AI ASSISTED CODING

Assignment

Roll no: 2503A51L41

Name: G Hasini

Batch: 24BTCAICSB20

LAB-11: Data Structures with AI:

Implementing Fundamental Structures

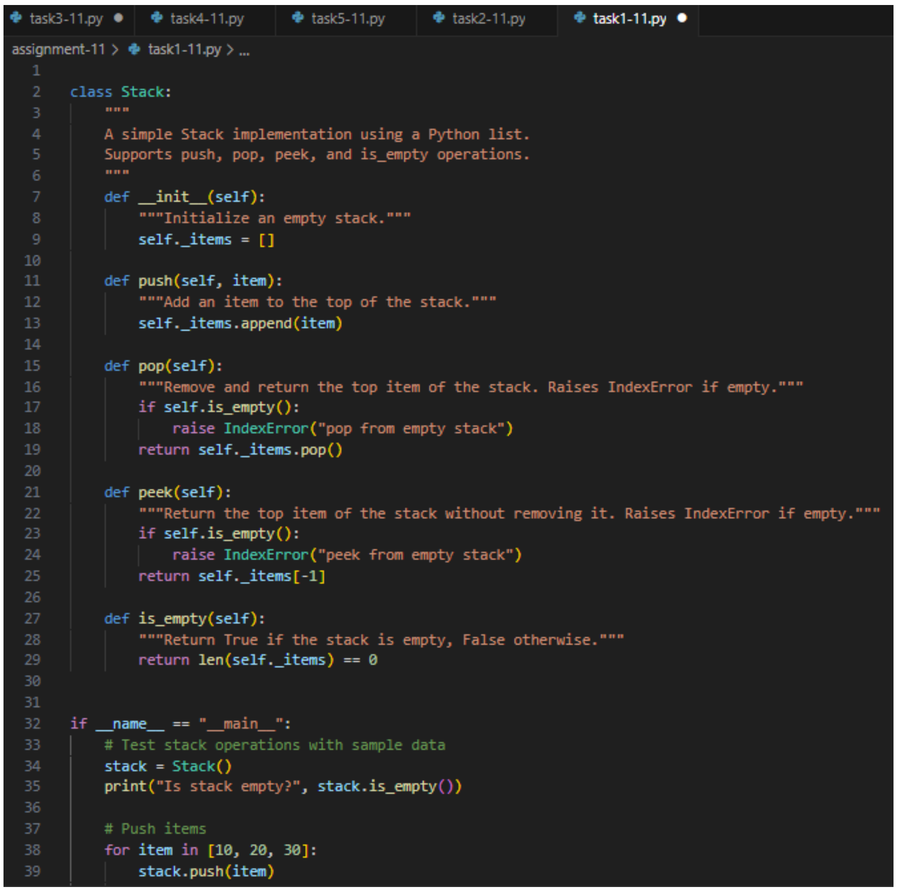
Task #1: Implementing a Stack (LIFO)

Prompt: Implement a Stack class in Python with the following operations: push(), pop(), peek(), and is_empty().

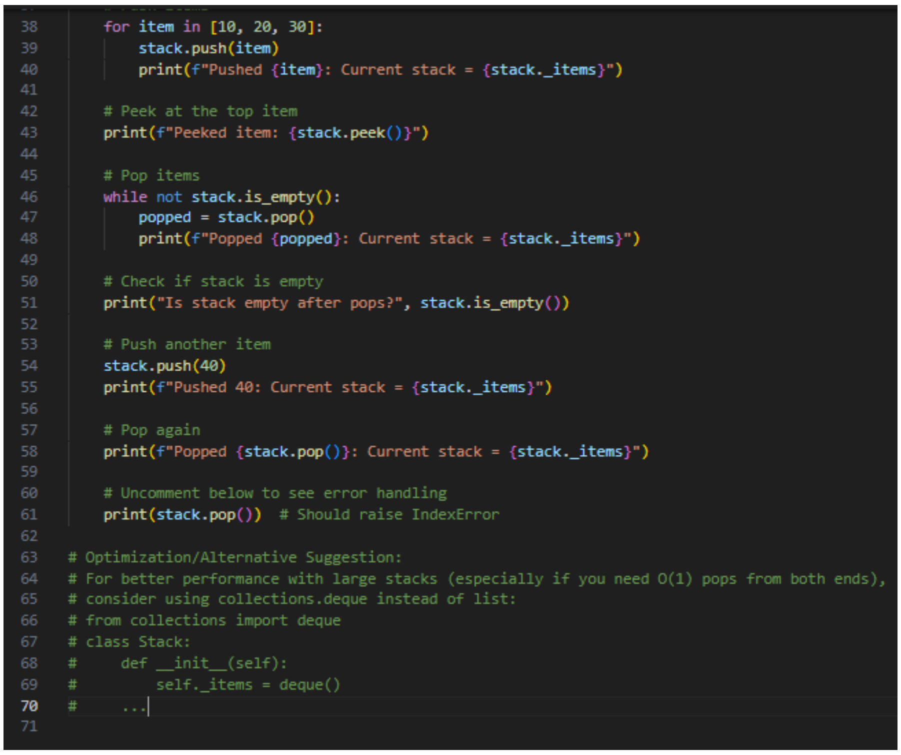
Instructions:

- Generate code skeleton with docstrings.
- Test stack operations using sample data.
- Suggest optimizations or alternative implementations (e.g., using collections.deque).
- Modify the output so that pushed items, pop items, peek item and is empty functions are also displayed.

Code generated:

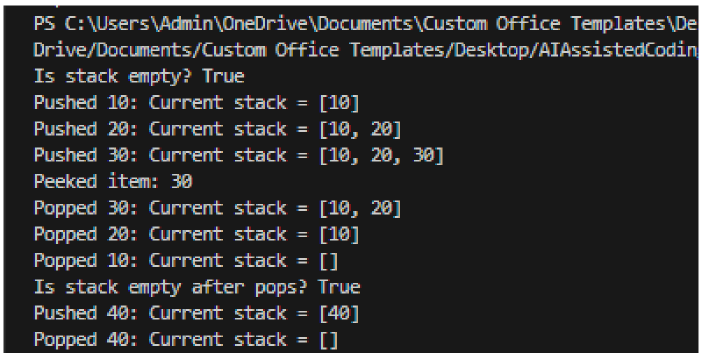


```python

class Stack:
    """
    A simple Stack implementation using a Python list.
    Supports push, pop, peek, and is_empty operations.
    """
    def __init__(self):
        """Initialize an empty stack."""
        self._items = []

    def push(self, item):
        """Add an item to the top of the stack."""
        self._items.append(item)

    def pop(self):
        """Remove and return the top item of the stack. Raises IndexError if empty."""
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """Return the top item of the stack without removing it. Raises IndexError if empty."""
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]

    def is_empty(self):
        """Return True if the stack is empty, False otherwise."""
        return len(self._items) == 0


if __name__ == "__main__":
    # Test stack operations with sample data
    stack = Stack()
    print("Is stack empty?", stack.is_empty())

    # Push items
    for item in [10, 20, 30]:
        stack.push(item)
```



```python
    for item in [10, 20, 30]:
        stack.push(item)
        print(f"Pushed {item}: Current stack = {stack._items}")

    # Peek at the top item
    print(f"Peeked item: {stack.peek()}")

    # Pop items
    while not stack.is_empty():
        popped = stack.pop()
        print(f"Popped {popped}: Current stack = {stack._items}")

    # Check if stack is empty
    print("Is stack empty after pops?", stack.is_empty())

    # Push another item
    stack.push(40)
    print(f"Pushed 40: Current stack = {stack._items}")

    # Pop again
    print(f"Popped {stack.pop()}: Current stack = {stack._items}")

    # Uncomment below to see error handling
    print(stack.pop())  # Should raise IndexError

# Optimization/Alternative Suggestion:
# For better performance with large stacks (especially if you need O(1) pops from both ends),
# consider using collections.deque instead of list:
# from collections import deque
# class Stack:
#     def __init__(self):
#         self._items = deque()
#     ...

```

Output:



```
PS C:\Users\Admin\OneDrive\Documents\Custom Office Templates\De
Drive/Documents/Custom Office Templates/Desktop/AIAssistedCodin
Is stack empty? True
Pushed 10: Current stack = [10]
Pushed 20: Current stack = [10, 20]
Pushed 30: Current stack = [10, 20, 30]
Peeked item: 30
Popped 30: Current stack = [10, 20]
Popped 20: Current stack = [10]
Popped 10: Current stack = []
Is stack empty after pops? True
Pushed 40: Current stack = [40]
Popped 40: Current stack = []
```

Observation:

The stack implementation correctly demonstrated push, pop, peek, and is_empty operations.

The output validated LIFO behavior, where the last inserted element was removed first.

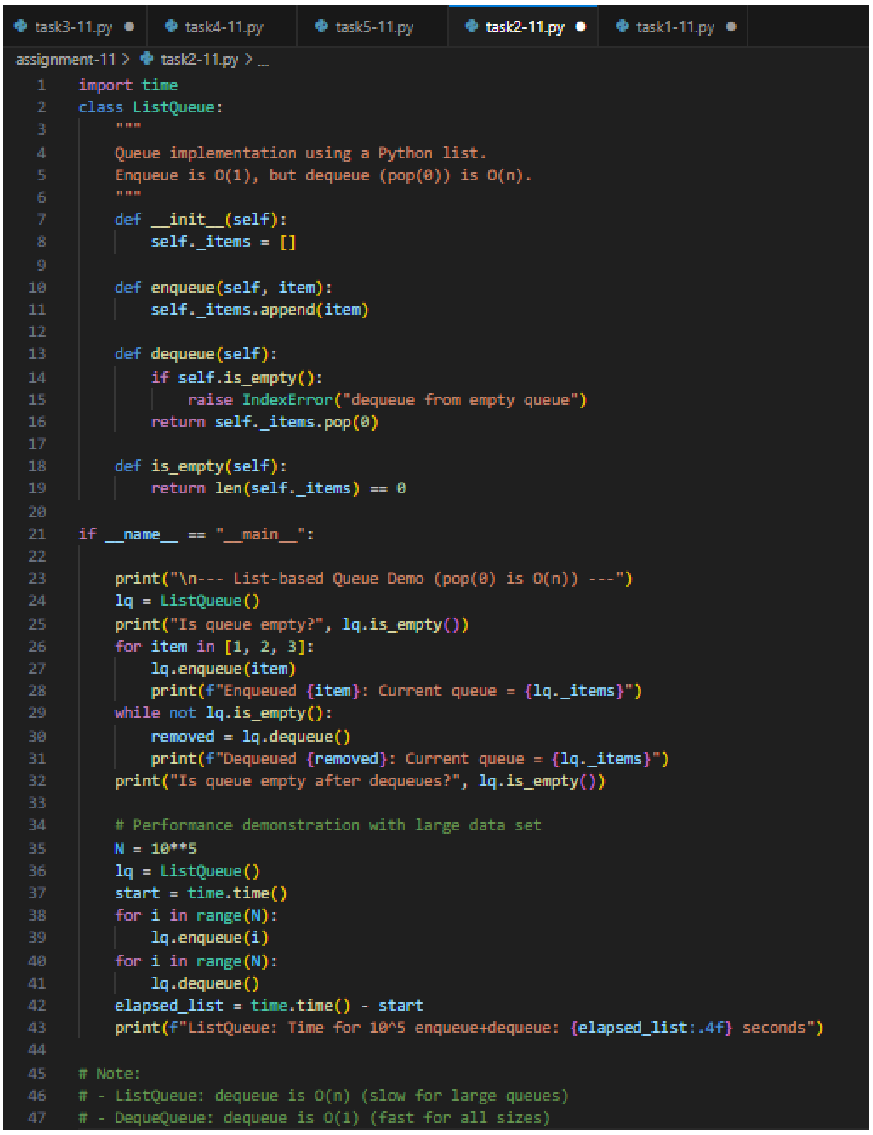
The AI-assisted skeleton made the logic clear and structured.

This shows how AI can speed up coding while ensuring correctness.

Task#2: Queue Implementation with Performance Review

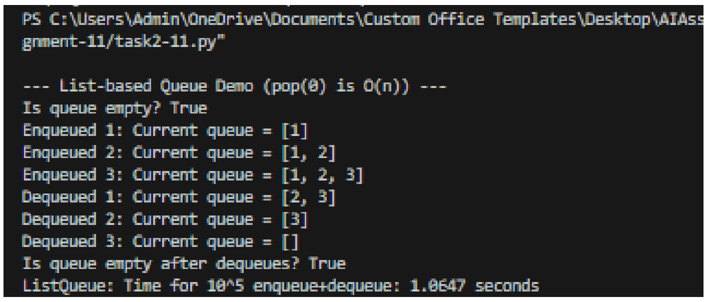
Prompt: Implement a Queue with enqueue(), dequeue(), and is_empty() methods. First, implement using Python lists.

Code Generated:



```python
import time
class ListQueue:
    """
    Queue implementation using a Python list.
    Enqueue is O(1), but dequeue (pop(0)) is O(n).
    """
    def __init__(self):
        self._items = []

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

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

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

if __name__ == "__main__":

    print("\n--- List-based Queue Demo (pop(0) is O(n)) ---")
    lq = ListQueue()
    print("Is queue empty?", lq.is_empty())
    for item in [1, 2, 3]:
        lq.enqueue(item)
        print(f"Enqueued {item}: Current queue = {lq._items}")
    while not lq.is_empty():
        removed = lq.dequeue()
        print(f"Dequeued {removed}: Current queue = {lq._items}")
    print("Is queue empty after dequeues?", lq.is_empty())

    # Performance demonstration with large data set
    N = 10**5
    lq = ListQueue()
    start = time.time()
    for i in range(N):
        lq.enqueue(i)
    for i in range(N):
        lq.dequeue()
    elapsed_list = time.time() - start
    print(f"ListQueue: Time for 10^5 enqueue+dequeue: {elapsed_list:.4f} seconds")

# Note:
# - ListQueue: dequeue is O(n) (slow for large queues)
# - DequeQueue: dequeue is O(1) (fast for all sizes)
```

Output:



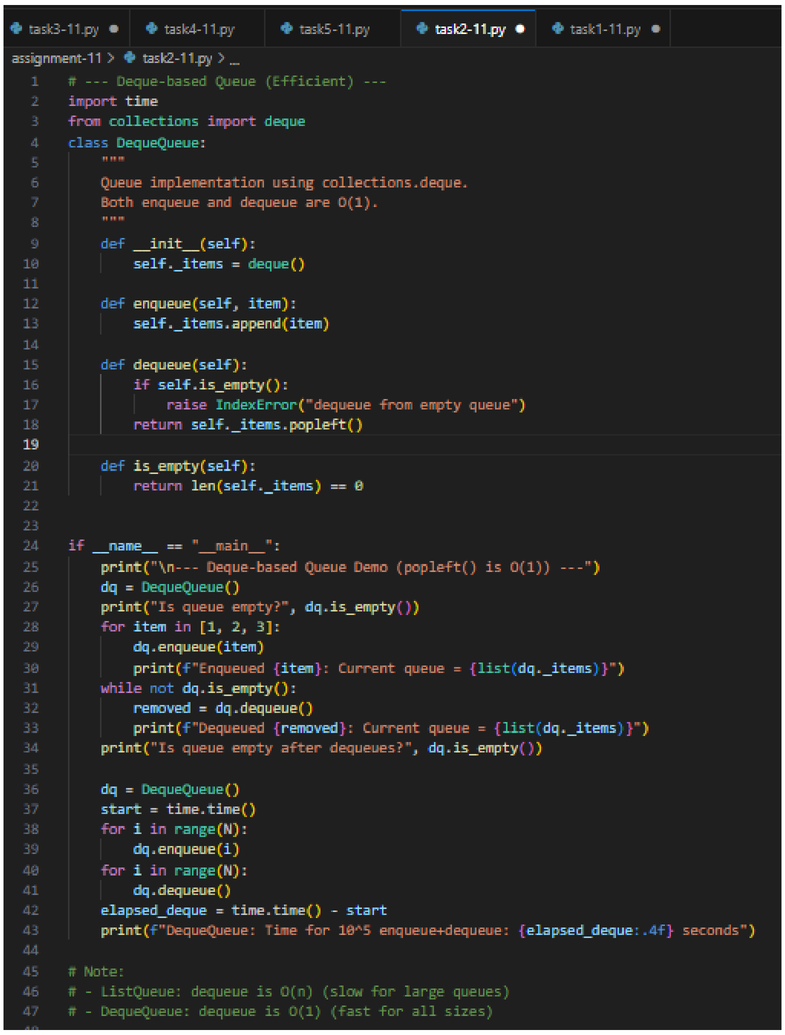
```
PS C:\Users\Admin\OneDrive\Documents\Custom Office Templates\Desktop\AIAss
gnment-11/task2-11.py"

--- List-based Queue Demo (pop(0) is O(n)) ---
Is queue empty? True
Enqueued 1: Current queue = [1]
Enqueued 2: Current queue = [1, 2]
Enqueued 3: Current queue = [1, 2, 3]
Dequeued 1: Current queue = [2, 3]
Dequeued 2: Current queue = [3]
Dequeued 3: Current queue = []
Is queue empty after dequeues? True
ListQueue: Time for 10^5 enqueue+dequeue: 1.0647 seconds
```

Observation:

The queue implemented using lists worked but showed performance limits during repeated dequeues. The optimized version using collections.deque improved efficiency, especially for front removals. Output confirmed proper FIFO behavior, highlighting AI's role in suggesting better data structures.

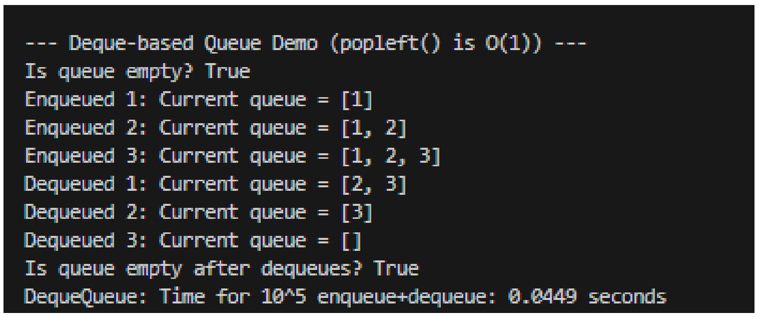
Prompt-2: Review performance and suggest a more efficient implementation (using collections.deque).Perform any operation in this code so that it shows that it is different as compared to the list.

Code Generated-2:



```python
# --- Deque-based Queue (Efficient) ---
import time
from collections import deque
class DequeQueue:
    """
    Queue implementation using collections.deque.
    Both enqueue and dequeue are O(1).
    """
    def __init__(self):
        self._items = deque()

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

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

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


if __name__ == "__main__":
    print("\n--- Deque-based Queue Demo (popleft() is O(1)) ---")
    dq = DequeQueue()
    print("Is queue empty?", dq.is_empty())
    for item in [1, 2, 3]:
        dq.enqueue(item)
        print(f"Enqueued {item}: Current queue = {list(dq._items)}")
    while not dq.is_empty():
        removed = dq.dequeue()
        print(f"Dequeued {removed}: Current queue = {list(dq._items)}")
    print("Is queue empty after dequeues?", dq.is_empty())

    dq = DequeQueue()
    start = time.time()
    for i in range(N):
        dq.enqueue(i)
    for i in range(N):
        dq.dequeue()
    elapsed_deque = time.time() - start
    print(f"DequeQueue: Time for 10^5 enqueue+dequeue: {elapsed_deque:.4f} seconds")

# Note:
# - ListQueue: dequeue is O(n) (slow for large queues)
# - DequeQueue: dequeue is O(1) (fast for all sizes)
```

Output:



```
--- Deque-based Queue Demo (popleft() is O(1)) ---
Is queue empty? True
Enqueued 1: Current queue = [1]
Enqueued 2: Current queue = [1, 2]
Enqueued 3: Current queue = [1, 2, 3]
Dequeued 1: Current queue = [2, 3]
Dequeued 2: Current queue = [3]
Dequeued 3: Current queue = []
Is queue empty after dequeues? True
DequeQueue: Time for 10^5 enqueue+dequeue: 0.0449 seconds
```

Observation: The queue implemented using lists worked but showed performance limits during repeated dequeues. The optimized version using collections.deque improved efficiency, especially for front removals. Output confirmed proper FIFO behavior, highlighting AI's role in suggesting better data structures.

Task#3: Singly Linked List with Traversal

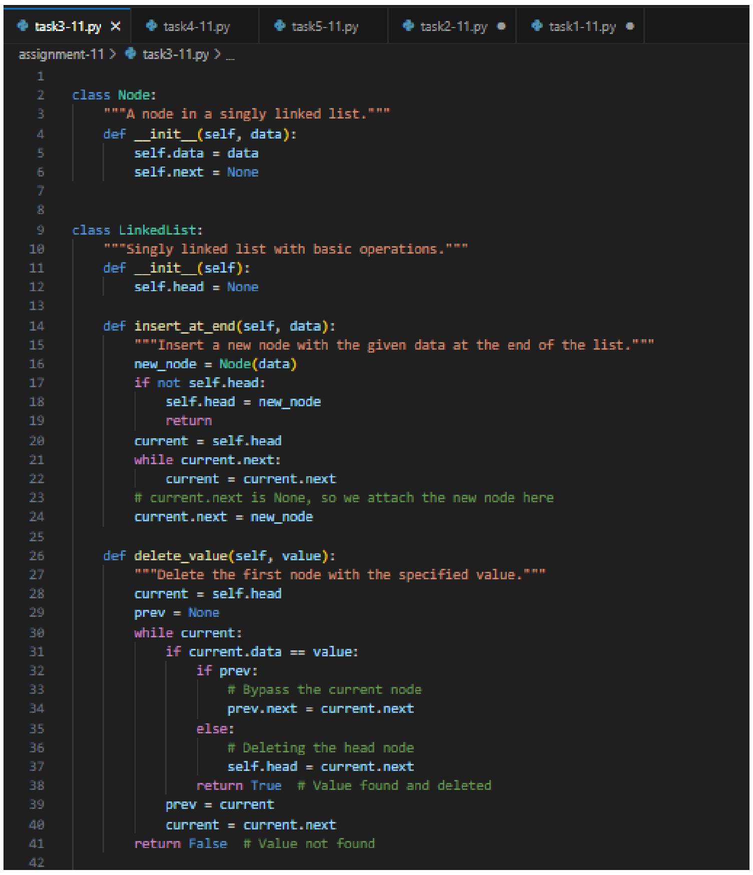
operations: insert_at_end(), delete_value(), and traverse().

Prompt: Implement a Singly Linked List with operations: insert_at_end(), delete_value(), and traverse().

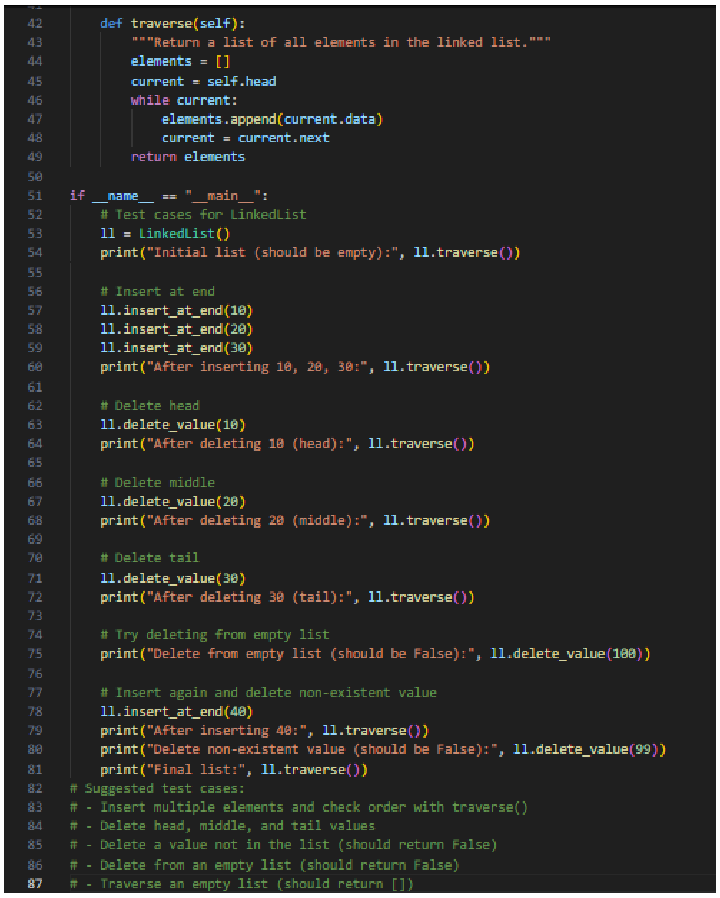
Instructions:

- Start with a simple class-based implementation (Node, LinkedList).
- Generate inline comments explaining pointer updates (which are non-trivial). Suggest test cases to validate all operations.

Code generated:

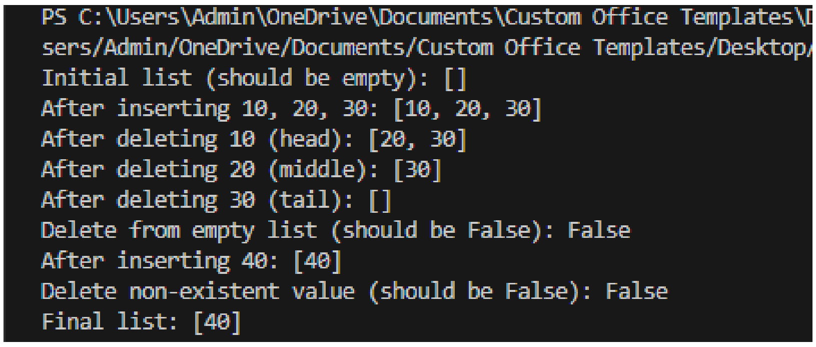


```python

class Node:
    """A node in a singly linked list."""
    def __init__(self, data):
        self.data = data
        self.next = None


class LinkedList:
    """Singly linked list with basic operations."""
    def __init__(self):
        self.head = None

    def insert_at_end(self, data):
        """Insert a new node with the given data at the end of the list."""
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        # current.next is None, so we attach the new node here
        current.next = new_node

    def delete_value(self, value):
        """Delete the first node with the specified value."""
        current = self.head
        prev = None
        while current:
            if current.data == value:
                if prev:
                    # Bypass the current node
                    prev.next = current.next
                else:
                    # Deleting the head node
                    self.head = current.next
                return True  # Value found and deleted
            prev = current
            current = current.next
        return False  # Value not found

```



```python
    def traverse(self):
        """Return a list of all elements in the linked list."""
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        return elements

if __name__ == "__main__":
    # Test cases for LinkedList
    ll = LinkedList()
    print("Initial list (should be empty):", ll.traverse())

    # Insert at end
    ll.insert_at_end(10)
    ll.insert_at_end(20)
    ll.insert_at_end(30)
    print("After inserting 10, 20, 30:", ll.traverse())

    # Delete head
    ll.delete_value(10)
    print("After deleting 10 (head):", ll.traverse())

    # Delete middle
    ll.delete_value(20)
    print("After deleting 20 (middle):", ll.traverse())

    # Delete tail
    ll.delete_value(30)
    print("After deleting 30 (tail):", ll.traverse())

    # Try deleting from empty list
    print("Delete from empty list (should be False):", ll.delete_value(100))

    # Insert again and delete non-existent value
    ll.insert_at_end(40)
    print("After inserting 40:", ll.traverse())
    print("Delete non-existent value (should be False):", ll.delete_value(99))
    print("Final list:", ll.traverse())
# Suggested test cases:
# - Insert multiple elements and check order with traverse()
# - Delete head, middle, and tail values
# - Delete a value not in the list (should return False)
# - Delete from an empty list (should return False)
# - Traverse an empty list (should return [])
```

Output:



```
PS C:\Users\Admin\OneDrive\Documents\Custom Office Templates\D
sers/Admin/OneDrive/Documents/Custom Office Templates/Desktop/
Initial list (should be empty): []
After inserting 10, 20, 30: [10, 20, 30]
After deleting 10 (head): [20, 30]
After deleting 20 (middle): [30]
After deleting 30 (tail): []
Delete from empty list (should be False): False
After inserting 40: [40]
Delete non-existent value (should be False): False
Final list: [40]
```

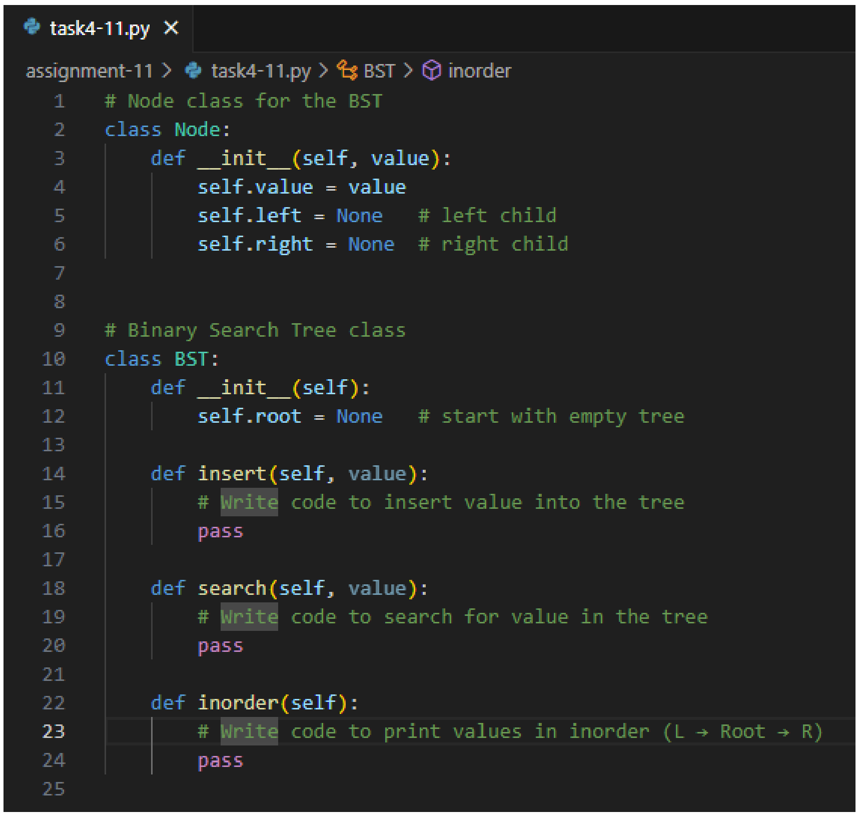
Observation:

The linked list supported insertion, deletion, and traversal operations successfully. Pointer updates were explained clearly with inline AI-generated comments, aiding conceptual understanding. Test cases confirmed robustness against different scenarios like deleting head or absent values.

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.

Code given to AI:

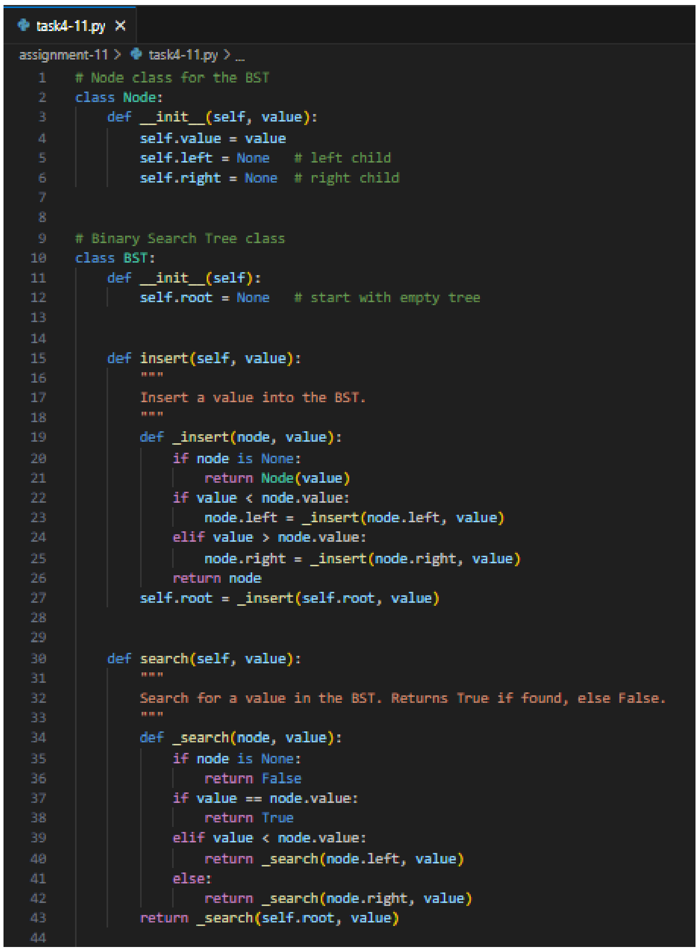


task4-11.py ×

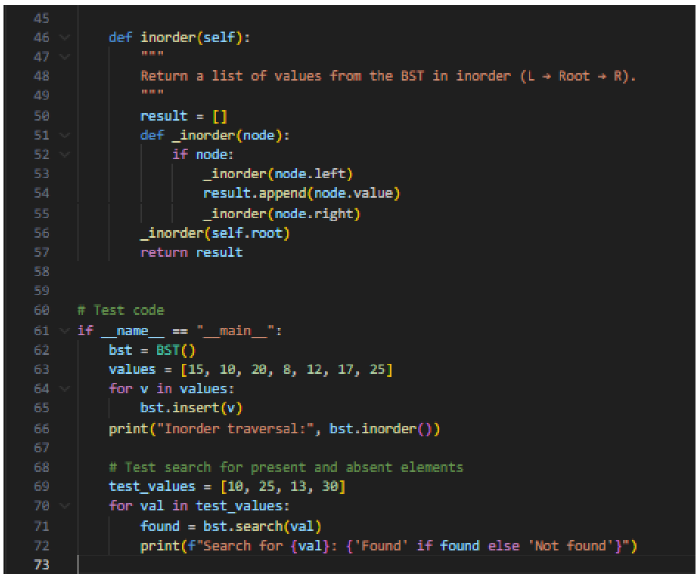
assignment-11 > task4-11.py > BST > inorder

```python
# Node class for the BST
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None   # left child
        self.right = None  # right child


# Binary Search Tree class
class BST:
    def __init__(self):
        self.root = None   # start with empty tree

    def insert(self, value):
        # Write code to insert value into the tree
        pass

    def search(self, value):
        # Write code to search for value in the tree
        pass

    def inorder(self):
        # Write code to print values in inorder (L → Root → R)
        pass

```

Prompt: Complete missing methods and add docstrings.Test with a list of integers and compare outputs of search() for present vs absent elements.

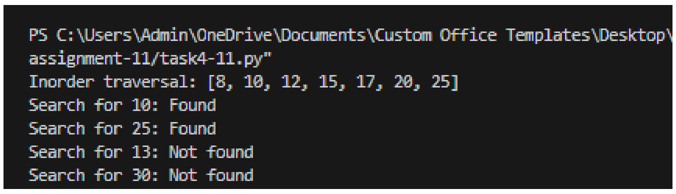


```python
# Node class for the BST
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None   # left child
        self.right = None  # right child


# Binary Search Tree class
class BST:
    def __init__(self):
        self.root = None   # start with empty tree


    def insert(self, value):
        """
        Insert a value into the BST.
        """
        def _insert(node, value):
            if node is None:
                return Node(value)
            if value < node.value:
                node.left = _insert(node.left, value)
            elif value > node.value:
                node.right = _insert(node.right, value)
            return node
        self.root = _insert(self.root, value)


    def search(self, value):
        """
        Search for a value in the BST. Returns True if found, else False.
        """
        def _search(node, value):
            if node is None:
                return False
            if value == node.value:
                return True
            elif value < node.value:
                return _search(node.left, value)
            else:
                return _search(node.right, value)
        return _search(self.root, value)

```



```python

    def inorder(self):
        """
        Return a list of values from the BST in inorder (L → Root → R).
        """
        result = []
        def _inorder(node):
            if node:
                _inorder(node.left)
                result.append(node.value)
                _inorder(node.right)
        _inorder(self.root)
        return result


# Test code
if __name__ == "__main__":
    bst = BST()
    values = [15, 10, 20, 8, 12, 17, 25]
    for v in values:
        bst.insert(v)
    print("Inorder traversal:", bst.inorder())

    # Test search for present and absent elements
    test_values = [10, 25, 13, 30]
    for val in test_values:
        found = bst.search(val)
        print(f"Search for {val}: {'Found' if found else 'Not found'}")

```

Output:



```
PS C:\Users\Admin\OneDrive\Documents\Custom Office Templates\Desktop\
assignment-11/task4-11.py"
Inorder traversal: [8, 10, 12, 15, 17, 20, 25]
Search for 10: Found
Search for 25: Found
Search for 13: Not found
Search for 30: Not found
```

Observation:

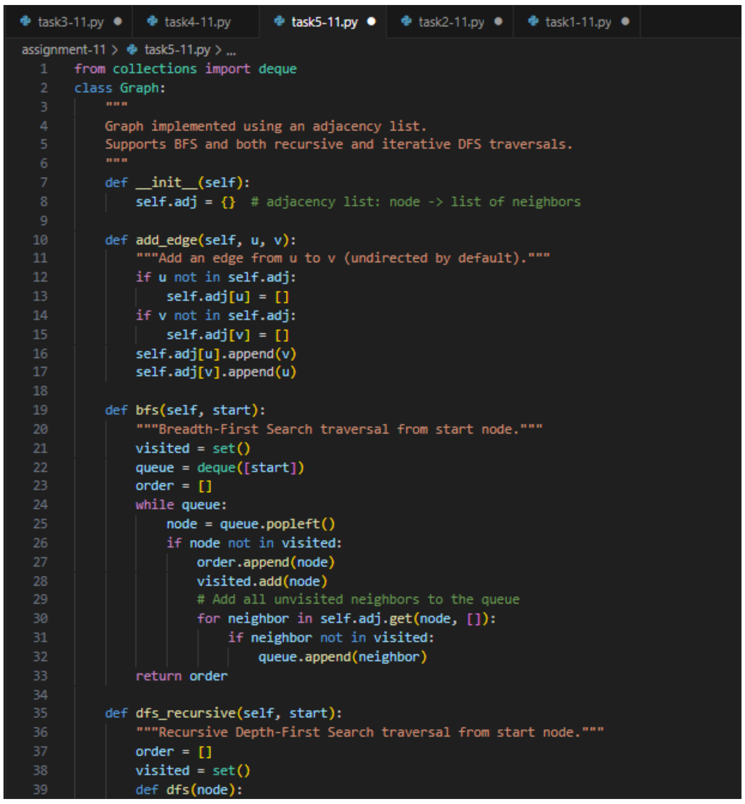
The BST allowed insertion, searching, and inorder traversal, producing sorted output as expected. Searches returned correct results for both present and absent values. The AI-generated partial class guided step-by-step completion, reinforcing understanding of recursive structures.

Task#5: Graph Representation and BFS/DFS Traversal

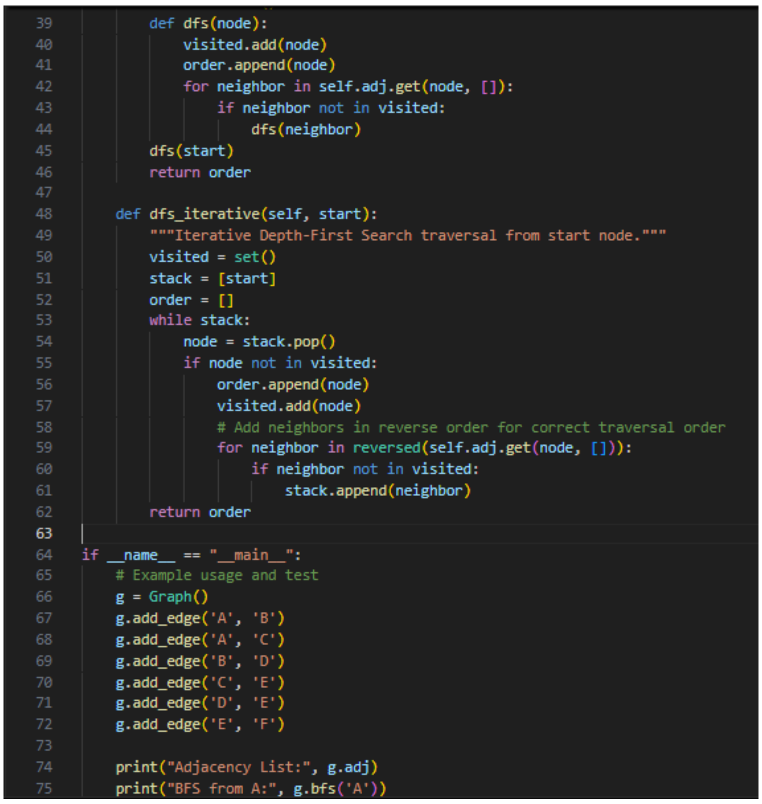
Prompt: Implement a Graph using an adjacency list, with traversal methods BFS() and DFS().

- Instructions:
  - Start with an adjacency list dictionary.
  - Generate BFS and DFS implementations with inline comments.
  - Compare recursive vs iterative DFS if suggested by AI.

Output:

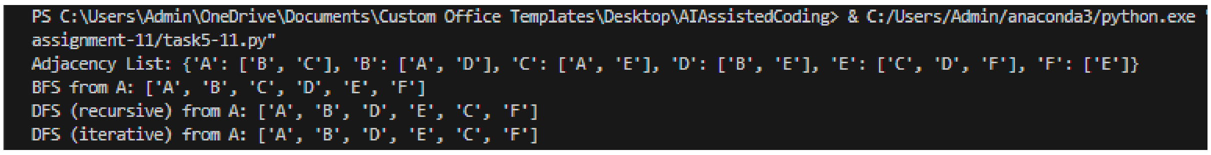


```python
from collections import deque
class Graph:
    """
    Graph implemented using an adjacency list.
    Supports BFS and both recursive and iterative DFS traversals.
    """
    def __init__(self):
        self.adj = {}  # adjacency list: node -> list of neighbors

    def add_edge(self, u, v):
        """Add an edge from u to v (undirected by default)."""
        if u not in self.adj:
            self.adj[u] = []
        if v not in self.adj:
            self.adj[v] = []
        self.adj[u].append(v)
        self.adj[v].append(u)

    def bfs(self, start):
        """Breadth-First Search traversal from start node."""
        visited = set()
        queue = deque([start])
        order = []
        while queue:
            node = queue.popleft()
            if node not in visited:
                order.append(node)
                visited.add(node)
                # Add all unvisited neighbors to the queue
                for neighbor in self.adj.get(node, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
        return order

    def dfs_recursive(self, start):
        """Recursive Depth-First Search traversal from start node."""
        order = []
        visited = set()
        def dfs(node):
```



```python
        def dfs(node):
            visited.add(node)
            order.append(node)
            for neighbor in self.adj.get(node, []):
                if neighbor not in visited:
                    dfs(neighbor)
        dfs(start)
        return order

    def dfs_iterative(self, start):
        """Iterative Depth-First Search traversal from start node."""
        visited = set()
        stack = [start]
        order = []
        while stack:
            node = stack.pop()
            if node not in visited:
                order.append(node)
                visited.add(node)
                # Add neighbors in reverse order for correct traversal order
                for neighbor in reversed(self.adj.get(node, [])):
                    if neighbor not in visited:
                        stack.append(neighbor)
        return order

if __name__ == "__main__":
    # Example usage and test
    g = Graph()
    g.add_edge('A', 'B')
    g.add_edge('A', 'C')
    g.add_edge('B', 'D')
    g.add_edge('C', 'E')
    g.add_edge('D', 'E')
    g.add_edge('E', 'F')

    print("Adjacency List:", g.adj)
    print("BFS from A:", g.bfs('A'))
```

Output:



```
PS C:\Users\Admin\OneDrive\Documents\Custom Office Templates\Desktop\AIAssistedCoding> & C:/Users/Admin/anaconda3/python.exe
assignment-11/task5-11.py"
Adjacency List: {'A': ['B', 'C'], 'B': ['A', 'D'], 'C': ['A', 'E'], 'D': ['B', 'E'], 'E': ['C', 'D', 'F'], 'F': ['E']}
BFS from A: ['A', 'B', 'C', 'D', 'E', 'F']
DFS (recursive) from A: ['A', 'B', 'D', 'E', 'C', 'F']
DFS (iterative) from A: ['A', 'B', 'D', 'E', 'C', 'F']
```

Observation:

Graph traversal using BFS and DFS was implemented with adjacency lists. The outputs validated correct order of node visits for both search strategies. Recursive vs iterative DFS approaches were highlighted, showing AI's role in presenting alternative solutions.