AI Assisted Coding Lab 11.4

Veekshitha Adharasani

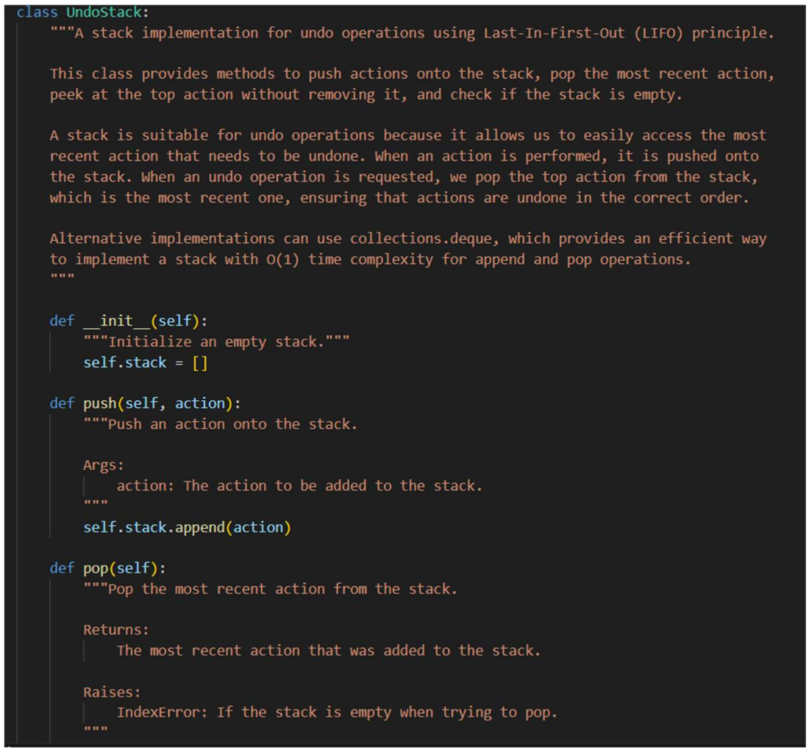
2303A52175

Batch – 41

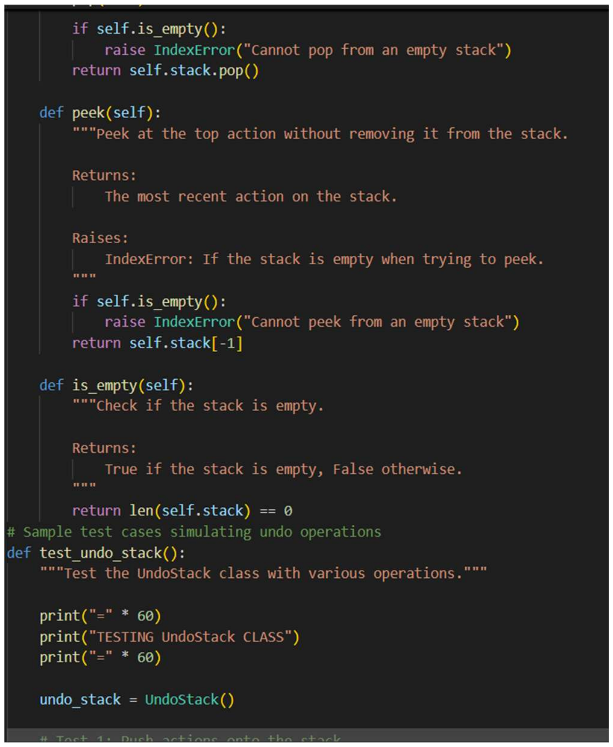
Question 1: Stack Implementation for Undo Operations (LIFO)

Prompt:

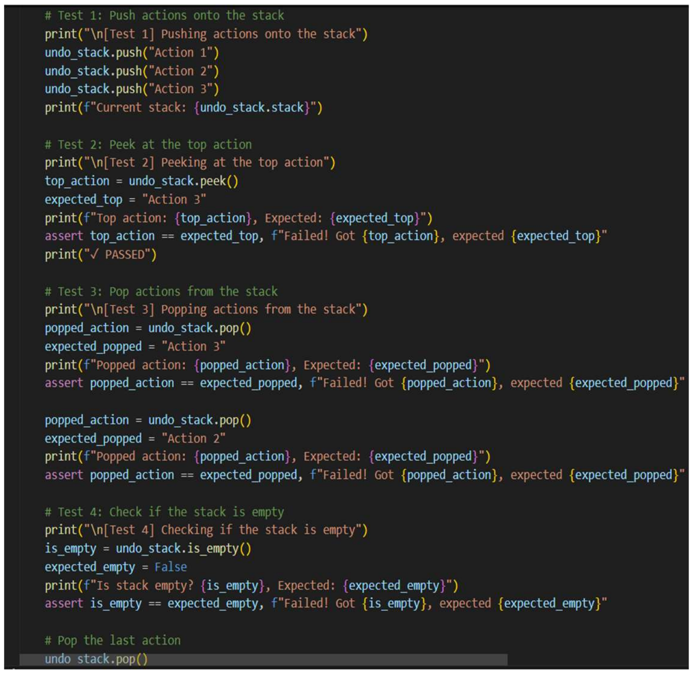
Generate a python code for stack implementation for Undo opeartions(LIFO). Implement a stack class in python with push(action), pop(),peek(),is_empty() methods. Generate a clean class skeleton with docstrings and explain why a stack is suitable for undo operations. Suggest alternative implementations using collections.deque. Generate sample test cases to demonstrate the functionality of the stack class.



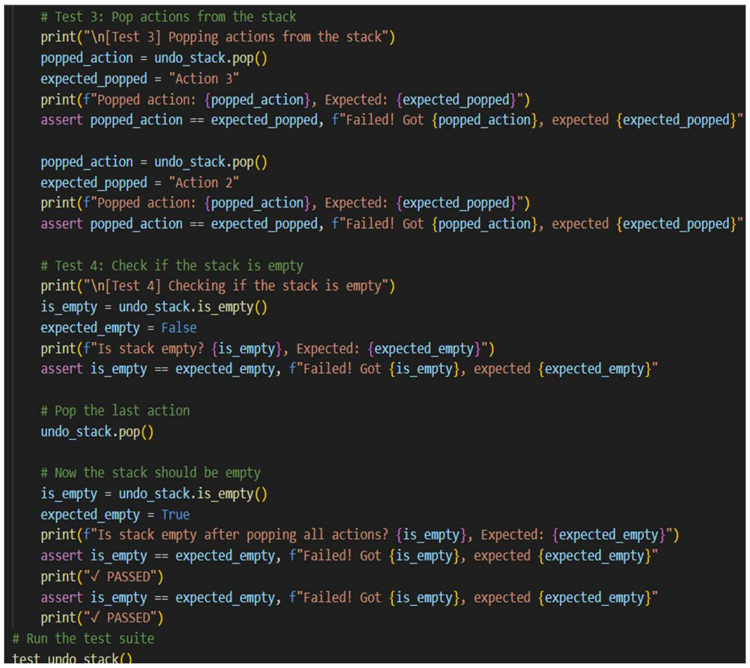
```python
class UndoStack:
    """A stack implementation for undo operations using Last-In-First-Out (LIFO) principle.

    This class provides methods to push actions onto the stack, pop the most recent action,
    peek at the top action without removing it, and check if the stack is empty.

    A stack is suitable for undo operations because it allows us to easily access the most
    recent action that needs to be undone. When an action is performed, it is pushed onto
    the stack. When an undo operation is requested, we pop the top action from the stack,
    which is the most recent one, ensuring that actions are undone in the correct order.

    Alternative implementations can use collections.deque, which provides an efficient way
    to implement a stack with O(1) time complexity for append and pop operations.
    """

    def __init__(self):
        """Initialize an empty stack."""
        self.stack = []

    def push(self, action):
        """Push an action onto the stack.

        Args:
            action: The action to be added to the stack.
        """
        self.stack.append(action)

    def pop(self):
        """Pop the most recent action from the stack.

        Returns:
            The most recent action that was added to the stack.

        Raises:
            IndexError: If the stack is empty when trying to pop.
        """
```



```python
        if self.is_empty():
            raise IndexError("Cannot pop from an empty stack")
        return self.stack.pop()

    def peek(self):
        """Peek at the top action without removing it from the stack.

        Returns:
            The most recent action on the stack.

        Raises:
            IndexError: If the stack is empty when trying to peek.
        """
        if self.is_empty():
            raise IndexError("Cannot peek from an empty stack")
        return self.stack[-1]

    def is_empty(self):
        """Check if the stack is empty.

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self.stack) == 0
# Sample test cases simulating undo operations
def test_undo_stack():
    """Test the UndoStack class with various operations."""

    print("=" * 60)
    print("TESTING UndoStack CLASS")
    print("=" * 60)

    undo_stack = UndoStack()

    # Test 1: Push actions onto the stack
```

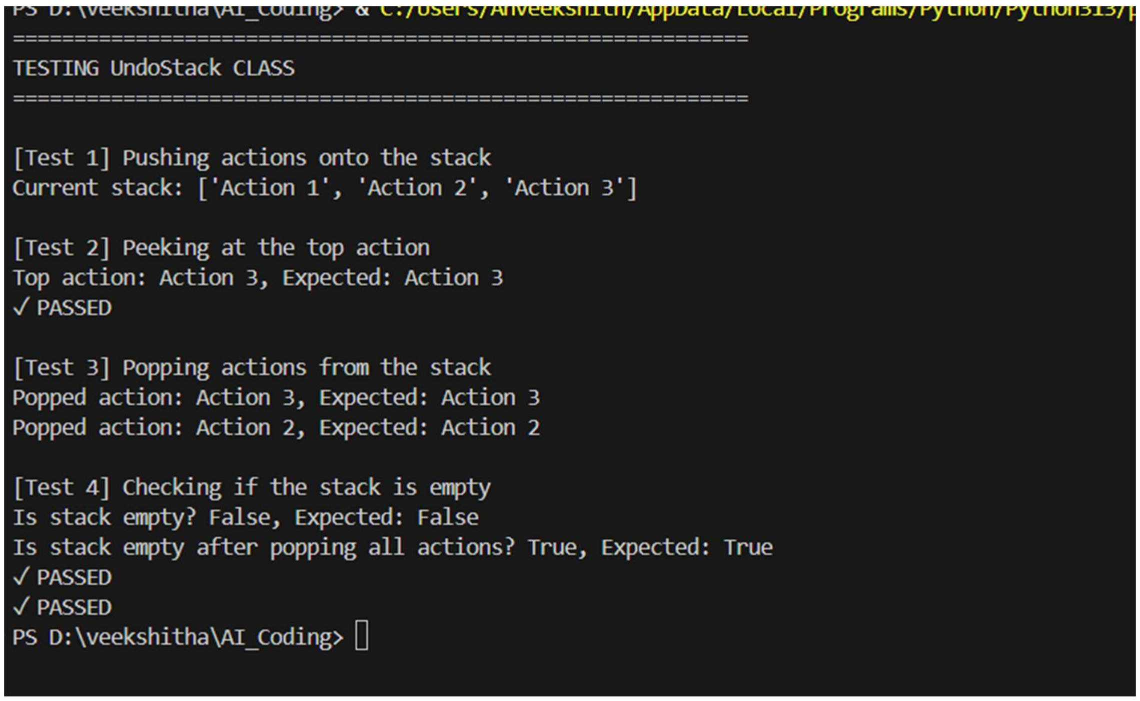


```python
    # Test 1: Push actions onto the stack
    print("\n[Test 1] Pushing actions onto the stack")
    undo_stack.push("Action 1")
    undo_stack.push("Action 2")
    undo_stack.push("Action 3")
    print(f"Current stack: {undo_stack.stack}")

    # Test 2: Peek at the top action
    print("\n[Test 2] Peeking at the top action")
    top_action = undo_stack.peek()
    expected_top = "Action 3"
    print(f"Top action: {top_action}, Expected: {expected_top}")
    assert top_action == expected_top, f"Failed! Got {top_action}, expected {expected_top}"
    print("✓ PASSED")

    # Test 3: Pop actions from the stack
    print("\n[Test 3] Popping actions from the stack")
    popped_action = undo_stack.pop()
    expected_popped = "Action 3"
    print(f"Popped action: {popped_action}, Expected: {expected_popped}")
    assert popped_action == expected_popped, f"Failed! Got {popped_action}, expected {expected_popped}"

    popped_action = undo_stack.pop()
    expected_popped = "Action 2"
    print(f"Popped action: {popped_action}, Expected: {expected_popped}")
    assert popped_action == expected_popped, f"Failed! Got {popped_action}, expected {expected_popped}"

    # Test 4: Check if the stack is empty
    print("\n[Test 4] Checking if the stack is empty")
    is_empty = undo_stack.is_empty()
    expected_empty = False
    print(f"Is stack empty? {is_empty}, Expected: {expected_empty}")
    assert is_empty == expected_empty, f"Failed! Got {is_empty}, expected {expected_empty}"

    # Pop the last action
    undo_stack.pop()
```



```python
    # Test 3: Pop actions from the stack
    print("\n[Test 3] Popping actions from the stack")
    popped_action = undo_stack.pop()
    expected_popped = "Action 3"
    print(f"Popped action: {popped_action}, Expected: {expected_popped}")
    assert popped_action == expected_popped, f"Failed! Got {popped_action}, expected {expected_popped}"

    popped_action = undo_stack.pop()
    expected_popped = "Action 2"
    print(f"Popped action: {popped_action}, Expected: {expected_popped}")
    assert popped_action == expected_popped, f"Failed! Got {popped_action}, expected {expected_popped}"

    # Test 4: Check if the stack is empty
    print("\n[Test 4] Checking if the stack is empty")
    is_empty = undo_stack.is_empty()
    expected_empty = False
    print(f"Is stack empty? {is_empty}, Expected: {expected_empty}")
    assert is_empty == expected_empty, f"Failed! Got {is_empty}, expected {expected_empty}"

    # Pop the last action
    undo_stack.pop()

    # Now the stack should be empty
    is_empty = undo_stack.is_empty()
    expected_empty = True
    print(f"Is stack empty after popping all actions? {is_empty}, Expected: {expected_empty}")
    assert is_empty == expected_empty, f"Failed! Got {is_empty}, expected {expected_empty}"
    print("✓ PASSED")
    assert is_empty == expected_empty, f"Failed! Got {is_empty}, expected {expected_empty}"
    print("✓ PASSED")
# Run the test suite
test_undo_stack()
```

Output:



```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Python/Python313/p
============================================================
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
Is stack empty after popping all actions? True, Expected: True
✓ PASSED
✓ PASSED
PS D:\veekshitha\AI_Coding>
```

Explanation:

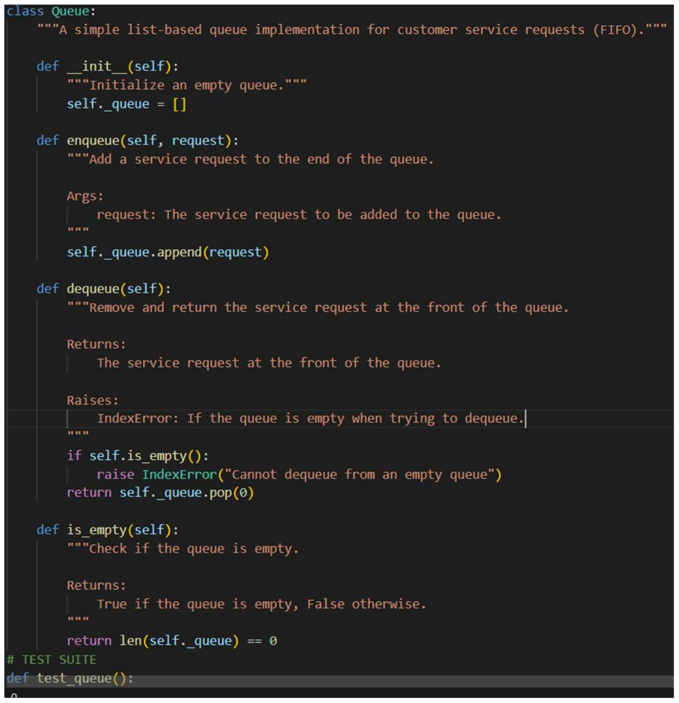
The list-based implementation of the UndoStack uses a standard Python list to store actions, which allows for easy appending and popping of elements. However, it can be less efficient for large stacks due to potential resizing operations when the list grows. On the other hand, using collections.deque provides a more efficient implementation for stack operations, as it is optimized for fast appends and pops from both ends with O(1) time complexity. Deque also handles memory more efficiently, making it a better choice for larger stacks or when performance is a concern. In summary, while the list-based implementation is simpler and sufficient for small stacks, the deque-based implementation offers improved performance and memory management for larger stacks or more frequent operations.

Question 2: Queue for Customer Service Requests (FIFO)

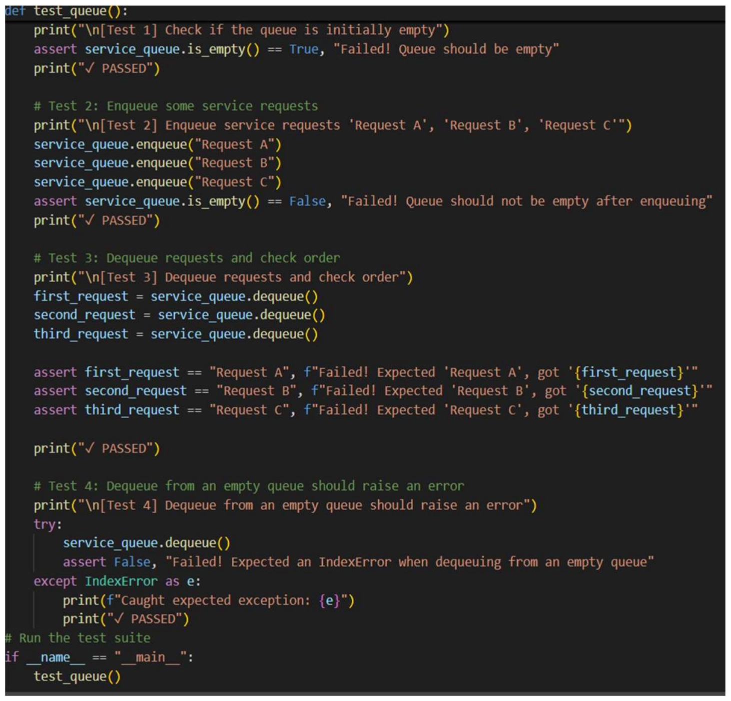
List Based Queue:

Prompt:

Generate a python code for queue for customer service request(FIFO). Here the code must handle the service requests in the order they arrive. Implement a queue class with enqueue(request),dequeue(),is_empty(). Generate the code in list-based queue implementation.



```python
class Queue:
    """A simple list-based queue implementation for customer service requests (FIFO)."""

    def __init__(self):
        """Initialize an empty queue."""
        self._queue = []

    def enqueue(self, request):
        """Add a service request to the end of the queue.

        Args:
            request: The service request to be added to the queue.
        """
        self._queue.append(request)

    def dequeue(self):
        """Remove and return the service request at the front of the queue.

        Returns:
            The service request at the front of the queue.

        Raises:
            IndexError: If the queue is empty when trying to dequeue.
        """
        if self.is_empty():
            raise IndexError("Cannot dequeue from an empty queue")
        return self._queue.pop(0)

    def is_empty(self):
        """Check if the queue is empty.

        Returns:
            True if the queue is empty, False otherwise.
        """
        return len(self._queue) == 0
# TEST SUITE
def test_queue():
```

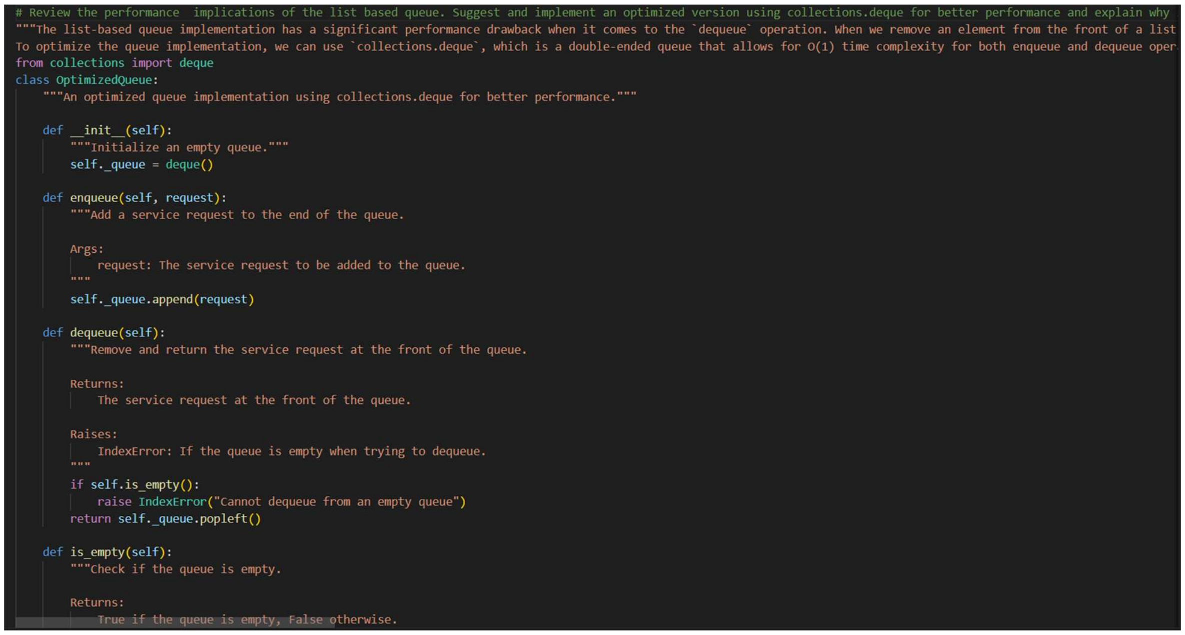


```python
def test_queue():
    print("\n[Test 1] Check if the queue is initially empty")
    assert service_queue.is_empty() == True, "Failed! Queue should be empty"
    print("✓ PASSED")

    # Test 2: Enqueue some service requests
    print("\n[Test 2] Enqueue service requests 'Request A', 'Request B', 'Request C'")
    service_queue.enqueue("Request A")
    service_queue.enqueue("Request B")
    service_queue.enqueue("Request C")
    assert service_queue.is_empty() == False, "Failed! Queue should not be empty after enqueuing"
    print("✓ PASSED")

    # Test 3: Dequeue requests and check order
    print("\n[Test 3] Dequeue requests and check order")
    first_request = service_queue.dequeue()
    second_request = service_queue.dequeue()
    third_request = service_queue.dequeue()

    assert first_request == "Request A", f"Failed! Expected 'Request A', got '{first_request}'"
    assert second_request == "Request B", f"Failed! Expected 'Request B', got '{second_request}'"
    assert third_request == "Request C", f"Failed! Expected 'Request C', got '{third_request}'"

    print("✓ PASSED")

    # Test 4: Dequeue from an empty queue should raise an error
    print("\n[Test 4] Dequeue from an empty queue should raise an error")
    try:
        service_queue.dequeue()
        assert False, "Failed! Expected an IndexError when dequeuing from an empty queue"
    except IndexError as e:
        print(f"Caught expected exception: {e}")
        print("✓ PASSED")
# Run the test suite
if __name__ == "__main__":
    test_queue()
```

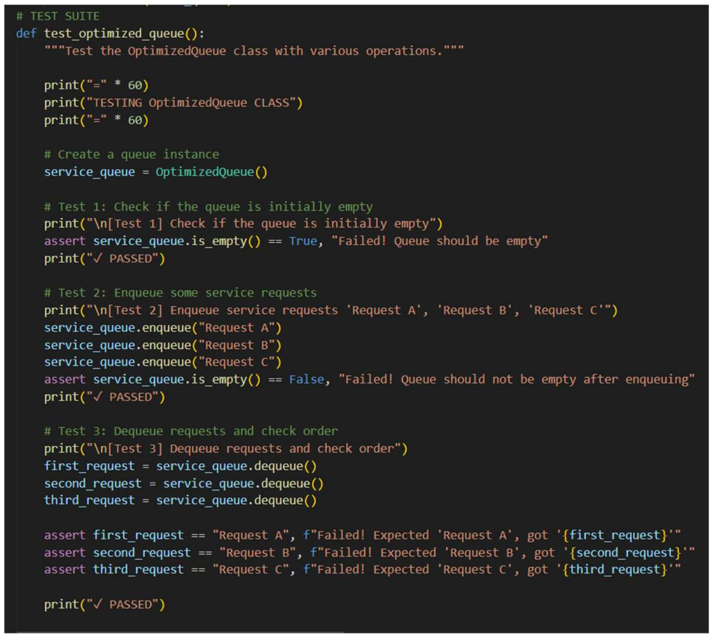
Deque-based optimized queue:

Prompt:

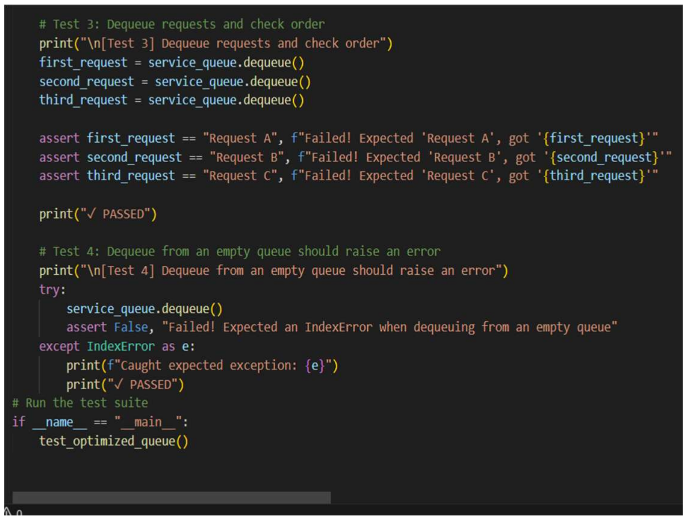
Review the performance implications of the list-based queue. Suggest and implement an optimized version using collections.deque for better performance and explain why it is more efficient. Generate an optimized code using collections.deque for the queue implementation.



```python
# Review the performance  implications of the list based queue. Suggest and implement an optimized version using collections.deque for better performance and explain why
"""The list-based queue implementation has a significant performance drawback when it comes to the `dequeue` operation. When we remove an element from the front of a list
To optimize the queue implementation, we can use `collections.deque`, which is a double-ended queue that allows for O(1) time complexity for both enqueue and dequeue oper
from collections import deque
class OptimizedQueue:
    """An optimized queue implementation using collections.deque for better performance."""

    def __init__(self):
        """Initialize an empty queue."""
        self._queue = deque()

    def enqueue(self, request):
        """Add a service request to the end of the queue.

        Args:
            request: The service request to be added to the queue.
        """
        self._queue.append(request)

    def dequeue(self):
        """Remove and return the service request at the front of the queue.

        Returns:
            The service request at the front of the queue.

        Raises:
            IndexError: If the queue is empty when trying to dequeue.
        """
        if self.is_empty():
            raise IndexError("Cannot dequeue from an empty queue")
        return self._queue.popleft()

    def is_empty(self):
        """Check if the queue is empty.

        Returns:
            True if the queue is empty, False otherwise.
```



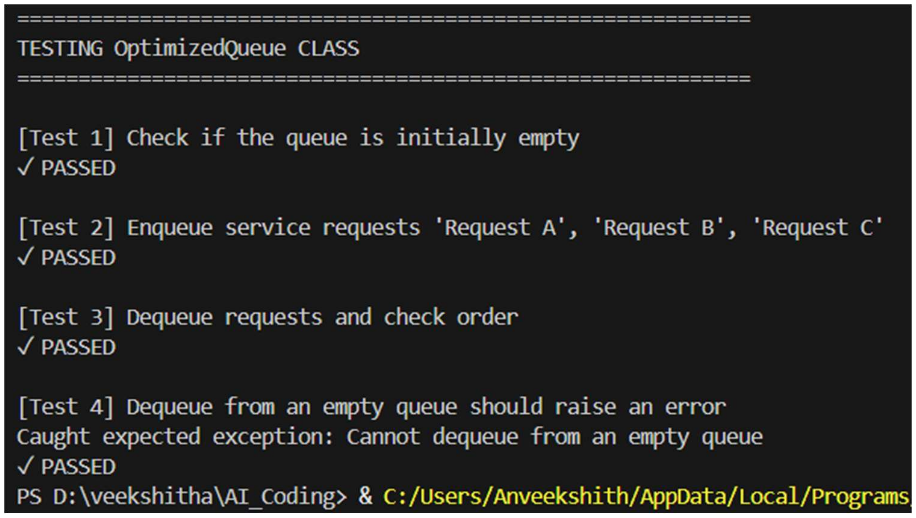
```python
# TEST SUITE
def test_optimized_queue():
    """Test the OptimizedQueue class with various operations."""

    print("=" * 60)
    print("TESTING OptimizedQueue CLASS")
    print("=" * 60)

    # Create a queue instance
    service_queue = OptimizedQueue()

    # Test 1: Check if the queue is initially empty
    print("\n[Test 1] Check if the queue is initially empty")
    assert service_queue.is_empty() == True, "Failed! Queue should be empty"
    print("✓ PASSED")

    # Test 2: Enqueue some service requests
    print("\n[Test 2] Enqueue service requests 'Request A', 'Request B', 'Request C'")
    service_queue.enqueue("Request A")
    service_queue.enqueue("Request B")
    service_queue.enqueue("Request C")
    assert service_queue.is_empty() == False, "Failed! Queue should not be empty after enqueuing"
    print("✓ PASSED")

    # Test 3: Dequeue requests and check order
    print("\n[Test 3] Dequeue requests and check order")
    first_request = service_queue.dequeue()
    second_request = service_queue.dequeue()
    third_request = service_queue.dequeue()

    assert first_request == "Request A", f"Failed! Expected 'Request A', got '{first_request}'"
    assert second_request == "Request B", f"Failed! Expected 'Request B', got '{second_request}'"
    assert third_request == "Request C", f"Failed! Expected 'Request C', got '{third_request}'"

    print("✓ PASSED")
```



```python
    # Test 3: Dequeue requests and check order
    print("\n[Test 3] Dequeue requests and check order")
    first_request = service_queue.dequeue()
    second_request = service_queue.dequeue()
    third_request = service_queue.dequeue()

    assert first_request == "Request A", f"Failed! Expected 'Request A', got '{first_request}'"
    assert second_request == "Request B", f"Failed! Expected 'Request B', got '{second_request}'"
    assert third_request == "Request C", f"Failed! Expected 'Request C', got '{third_request}'"

    print("✓ PASSED")
```

```python
    # Test 4: Dequeue from an empty queue should raise an error
    print("\n[Test 4] Dequeue from an empty queue should raise an error")
    try:
        service_queue.dequeue()
        assert False, "Failed! Expected an IndexError when dequeuing from an empty queue"
    except IndexError as e:
        print(f"Caught expected exception: {e}")
        print("✓ PASSED")
# Run the test suite
if __name__ == "__main__":
    test_optimized_queue()
```

Output:



```
============================================================
TESTING OptimizedQueue CLASS
============================================================

[Test 1] Check if the queue is initially empty
✓ PASSED

[Test 2] Enqueue service requests 'Request A', 'Request B', 'Request C'
✓ PASSED

[Test 3] Dequeue requests and check order
✓ PASSED

[Test 4] Dequeue from an empty queue should raise an error
Caught expected exception: Cannot dequeue from an empty queue
✓ PASSED
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs
```

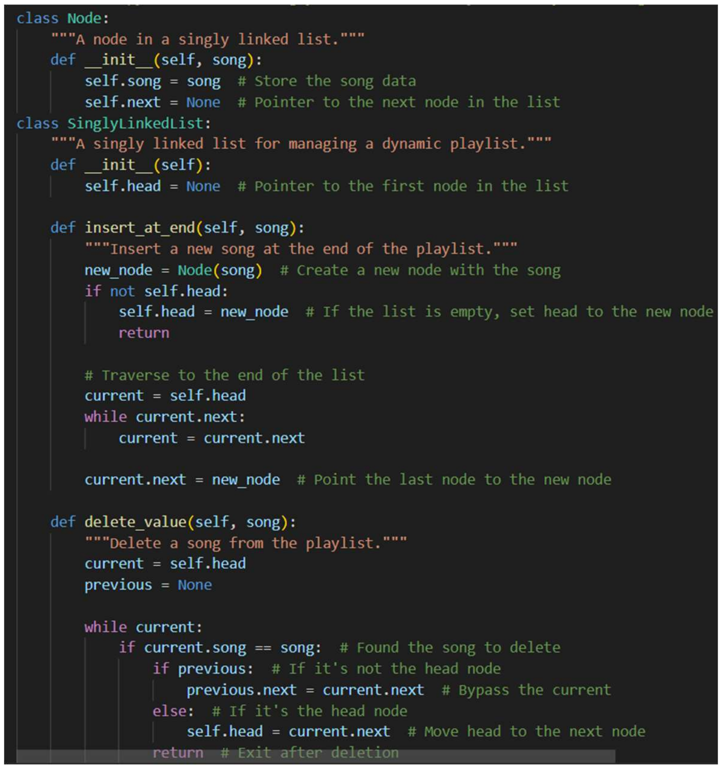
Explanation:

The list-based queue has O(n) time complexity for dequeue operations due to element shifting, which can lead to significant performance degradation as the queue grows. In contrast, the `collections.deque` implementation provides O(1) time complexity for both enqueue and dequeue operations, ensuring consistent performance regardless of queue size. This optimization is crucial in real-world applications where high throughput and low latency are required, such as in web servers or task scheduling systems.

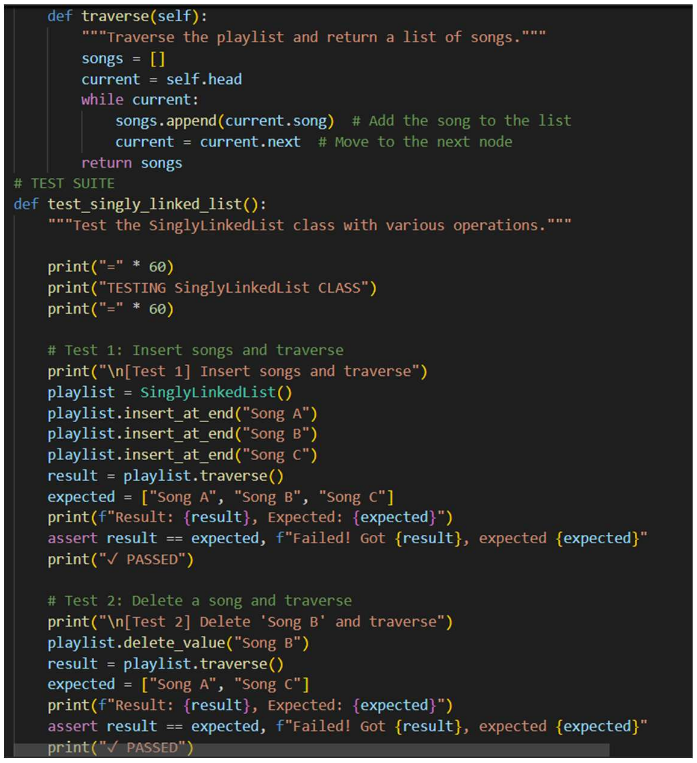
Question 3: Singly Linked List for Dynamic Playlist Management

Prompt:

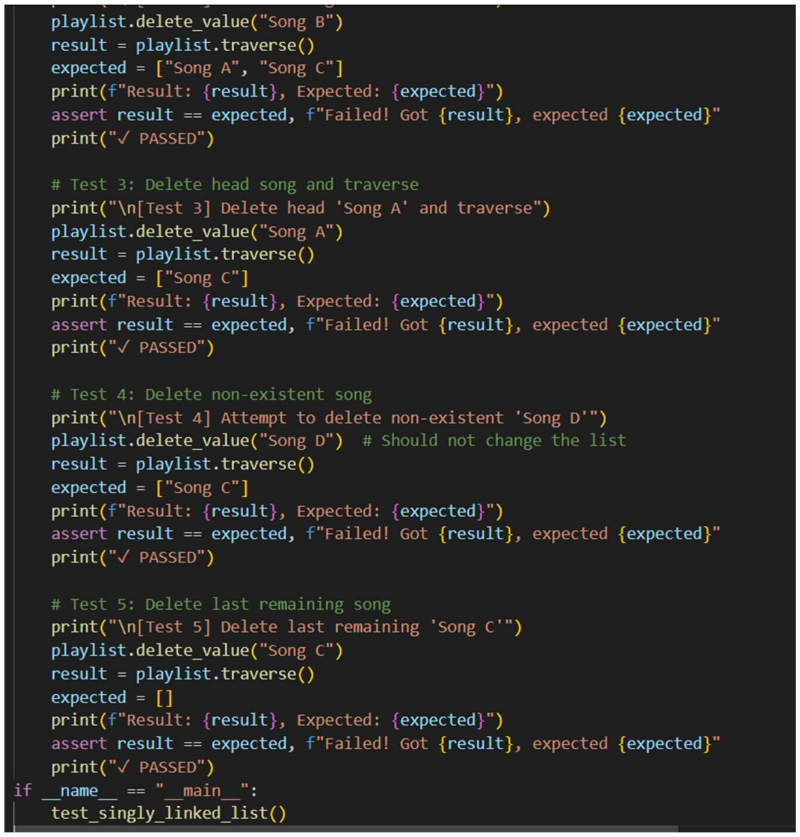
Generate a python code for Singly Linked List for Dynamic Playlist Management. Implememt a Singly Linked list with insert_at_end(song), delete_value(song), traverse(). Add inline comments explaining pointer manipulation. Highlight tricky parts of insertion and deletion. Generate test cases validating all operations. Also add edge case test scenerios. Generate a fully functioned linked list implementation.



```python
class Node:
    """A node in a singly linked list."""
    def __init__(self, song):
        self.song = song  # Store the song data
        self.next = None  # Pointer to the next node in the list
class SinglyLinkedList:
    """A singly linked list for managing a dynamic playlist."""
    def __init__(self):
        self.head = None  # Pointer to the first node in the list

    def insert_at_end(self, song):
        """Insert a new song at the end of the playlist."""
        new_node = Node(song)  # Create a new node with the song
        if not self.head:
            self.head = new_node  # If the list is empty, set head to the new node
            return

        # Traverse to the end of the list
        current = self.head
        while current.next:
            current = current.next

        current.next = new_node  # Point the last node to the new node

    def delete_value(self, song):
        """Delete a song from the playlist."""
        current = self.head
        previous = None

        while current:
            if current.song == song:  # Found the song to delete
                if previous:  # If it's not the head node
                    previous.next = current.next  # Bypass the current
                else:  # If it's the head node
                    self.head = current.next  # Move head to the next node
                return  # Exit after deletion
```

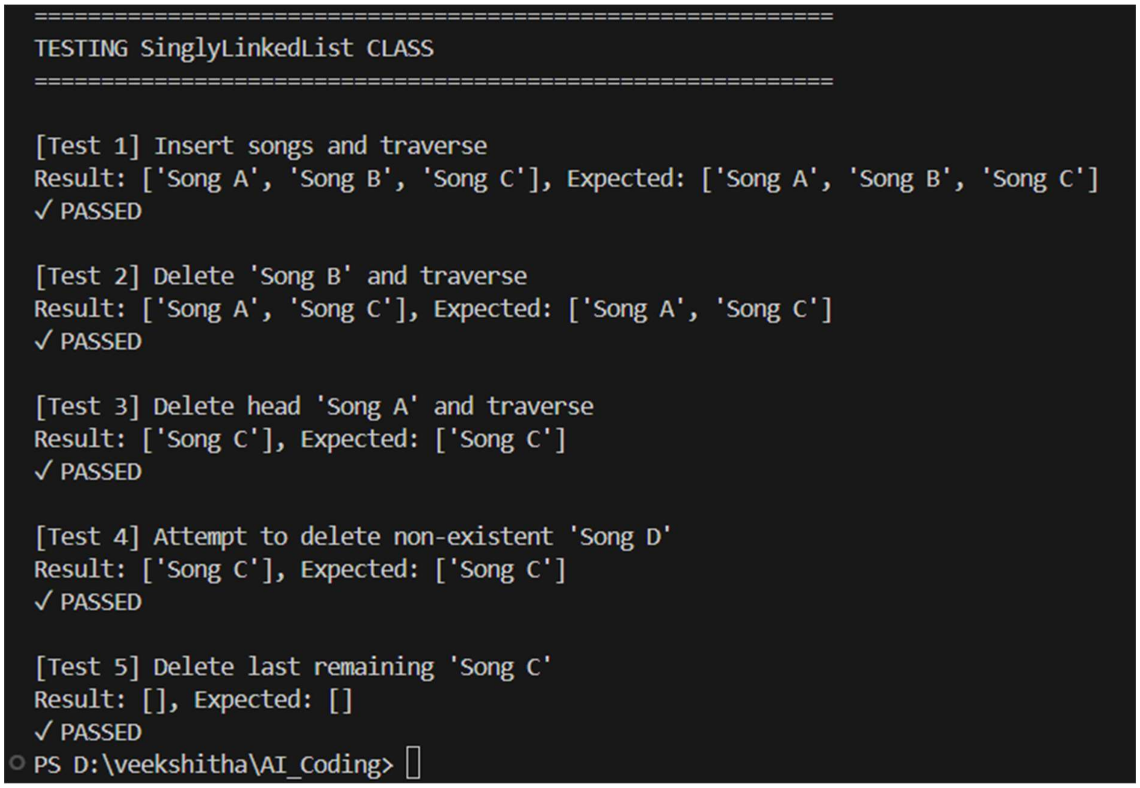


```python
    def traverse(self):
        """Traverse the playlist and return a list of songs."""
        songs = []
        current = self.head
        while current:
            songs.append(current.song)  # Add the song to the list
            current = current.next  # Move to the next node
        return songs
# TEST SUITE
def test_singly_linked_list():
    """Test the SinglyLinkedList class with various operations."""

    print("=" * 60)
    print("TESTING SinglyLinkedList CLASS")
    print("=" * 60)

    # Test 1: Insert songs and traverse
    print("\n[Test 1] Insert songs and traverse")
    playlist = SinglyLinkedList()
    playlist.insert_at_end("Song A")
    playlist.insert_at_end("Song B")
    playlist.insert_at_end("Song C")
    result = playlist.traverse()
    expected = ["Song A", "Song B", "Song C"]
    print(f"Result: {result}, Expected: {expected}")
    assert result == expected, f"Failed! Got {result}, expected {expected}"
    print("✓ PASSED")

    # Test 2: Delete a song and traverse
    print("\n[Test 2] Delete 'Song B' and traverse")
    playlist.delete_value("Song B")
    result = playlist.traverse()
    expected = ["Song A", "Song C"]
    print(f"Result: {result}, Expected: {expected}")
    assert result == expected, f"Failed! Got {result}, expected {expected}"
    print("✓ PASSED")
```



```python
    playlist.delete_value("Song B")
    result = playlist.traverse()
    expected = ["Song A", "Song C"]
    print(f"Result: {result}, Expected: {expected}")
    assert result == expected, f"Failed! Got {result}, expected {expected}"
    print("✓ PASSED")

    # Test 3: Delete head song and traverse
    print("\n[Test 3] Delete head 'Song A' and traverse")
    playlist.delete_value("Song A")
    result = playlist.traverse()
    expected = ["Song C"]
    print(f"Result: {result}, Expected: {expected}")
    assert result == expected, f"Failed! Got {result}, expected {expected}"
    print("✓ PASSED")

    # Test 4: Delete non-existent song
    print("\n[Test 4] Attempt to delete non-existent 'Song D'")
    playlist.delete_value("Song D")  # Should not change the list
    result = playlist.traverse()
    expected = ["Song C"]
    print(f"Result: {result}, Expected: {expected}")
    assert result == expected, f"Failed! Got {result}, expected {expected}"
    print("✓ PASSED")

    # Test 5: Delete last remaining song
    print("\n[Test 5] Delete last remaining 'Song C'")
    playlist.delete_value("Song C")
    result = playlist.traverse()
    expected = []
    print(f"Result: {result}, Expected: {expected}")
    assert result == expected, f"Failed! Got {result}, expected {expected}"
    print("✓ PASSED")
if __name__ == "__main__":
    test_singly_linked_list()
```

Output:



```
============================================================
TESTING SinglyLinkedList CLASS
============================================================

[Test 1] Insert songs and traverse
Result: ['Song A', 'Song B', 'Song C'], Expected: ['Song A', 'Song B', 'Song C']
✓ PASSED

[Test 2] Delete 'Song B' and traverse
Result: ['Song A', 'Song C'], Expected: ['Song A', 'Song C']
✓ PASSED

[Test 3] Delete head 'Song A' and traverse
Result: ['Song C'], Expected: ['Song C']
✓ PASSED

[Test 4] Attempt to delete non-existent 'Song D'
Result: ['Song C'], Expected: ['Song C']
✓ PASSED

[Test 5] Delete last remaining 'Song C'
Result: [], Expected: []
✓ PASSED
PS D:\veekshitha\AI_Coding>
```

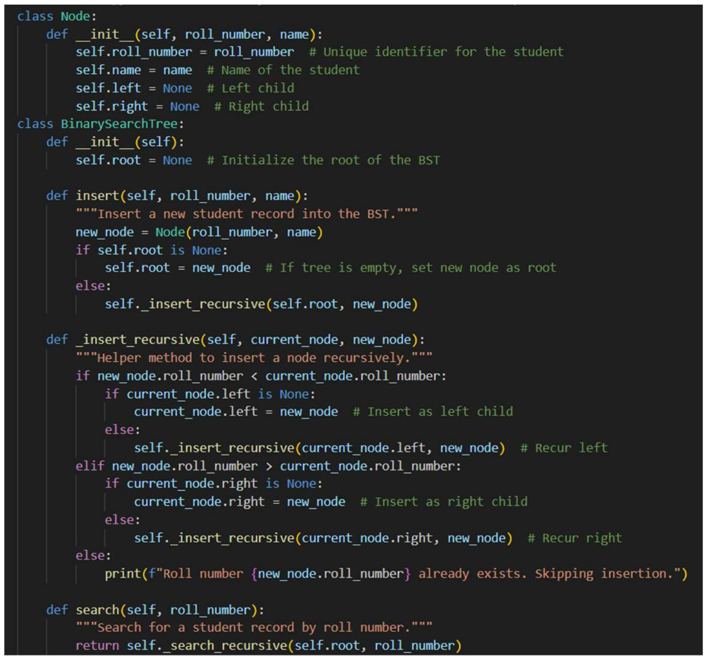
Explanation:

The code implements a singly linked list to manage a dynamic playlist. It defines a `Node` class to represent each song and a `SinglyLinkedList` class to handle the playlist operations. The `insert_at_end` method adds songs to the end of the list, while the `delete_value` method removes a specified song. The `traverse` method returns a list of all songs in the playlist. The test suite validates these operations, including edge cases such as deleting non-existent songs and handling an empty playlist.

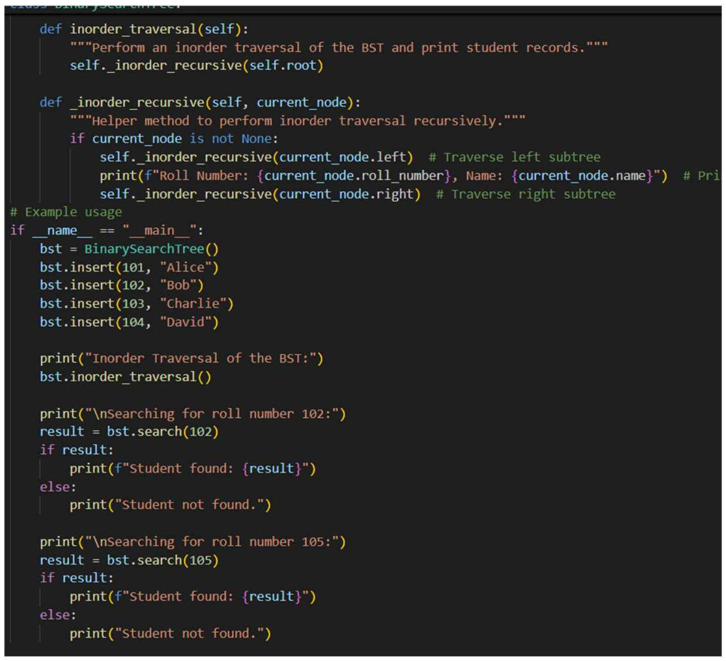
Question 4: Binary Search Tree for Fast Record Lookup

Prompt:

Generate a python code of Binary Search Tree for Fast Record Lookup. Build a student record system where quick seraching by roll number is required. Implement the following functionalities: insert(value), search(value), inorder_traversal(). Add meaningful comments to the code. The code should be well-structured and clean with proper insert and serach behavior and output.

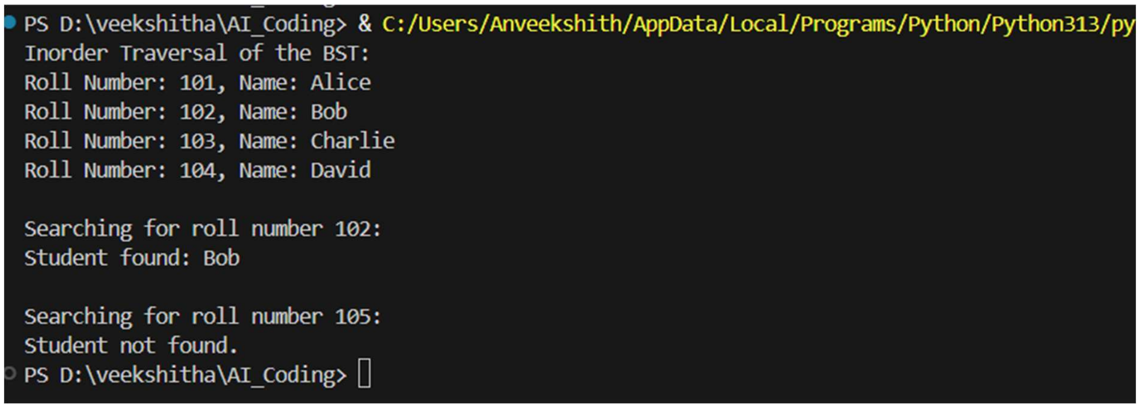


```python
class Node:
    def __init__(self, roll_number, name):
        self.roll_number = roll_number  # Unique identifier for the student
        self.name = name  # Name of the student
        self.left = None  # Left child
        self.right = None  # Right child
class BinarySearchTree:
    def __init__(self):
        self.root = None  # Initialize the root of the BST

    def insert(self, roll_number, name):
        """Insert a new student record into the BST."""
        new_node = Node(roll_number, name)
        if self.root is None:
            self.root = new_node  # If tree is empty, set new node as root
        else:
            self._insert_recursive(self.root, new_node)

    def _insert_recursive(self, current_node, new_node):
        """Helper method to insert a node recursively."""
        if new_node.roll_number < current_node.roll_number:
            if current_node.left is None:
                current_node.left = new_node  # Insert as left child
            else:
                self._insert_recursive(current_node.left, new_node)  # Recur left
        elif new_node.roll_number > current_node.roll_number:
            if current_node.right is None:
                current_node.right = new_node  # Insert as right child
            else:
                self._insert_recursive(current_node.right, new_node)  # Recur right
        else:
            print(f"Roll number {new_node.roll_number} already exists. Skipping insertion.")

    def search(self, roll_number):
        """Search for a student record by roll number."""
        return self._search_recursive(self.root, roll_number)
```



```python
    def inorder_traversal(self):
        """Perform an inorder traversal of the BST and print student records."""
        self._inorder_recursive(self.root)

    def _inorder_recursive(self, current_node):
        """Helper method to perform inorder traversal recursively."""
        if current_node is not None:
            self._inorder_recursive(current_node.left)  # Traverse left subtree
            print(f"Roll Number: {current_node.roll_number}, Name: {current_node.name}")  # Pri
            self._inorder_recursive(current_node.right)  # Traverse right subtree
# Example usage
if __name__ == "__main__":
    bst = BinarySearchTree()
    bst.insert(101, "Alice")
    bst.insert(102, "Bob")
    bst.insert(103, "Charlie")
    bst.insert(104, "David")

    print("Inorder Traversal of the BST:")
    bst.inorder_traversal()

    print("\nSearching for roll number 102:")
    result = bst.search(102)
    if result:
        print(f"Student found: {result}")
    else:
        print("Student not found.")

    print("\nSearching for roll number 105:")
    result = bst.search(105)
    if result:
        print(f"Student found: {result}")
    else:
        print("Student not found.")
```

Output:



```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Python/Python313/py
Inorder Traversal of the BST:
Roll Number: 101, Name: Alice
Roll Number: 102, Name: Bob
Roll Number: 103, Name: Charlie
Roll Number: 104, Name: David

Searching for roll number 102:
Student found: Bob

Searching for roll number 105:
Student not found.
PS D:\veekshitha\AI_Coding> 
```

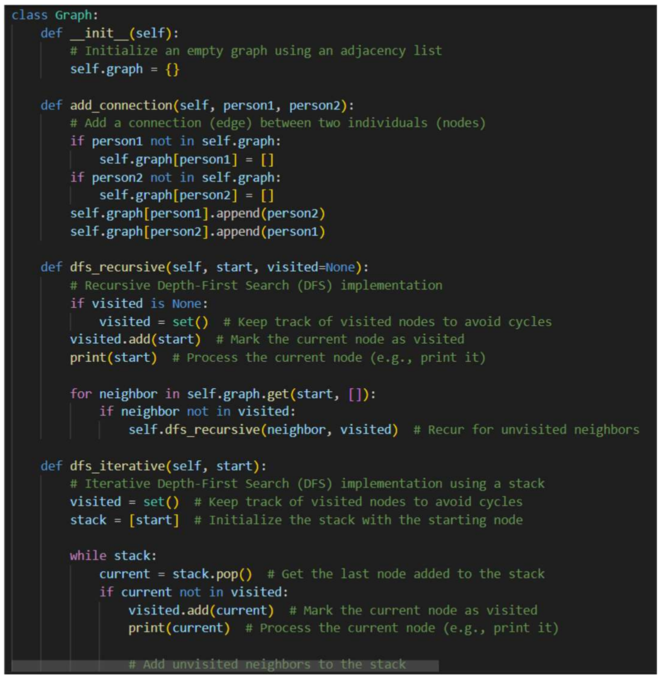
Explanation:

A Binary Search Tree (BST) improves search efficiency compared to a linear search by organizing data in a hierarchical structure. In a BST, each node has at most two children, and the left child contains values less than the parent node while the right child contains values greater than the parent node. This allows for a logarithmic time complexity (O(log n)) for search operations in a balanced BST, as it effectively halves the search space with each comparison. In contrast, a linear search has a time complexity of O(n) since it may require checking each element sequentially. The best case performance of a BST occurs when the tree is perfectly balanced, while the worst case occurs when the tree is skewed (e.g., all nodes are on one side), resulting in O(n) time complexity similar to a linear search.

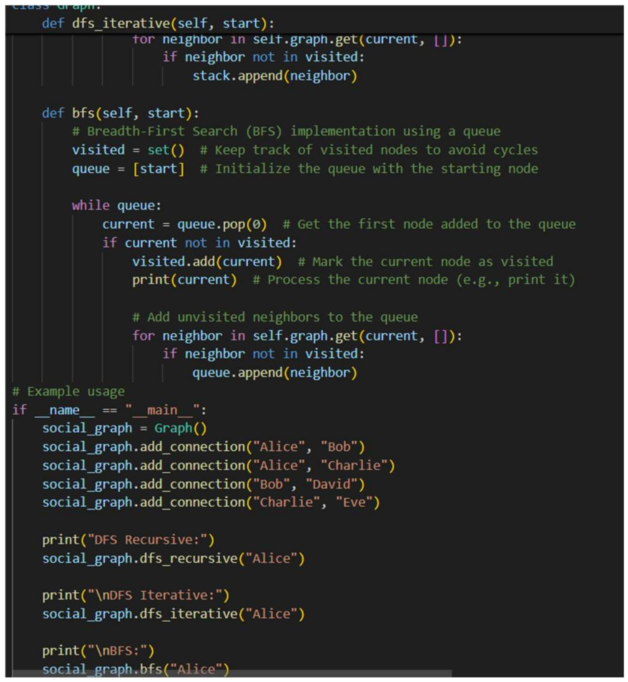
Question 5: Graph Traversal for Social Network Connections

Prompt:

Generate a python code of Graph Traversal for Social Network Connections. Represent the social network as a graph, where nodes represent individuals and edges represent connections between them. Implement both Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms to traverse the graph and find all connections for a given individual. Add inline comments explaining traversal steps. Compare recursive and iterative DFS approaches. Give a clear, clean and efficient implementation.

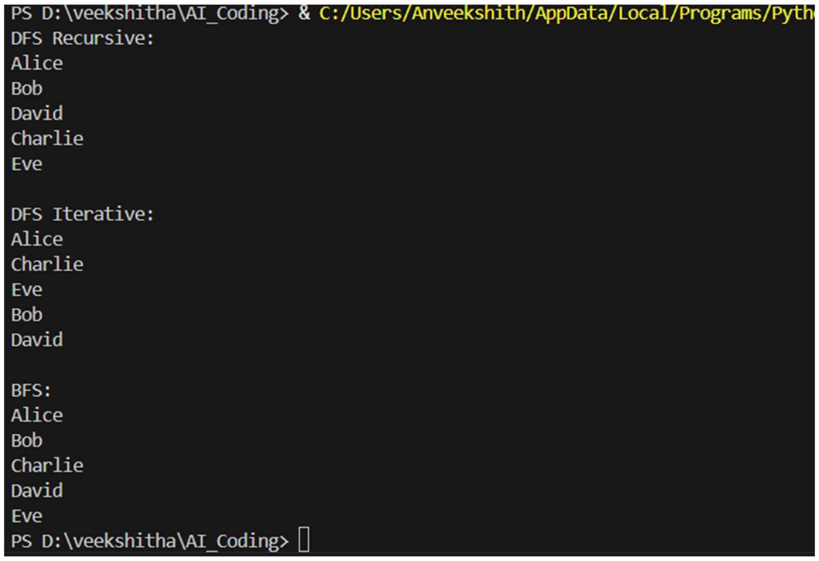


```python
class Graph:
    def __init__(self):
        # Initialize an empty graph using an adjacency list
        self.graph = {}

    def add_connection(self, person1, person2):
        # Add a connection (edge) between two individuals (nodes)
        if person1 not in self.graph:
            self.graph[person1] = []
        if person2 not in self.graph:
            self.graph[person2] = []
        self.graph[person1].append(person2)
        self.graph[person2].append(person1)

    def dfs_recursive(self, start, visited=None):
        # Recursive Depth-First Search (DFS) implementation
        if visited is None:
            visited = set()  # Keep track of visited nodes to avoid cycles
        visited.add(start)  # Mark the current node as visited
        print(start)  # Process the current node (e.g., print it)

        for neighbor in self.graph.get(start, []):
            if neighbor not in visited:
                self.dfs_recursive(neighbor, visited)  # Recur for unvisited neighbors

    def dfs_iterative(self, start):
        # Iterative Depth-First Search (DFS) implementation using a stack
        visited = set()  # Keep track of visited nodes to avoid cycles
        stack = [start]  # Initialize the stack with the starting node

        while stack:
            current = stack.pop()  # Get the last node added to the stack
            if current not in visited:
                visited.add(current)  # Mark the current node as visited
                print(current)  # Process the current node (e.g., print it)

                # Add unvisited neighbors to the stack
```



```python
    def dfs_iterative(self, start):
                for neighbor in self.graph.get(current, []):
                    if neighbor not in visited:
                        stack.append(neighbor)

    def bfs(self, start):
        # Breadth-First Search (BFS) implementation using a queue
        visited = set()  # Keep track of visited nodes to avoid cycles
        queue = [start]  # Initialize the queue with the starting node

        while queue:
            current = queue.pop(0)  # Get the first node added to the queue
            if current not in visited:
                visited.add(current)  # Mark the current node as visited
                print(current)  # Process the current node (e.g., print it)

                # Add unvisited neighbors to the queue
                for neighbor in self.graph.get(current, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
# Example usage
if __name__ == "__main__":
    social_graph = Graph()
    social_graph.add_connection("Alice", "Bob")
    social_graph.add_connection("Alice", "Charlie")
    social_graph.add_connection("Bob", "David")
    social_graph.add_connection("Charlie", "Eve")

    print("DFS Recursive:")
    social_graph.dfs_recursive("Alice")

    print("\nDFS Iterative:")
    social_graph.dfs_iterative("Alice")

    print("\nBFS:")
    social_graph.bfs("Alice")
```

Output:



```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Pyth
DFS Recursive:
Alice
Bob
David
Charlie
Eve

DFS Iterative:
Alice
Charlie
Eve
Bob
David

BFS:
Alice
Bob
Charlie
David
Eve
PS D:\veekshitha\AI_Coding> 
```

Explanation:

DFS (Depth-First Search) explores as far down a branch as possible before backtracking, making it suitable for scenarios like maze solving or pathfinding where you want to explore all possibilities. BFS (Breadth-First Search), on the other hand, explores all neighbors at the present depth before moving on to nodes at the next depth level, making it ideal for finding the shortest path in unweighted graphs or level-order traversal in trees. The recursive DFS approach is often more elegant and easier to understand, while the iterative approach can be more efficient in terms of memory usage, especially for deep graphs.