Assignment-11.4

G.Sai teja

2303A52135

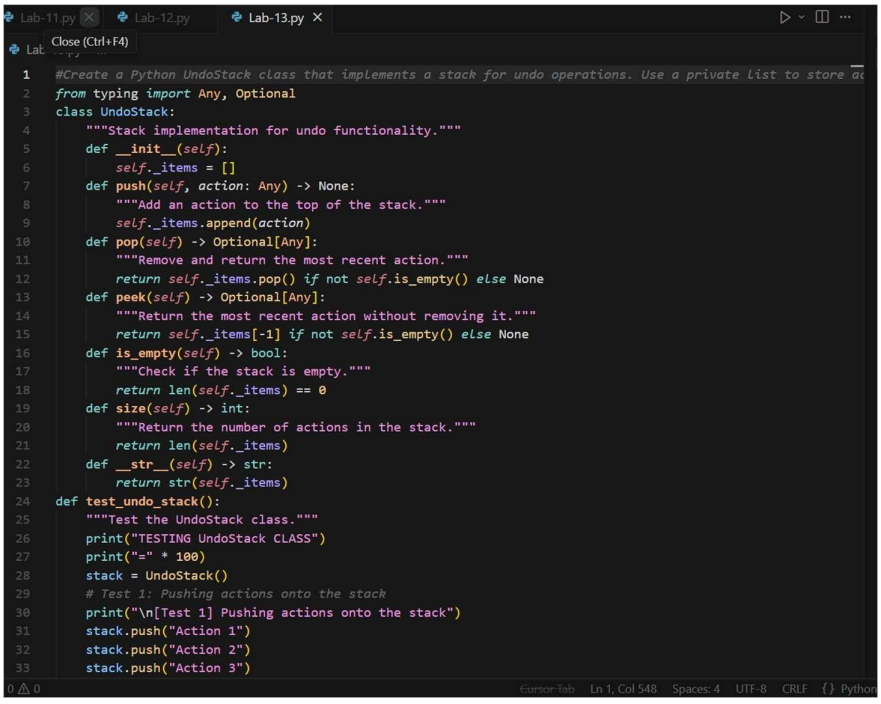
Batch 41

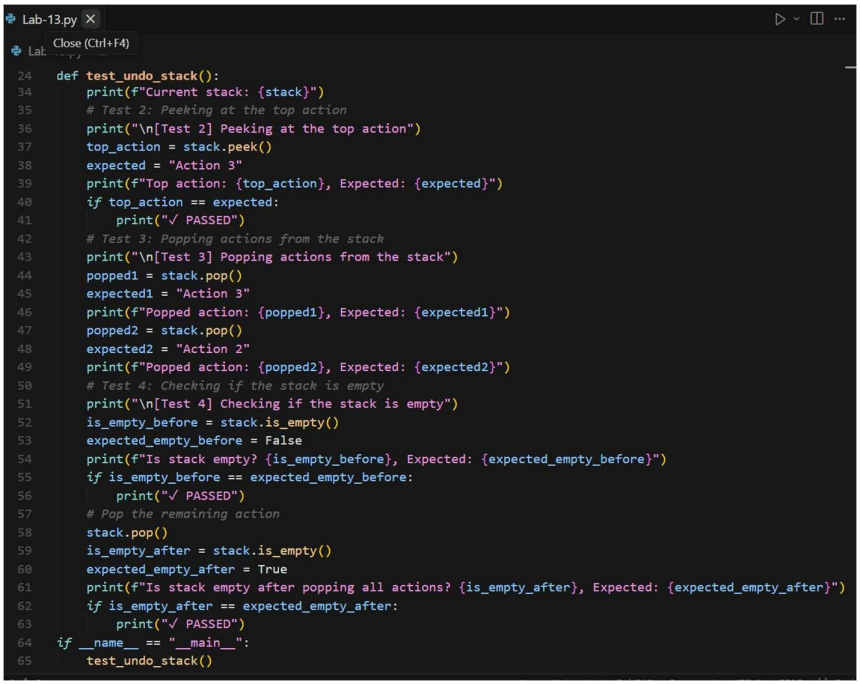
Task-1: Stack Implementation for Undo Operations (LIFO)

PROMPT:

Create a Python UndoStack class that implements a stack for undo operations. Use a private list to store actions. Include push() to add actions, pop() to remove and return the most recent (return None if empty), peek() to view without removing (return None if empty), is_empty() to check emptiness, size() to return the count, and __str__() for string representation. Use type hints (typing.Any, typing.Optional). Add a test function that exercises all methods, including edge cases like empty stacks. Handle empty stack cases without exceptions.

CODE:





OUTPUT:

```
TESTING UndoStack CLASS
================================================================================
==================

[Test 1] Pushing actions onto the stack
Current stack: ['Action 1', 'Action 2', 'Action 3']

[Test 2] Peeking at the top action
Top action: Action 3, Expected: Action 3
✓ PASSED

[Test 3] Popping actions from the stack
Popped action: Action 3, Expected: Action 3
Popped action: Action 2, Expected: Action 2

[Test 4] Checking if the stack is empty
Is stack empty? False, Expected: False
✓ PASSED
Is stack empty after popping all actions? True, Expected: True
✓ PASSED
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

EXPLANATION:

The UndoStack class implements a stack using a private _items list. It provides push() to add actions, pop() to remove and return the top item (returns None if empty), peek() to view the top without removing it (returns None if empty), is_empty() to check if the stack is empty, size() to return the count, and __str__() for string representation. The class uses type hints (Any, Optional) and handles empty stacks without exceptions. The test_undo_stack() function exercises all methods: it pushes three actions, peeks at the top, pops items in LIFO order, and verifies is_empty() before and after emptying the stack. The implementation follows standard stack behavior where the last item added is the first one removed.

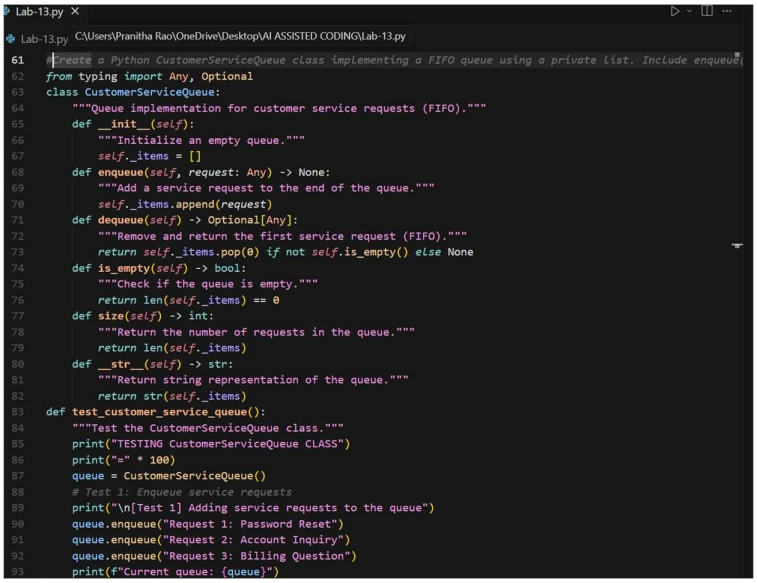
Task-2: Queue for Customer Service Requests (FIFO)

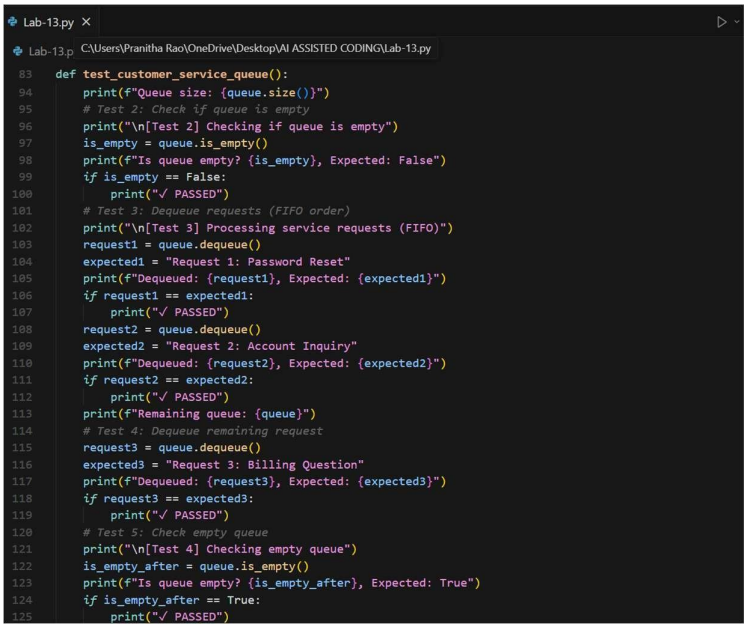
List Based Queue:

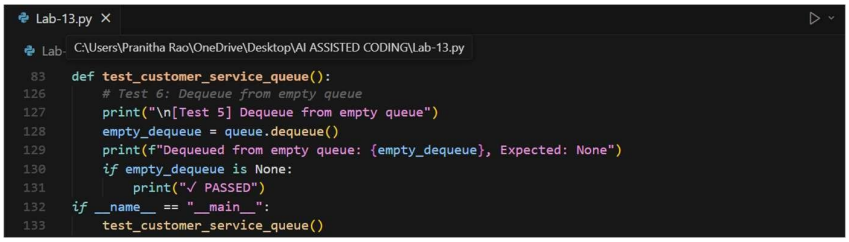
PROMPT:

Create a Python CustomerServiceQueue class implementing a FIFO queue using a private list. Include enqueue(request: Any), dequeue() -> Optional[Any] (returns None if empty), is_empty() -> bool, size() -> int, and __str__() methods. Use type hints (typing.Any, typing.Optional) and handle empty queue cases gracefully. Provide a comprehensive test function covering all methods and edge cases, including dequeuing from an empty queue.

CODE:







OUTPUT:

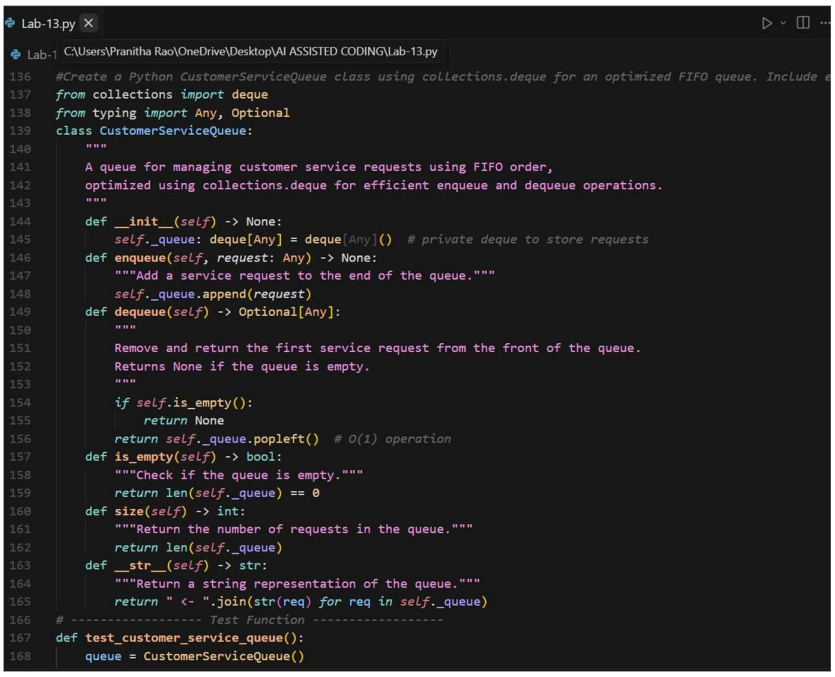
```
rograms/Python/Python314/python.exe "c:/Users/saite/Downloads/AI ASSISTENT CODING
/As11.py/task2.py"
TESTING CustomerServiceQueue CLASS
================================================================================
====================

[Test 1] Adding service requests to the queue
Current queue: ['Request 1: Password Reset', 'Request 2: Account Inquiry', 'Reque
st 3: Billing Question']
Queue size: 3

[Test 2] Checking if queue is empty
Is queue empty? False, Expected: False
✓ PASSED

[Test 3] Processing service requests (FIFO)
Dequeued: Request 1: Password Reset, Expected: Request 1: Password Reset
✓ PASSED
Dequeued: Request 2: Account Inquiry, Expected: Request 2: Account Inquiry
✓ PASSED
Remaining queue: ['Request 3: Billing Question']
Dequeued: Request 3: Billing Question, Expected: Request 3: Billing Question
✓ PASSED

[Test 4] Checking empty queue
Is queue empty? True, Expected: True
✓ PASSED
```

Deque-based optimized queue:

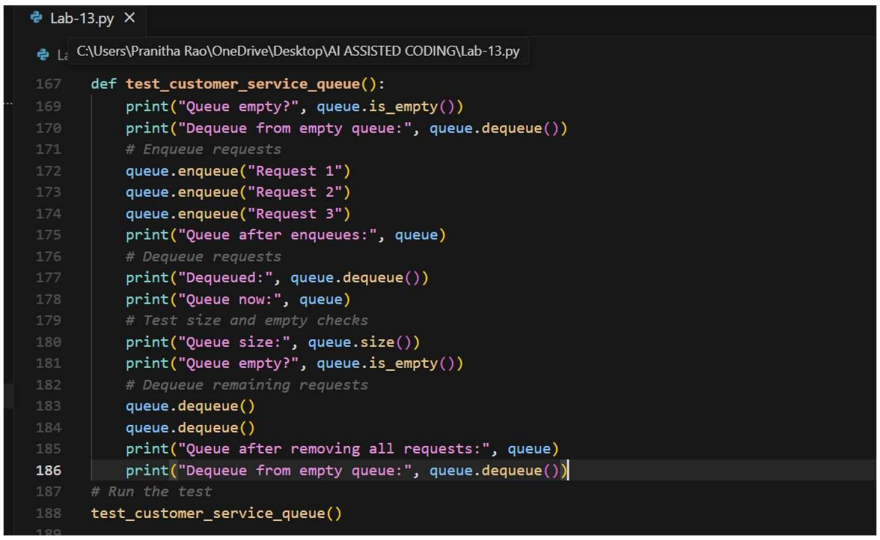
PROMPT:

Create a Python CustomerServiceQueue class using collections.deque for an optimized FIFO queue. Include enqueue(request: Any), dequeue() -> Optional[Any] (returns None if empty), is_empty() -> bool, size() -> int, and __str__() methods. Ensure all operations are efficient (O(1)) and handle empty queue cases gracefully. Also, provide a test function covering all methods and edge cases, with a brief explanation of why deque is more efficient than a list-based queue.

CODE:

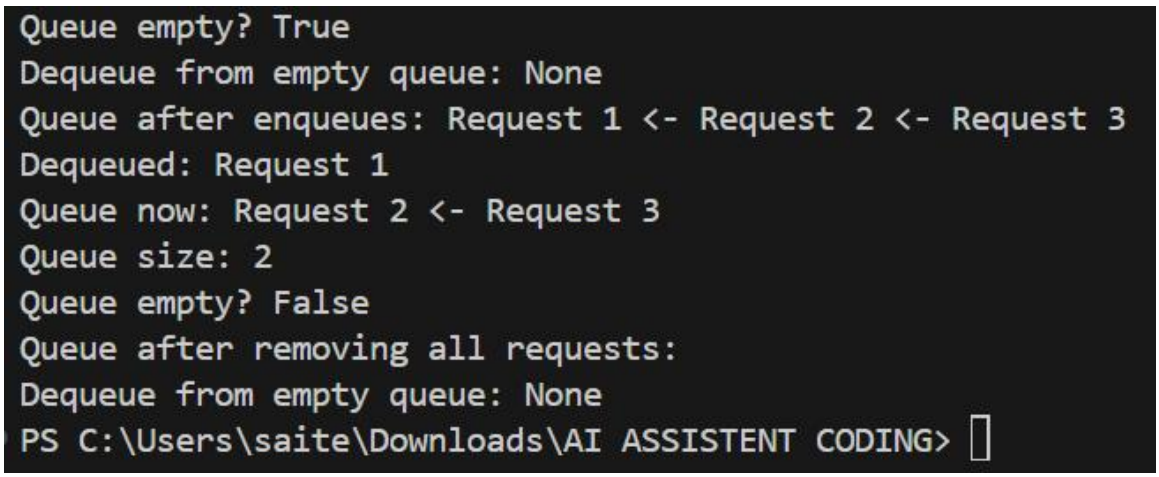


```python
#Create a Python CustomerServiceQueue class using collections.deque for an optimized FIFO queue. Include e
from collections import deque
from typing import Any, Optional
class CustomerServiceQueue:
    """
    A queue for managing customer service requests using FIFO order,
    optimized using collections.deque for efficient enqueue and dequeue operations.
    """
    def __init__(self) -> None:
        self._queue: deque[Any] = deque[Any]()  # private deque to store requests
    def enqueue(self, request: Any) -> None:
        """Add a service request to the end of the queue."""
        self._queue.append(request)
    def dequeue(self) -> Optional[Any]:
        """
        Remove and return the first service request from the front of the queue.
        Returns None if the queue is empty.
        """
        if self.is_empty():
            return None
        return self._queue.popleft()  # O(1) operation
    def is_empty(self) -> bool:
        """Check if the queue is empty."""
        return len(self._queue) == 0
    def size(self) -> int:
        """Return the number of requests in the queue."""
        return len(self._queue)
    def __str__(self) -> str:
        """Return a string representation of the queue."""
        return " <- ".join(str(req) for req in self._queue)
# ----------------- Test Function -----------------
def test_customer_service_queue():
    queue = CustomerServiceQueue()
```



```python
def test_customer_service_queue():
    print("Queue empty?", queue.is_empty())
    print("Dequeue from empty queue:", queue.dequeue())
    # Enqueue requests
    queue.enqueue("Request 1")
    queue.enqueue("Request 2")
    queue.enqueue("Request 3")
    print("Queue after enqueues:", queue)
    # Dequeue requests
    print("Dequeued:", queue.dequeue())
    print("Queue now:", queue)
    # Test size and empty checks
    print("Queue size:", queue.size())
    print("Queue empty?", queue.is_empty())
    # Dequeue remaining requests
    queue.dequeue()
    queue.dequeue()
    print("Queue after removing all requests:", queue)
    print("Dequeue from empty queue:", queue.dequeue())
# Run the test
test_customer_service_queue()
```

OUTPUT:



```
Queue empty? True
Dequeue from empty queue: None
Queue after enqueues: Request 1 <- Request 2 <- Request 3
Dequeued: Request 1
Queue now: Request 2 <- Request 3
Queue size: 2
Queue empty? False
Queue after removing all requests:
Dequeue from empty queue: None
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

EXPLANATION:

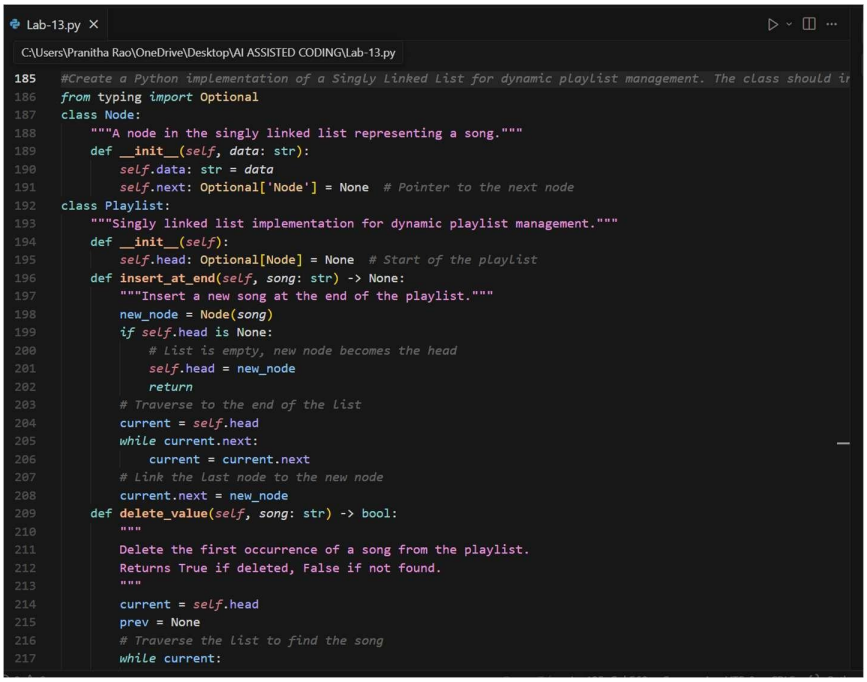
In a customer service system, a list-based queue can be implemented using a Python list where enqueue is performed with append() and dequeue with pop(0). While adding requests is efficient (O(1)), removing requests from the front is slow (O(n)) because all remaining elements must be shifted, which can cause delays as the number of requests grows. An optimized alternative uses collections.deque, where both append() and popleft() operations run in O(1) time. This makes deque-based queues highly efficient for FIFO operations, allowing the system to handle large numbers of service requests quickly and consistently, ensuring that requests are processed in the order they arrive without performance degradation.

Task-3: Singly Linked List for Dynamic Playlist Management

PROMPT:

Create a Python implementation of a Singly Linked List for dynamic playlist management. The class should include insert_at_end(song) to add a song at the end, delete_value(song) to remove a song by value, and traverse() to display all songs in order. Add inline comments explaining pointer manipulation and highlight tricky parts such as inserting at the end, deleting the head, or handling a single-node list. Also, suggest and implement test cases for edge scenarios, including an empty list, a single-node list, and deletion at the head, middle, and tail.

CODE:



Lab-13.py

C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-13.py

```
192  class Playlist:
209      def delete_value(self, song: str) -> bool:
218              if current.data == song:
219                  if prev is None:
220                      # Deleting the head node
221                      self.head = current.next
222                  else:
223                      # Skip over the current node
224                      prev.next = current.next
225                  return True
226              prev = current
227              current = current.next
228          return False  # Song not found
229      def traverse(self) -> None:
230          """Print all songs in the playlist in order."""
231          current = self.head
232          if not current:
233              print("Playlist is empty.")
234              return
235          songs = []
236          while current:
237              songs.append(current.data)
238              current = current.next
239          print(" -> ".join(songs))
240  # ----------------- Test Function ------------------
241  def test_playlist():
242      playlist = Playlist()
243      # Edge case: traverse empty playlist
244      playlist.traverse()
245      # Insert songs
246      playlist.insert_at_end("Song A")
247      playlist.insert_at_end("Song B")
248      playlist.insert_at_end("Song C")
```

Lab-13.py

C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-13.py

```
241  def test_playlist():
249      playlist.traverse()
250      # Delete head song
251      playlist.delete_value("Song A")
252      playlist.traverse()
253      # Delete middle song
254      playlist.delete_value("Song B")
255      playlist.traverse()
256      # Delete tail song
257      playlist.delete_value("Song C")
258      playlist.traverse()
259      # Attempt to delete from empty list
260      result = playlist.delete_value("Song X")
261      print("Delete non-existent song:", result)
262  # Run test
263  test_playlist()
264
```

OUTPUT:

```
Playlist is empty.
Song A -> Song B -> Song C
Song B -> Song C
Song C
Playlist is empty.
Delete non-existent song: False
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

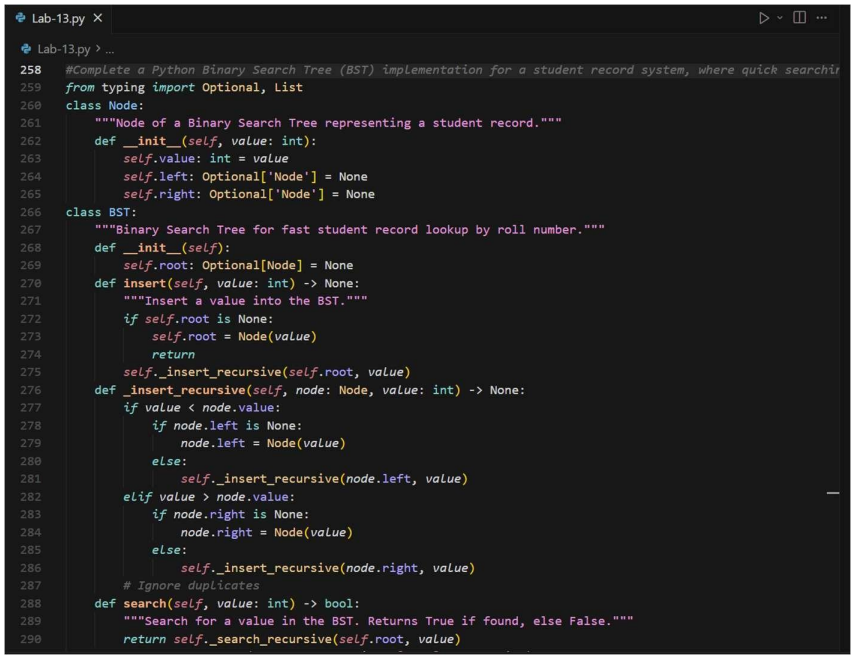
EXPLANATION:

The Playlist class uses a singly linked list to manage songs dynamically. insert_at_end adds a song at the end by traversing the list and linking the new node, while delete_value removes a song by updating pointers, handling edge cases like deleting the head, tail, or a single-node list. traverse prints all songs in order, or indicates if the playlist is empty. This structure allows efficient insertion and deletion without shifting elements, and the test cases cover empty lists, single-node lists, and deletions at different positions to ensure robust functionality.

Task-4: Binary Search Tree for Fast Record Lookup PROMPT:

Complete a Python Binary Search Tree (BST) implementation for a student record system, where quick searching by roll number is needed. The class should include insert(value) to add a new roll number, search(value) to check if a roll number exists, and inorder_traversal() to return all roll numbers in sorted order. Add meaningful docstrings, ensure correct BST behavior, and provide a brief explanation of how BST improves search efficiency compared to linear search, including best-case and worst-case performance.

CODE:



```python
class BST:
    def _search_recursive(self, node: Optional[Node], value: int) -> bool:
        if node is None:
            return False
        if value == node.value:
            return True
        elif value < node.value:
            return self._search_recursive(node.left, value)
        else:
            return self._search_recursive(node.right, value)
    def inorder_traversal(self) -> List[int]:
        """Return a list of values from the BST in sorted order."""
        result: List[int] = []
        self._inorder_recursive(self.root, result)
        return result
    def _inorder_recursive(self, node: Optional[Node], result: List[int]) -> None:
        if node:
            self._inorder_recursive(node.left, result)
            result.append(node.value)
            self._inorder_recursive(node.right, result)
# ----------------- Test Function -----------------
def test_bst():
    bst = BST()
    # Insert roll numbers
    for roll in [50, 30, 70, 20, 40, 60, 80]:
        bst.insert(roll)
    print("Inorder Traversal (sorted):", bst.inorder_traversal())
    print("Search 40:", bst.search(40))
    print("Search 90:", bst.search(90))
test_bst()
```

OUTPUT:

```
Inorder Traversal (sorted): [20, 30, 40, 50, 60, 70, 80]
Search 40: True
Search 90: False
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

EXPLANATION:

The BST stores student roll numbers so that each node's left child has smaller values and the right child has larger values. This structure allows search to skip half of the remaining nodes at each step, improving efficiency over linear search. In the best case, the BST is balanced, giving O(log n) search time, while in the worst case, it becomes skewed like a linked list, leading to O(n) search time. Inorder traversal returns all records in sorted order, making BST both efficient and ordered for record lookup.

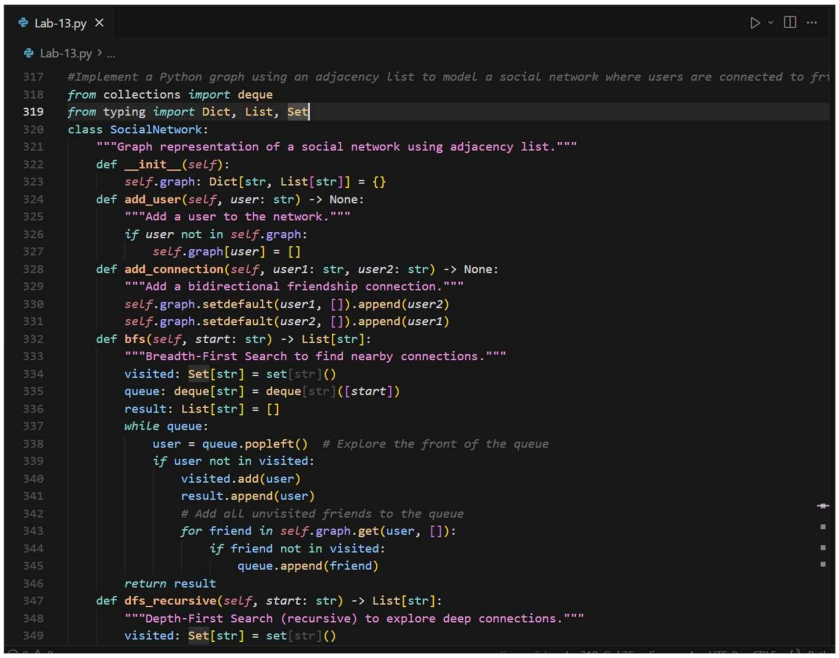
Task-5: Graph Traversal for Social Network Connections

PROMPT:

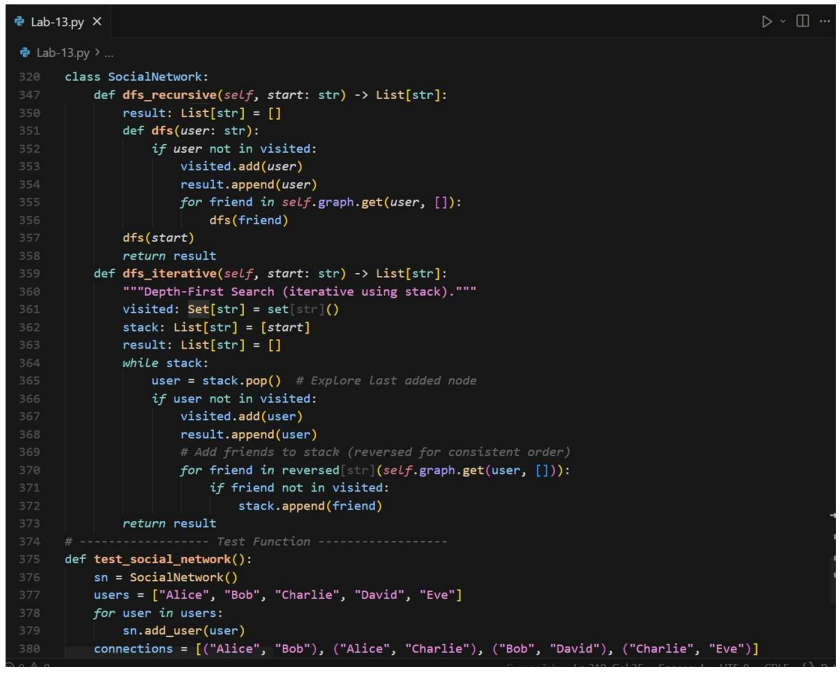
Implement a Python graph using an adjacency list to model a social network where users are connected to friends. Include Breadth-First Search (BFS) to find nearby connections and Depth-First Search (DFS) to explore deeper connection paths. Add

inline comments explaining traversal steps, compare recursive and iterative DFS approaches, and suggest practical use cases for BFS versus DFS.

CODE:



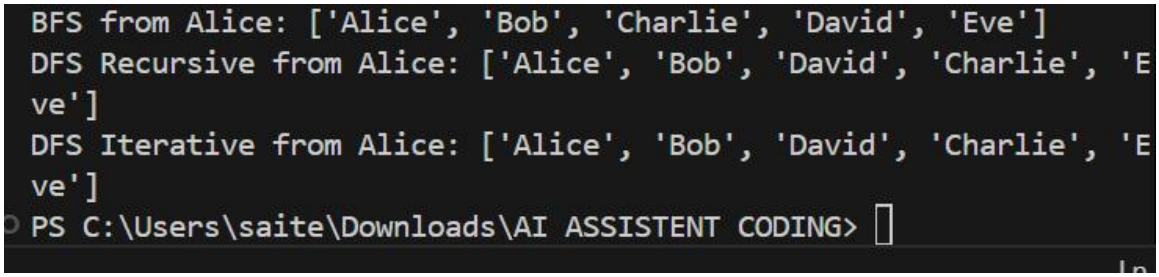
```python
#Implement a Python graph using an adjacency list to model a social network where users are connected to fri
from collections import deque
from typing import Dict, List, Set
class SocialNetwork:
    """Graph representation of a social network using adjacency list."""
    def __init__(self):
        self.graph: Dict[str, List[str]] = {}
    def add_user(self, user: str) -> None:
        """Add a user to the network."""
        if user not in self.graph:
            self.graph[user] = []
    def add_connection(self, user1: str, user2: str) -> None:
        """Add a bidirectional friendship connection."""
        self.graph.setdefault(user1, []).append(user2)
        self.graph.setdefault(user2, []).append(user1)
    def bfs(self, start: str) -> List[str]:
        """Breadth-First Search to find nearby connections."""
        visited: Set[str] = set[str]()
        queue: deque[str] = deque[str]([start])
        result: List[str] = []
        while queue:
            user = queue.popleft()  # Explore the front of the queue
            if user not in visited:
                visited.add(user)
                result.append(user)
                # Add all unvisited friends to the queue
                for friend in self.graph.get(user, []):
                    if friend not in visited:
                        queue.append(friend)
        return result
    def dfs_recursive(self, start: str) -> List[str]:
        """Depth-First Search (recursive) to explore deep connections."""
        visited: Set[str] = set[str]()
```



```python
class SocialNetwork:
    def dfs_recursive(self, start: str) -> List[str]:
        result: List[str] = []
        def dfs(user: str):
            if user not in visited:
                visited.add(user)
                result.append(user)
                for friend in self.graph.get(user, []):
                    dfs(friend)
        dfs(start)
        return result
    def dfs_iterative(self, start: str) -> List[str]:
        """Depth-First Search (iterative using stack)."""
        visited: Set[str] = set[str]()
        stack: List[str] = [start]
        result: List[str] = []
        while stack:
            user = stack.pop()  # Explore last added node
            if user not in visited:
                visited.add(user)
                result.append(user)
                # Add friends to stack (reversed for consistent order)
                for friend in reversed[str](self.graph.get(user, [])):
                    if friend not in visited:
                        stack.append(friend)
        return result
# ------------------ Test Function ------------------
def test_social_network():
    sn = SocialNetwork()
    users = ["Alice", "Bob", "Charlie", "David", "Eve"]
    for user in users:
        sn.add_user(user)
    connections = [("Alice", "Bob"), ("Alice", "Charlie"), ("Bob", "David"), ("Charlie", "Eve")]
```

```python
def test_social_network():
    for u1, u2 in connections:
        sn.add_connection(u1, u2)
    print("BFS from Alice:", sn.bfs("Alice"))
    print("DFS Recursive from Alice:", sn.dfs_recursive("Alice"))
    print("DFS Iterative from Alice:", sn.dfs_iterative("Alice"))
test_social_network()
```

OUTPUT:



EXPLANATION:

The graph uses an adjacency list to represent user connections. BFS explores nearby friends level by level, making it ideal for finding shortest paths or friends-of-friends. DFS explores as deep as possible along each path, either recursively or iteratively with a stack, useful for tracing long connection chains or detecting cycles. Recursive DFS is simpler to write, while iterative DFS avoids recursion limits. BFS is practical for recommendations or network reach, whereas DFS is better for in-depth network exploration or backtracking tasks.