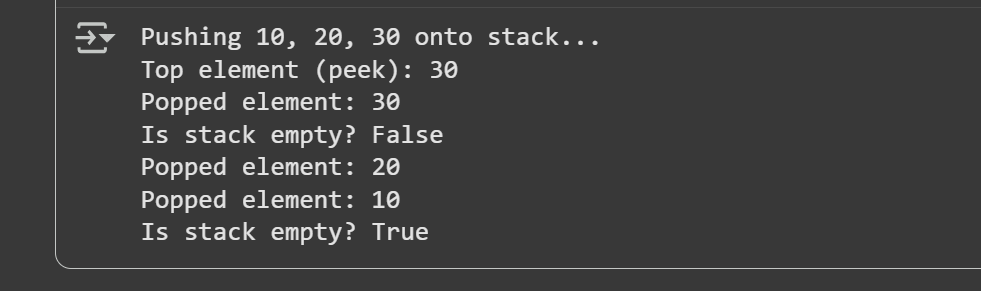
- A simple test example with sample data

- At the end, suggest possible optimizations (like using collections.deque).

Code:



Output:



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 a Queue class in Python with methods:

- enqueue(item)

- dequeue()

- is\_empty()

Step 1: Implement using Python lists.

Step 2: Implement an optimized version using collections.deque.

Include:

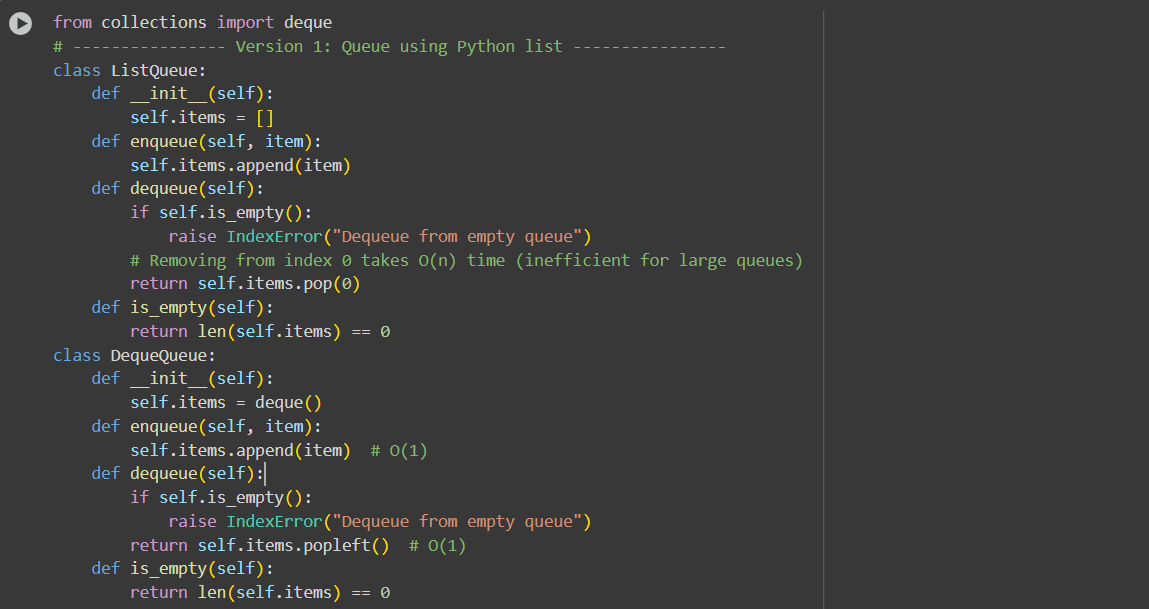
- Google-style docstrings

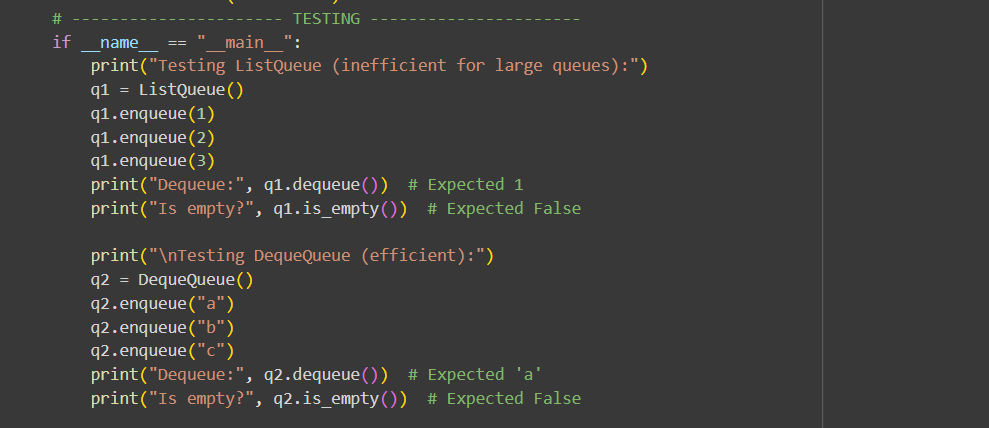
- Inline comments for tricky parts

- A small test example with sample data

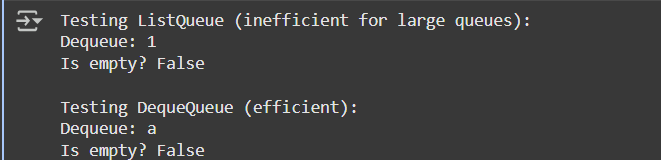
- A performance comparison between list and deque implementations

Code:





Output:



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 in Python with:

- insert\_at\_end(value)

- delete\_value(value)

- traverse()

Include:

- A Node class and a LinkedList class

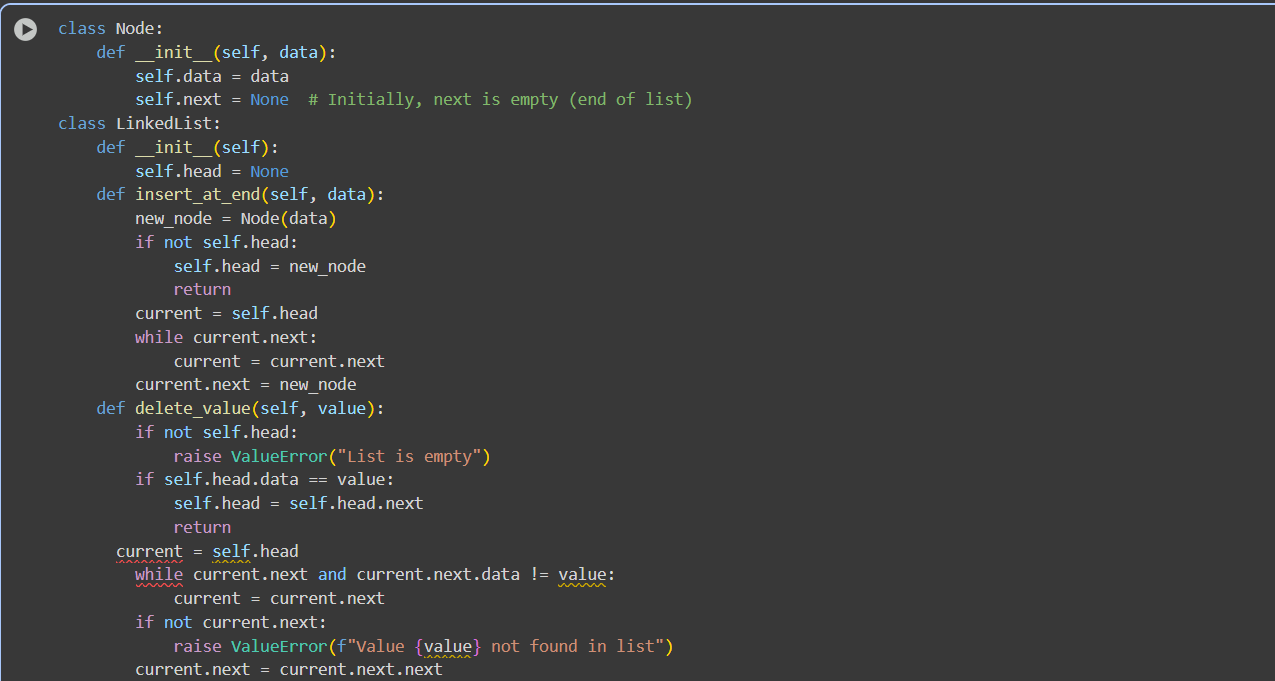
- Inline comments explaining pointer updates in insert and delete

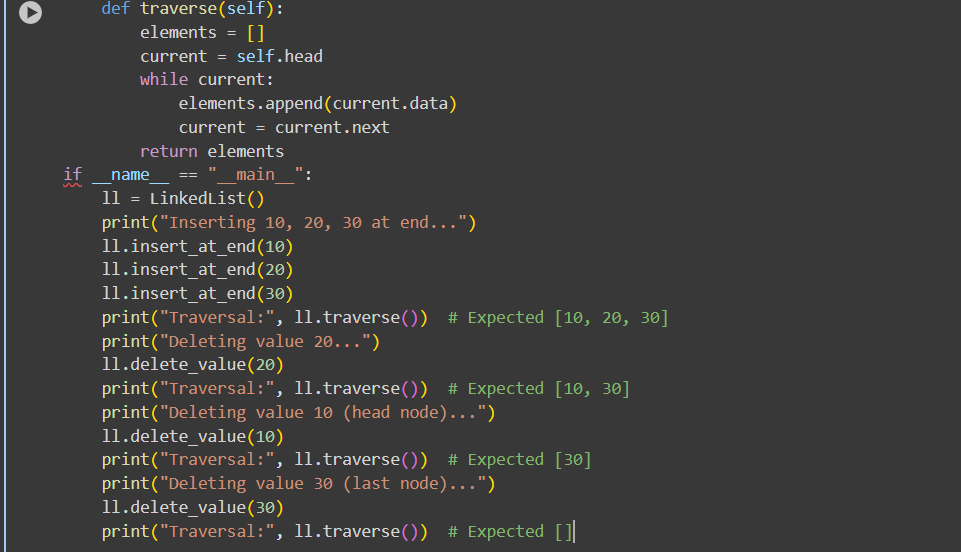
- Google-style docstrings

- A simple test example with sample data

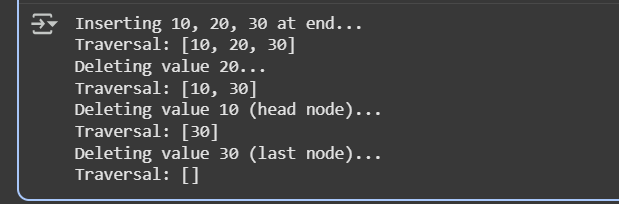
- Suggested test cases to validate all operations

Code:





Output:



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:

You are given partially written classes for a Binary Search Tree:

class Node:

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

self.key = key

self.left = None

self.right = None

class BST:

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

self.root = None

Complete the BST class by implementing:

- insert(key)

- search(key)

- inorder\_traversal()

Include:

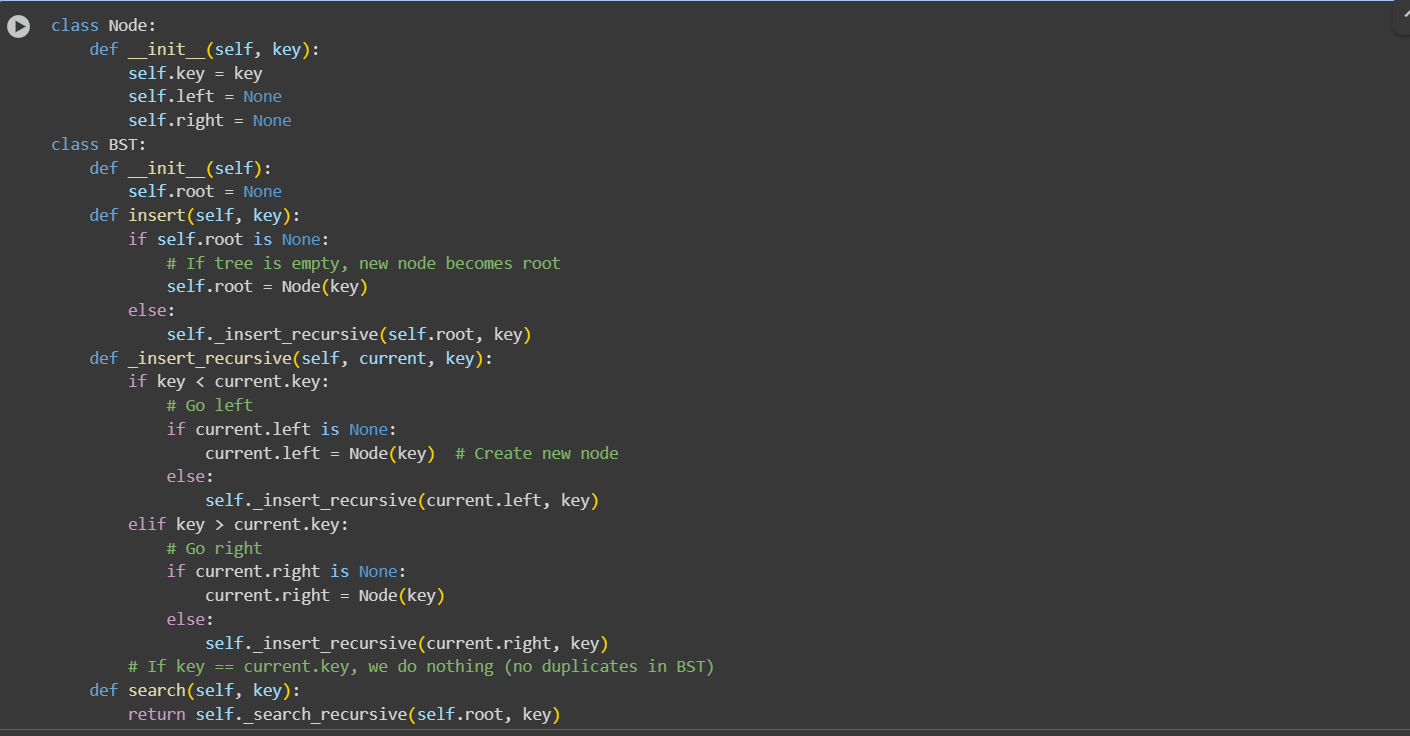
- Google-style docstrings for all methods

- Inline comments explaining pointer updates

- A test script that inserts a list of integers, prints inorder traversal,

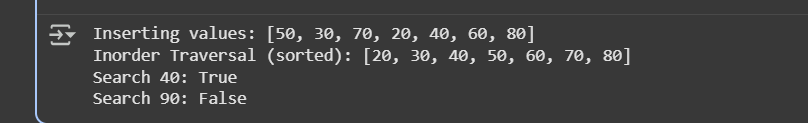
and tests search() for a present and absent element

Code:





Output:



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 in Python using an adjacency list (dictionary).

Add methods:

- add\_edge(u, v) # undirected graph

- bfs(start)

- dfs\_recursive(start)

- dfs\_iterative(start)

Include:

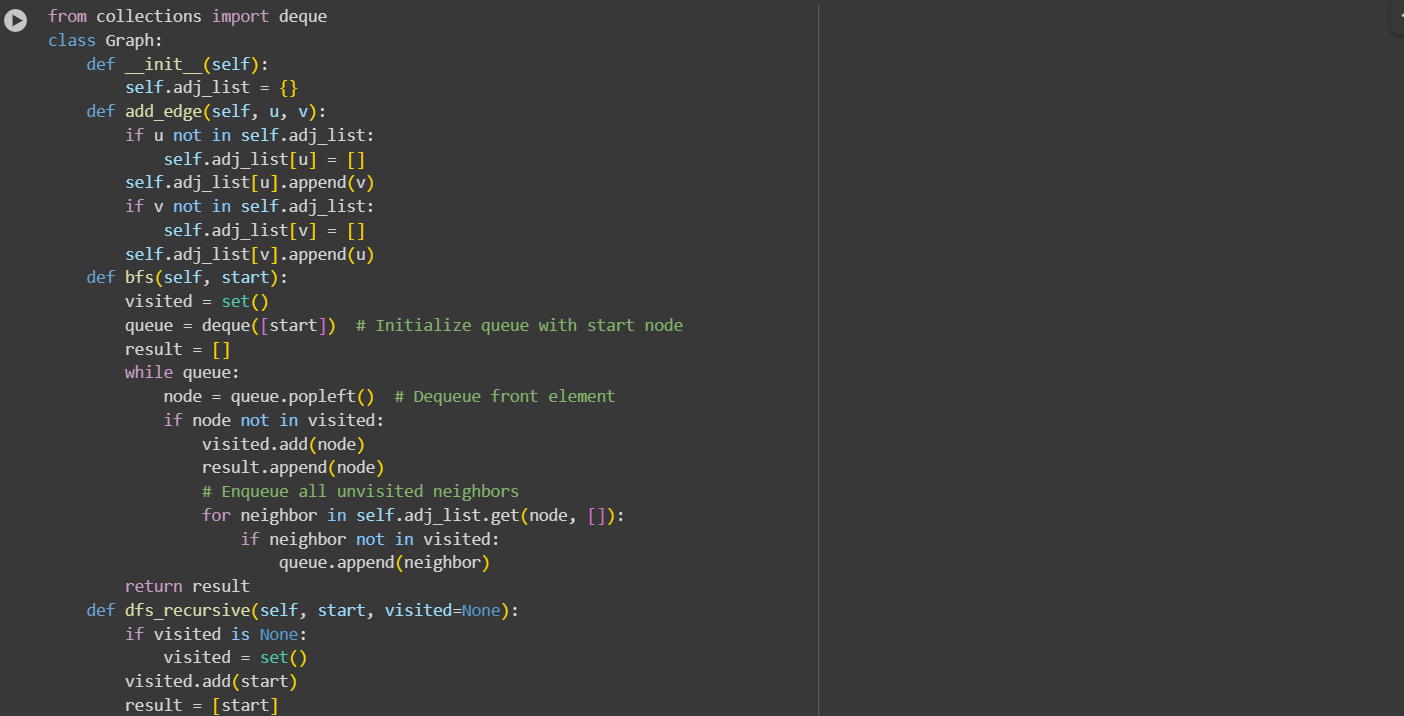
- Inline comments explaining traversal steps

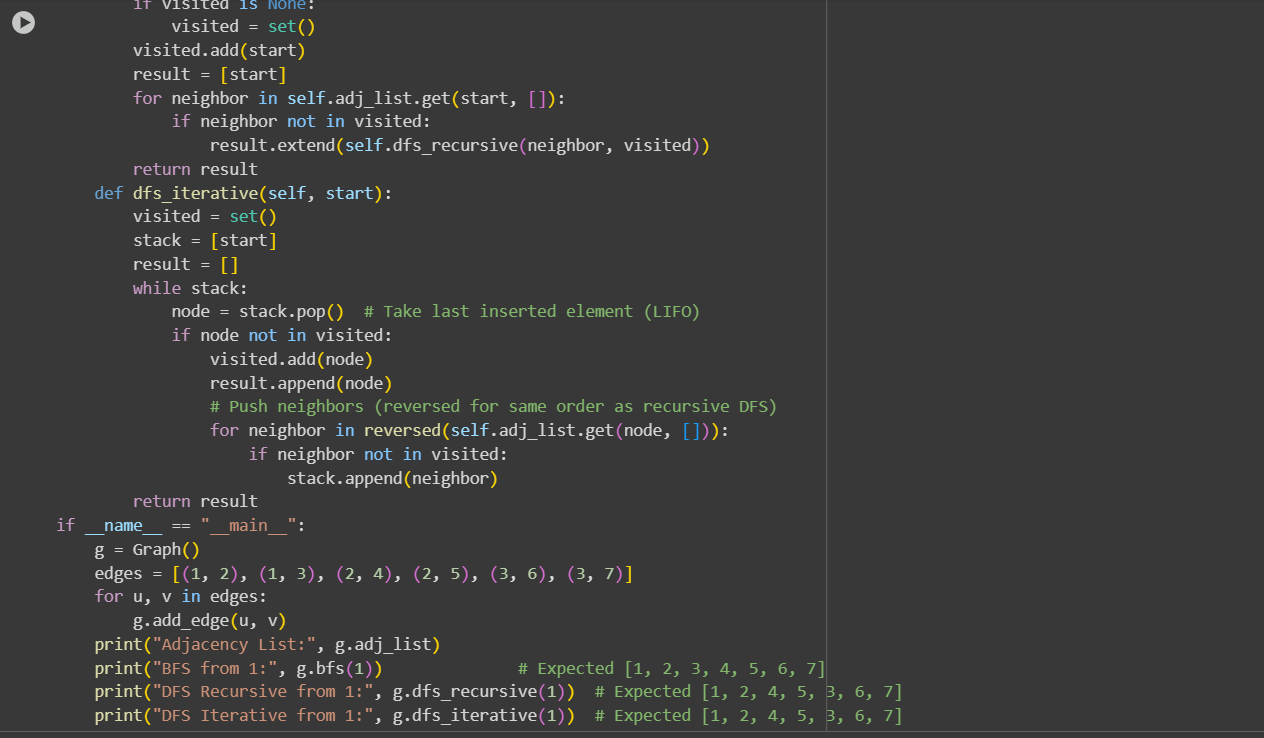
- Google-style docstrings

- A test script with a sample graph

- A note comparing recursive vs iterative DFS

Code:





Output:

