LAB ASSIGNMENT- 11.4

Name: Sindhu Katakam

Enrollment number: 2303A52481

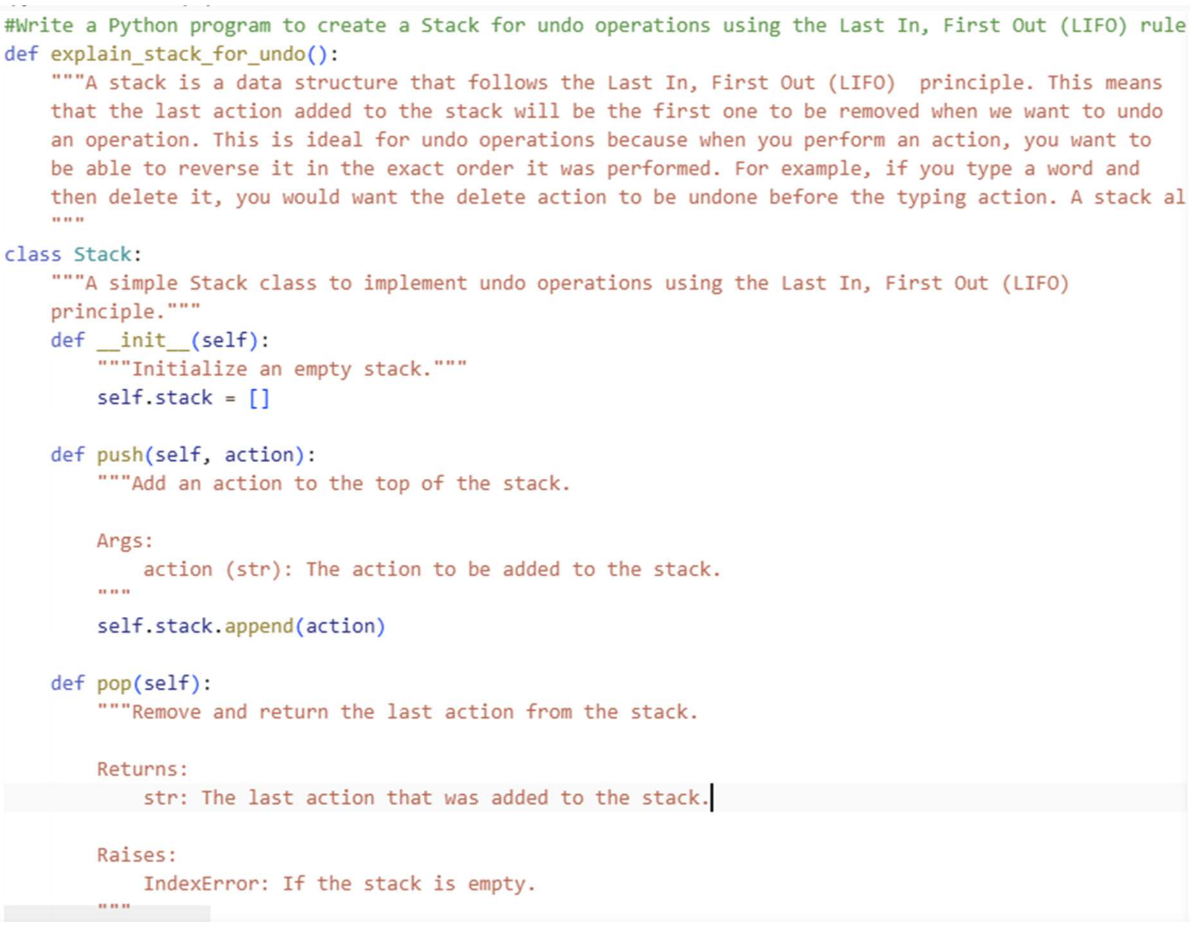
Batch-41

Task 1: Stack Implementation for Undo Operations (LIFO)

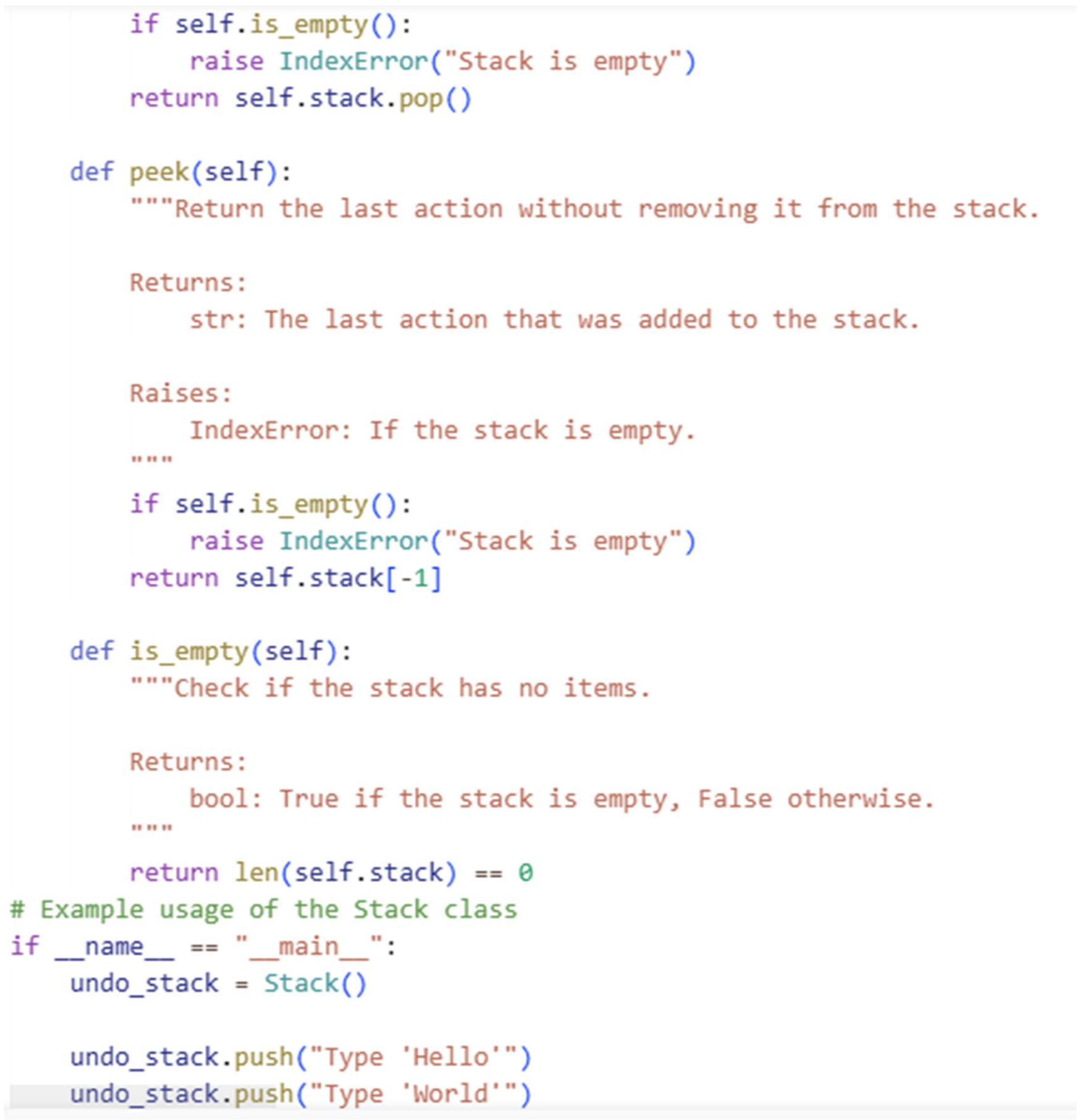
Prompt:

Write a Python program to create a Stack for undo operations using the Last In, First Out (LIFO) rule. Create a simple Stack class with clear docstrings and include these methods: push(action) to add an action, pop() to remove and return the last action, peek() to see the last action without removing it, and is_empty() to check if the stack has no items. Explain in simple words why a stack is good for undo operations. Also show another way to make a stack using collections.deque.

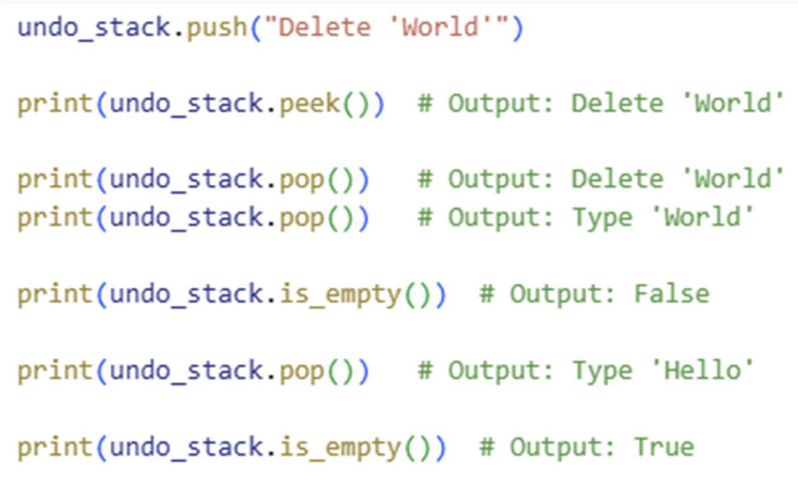
Code:



```python
#Write a Python program to create a Stack for undo operations using the Last In, First Out (LIFO) rule
def explain_stack_for_undo():
    """A stack is a data structure that follows the Last In, First Out (LIFO)  principle. This means
    that the last action added to the stack will be the first one to be removed when we want to undo
    an operation. This is ideal for undo operations because when you perform an action, you want to
    be able to reverse it in the exact order it was performed. For example, if you type a word and
    then delete it, you would want the delete action to be undone before the typing action. A stack al
    """
class Stack:
    """A simple Stack class to implement undo operations using the Last In, First Out (LIFO)
    principle."""
    def __init__(self):
        """Initialize an empty stack."""
        self.stack = []

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

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

    def pop(self):
        """Remove and return the last action from the stack.

        Returns:
            str: The last action that was added to the stack.

        Raises:
            IndexError: If the stack is empty.
        """
```

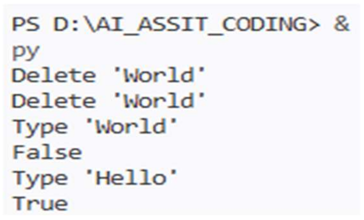


```python
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.stack.pop()

    def peek(self):
        """Return the last action without removing it from the stack.

        Returns:
            str: The last action that was added to the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Stack is empty")
        return self.stack[-1]

    def is_empty(self):
        """Check if the stack has no items.

        Returns:
            bool: True if the stack is empty, False otherwise.
        """
        return len(self.stack) == 0
# Example usage of the Stack class
if __name__ == "__main__":
    undo_stack = Stack()

    undo_stack.push("Type 'Hello'")
    undo_stack.push("Type 'World'")
```



```python
undo_stack.push("Delete 'World'")

print(undo_stack.peek())  # Output: Delete 'World'

print(undo_stack.pop())   # Output: Delete 'World'
print(undo_stack.pop())   # Output: Type 'World'

print(undo_stack.is_empty())  # Output: False

print(undo_stack.pop())   # Output: Type 'Hello'

print(undo_stack.is_empty())  # Output: True
```

Output:



```
PS D:\AI_ASSIT_CODING> &
py
Delete 'World'
Delete 'World'
Type 'World'
False
Type 'Hello'
True
```

Explanation:

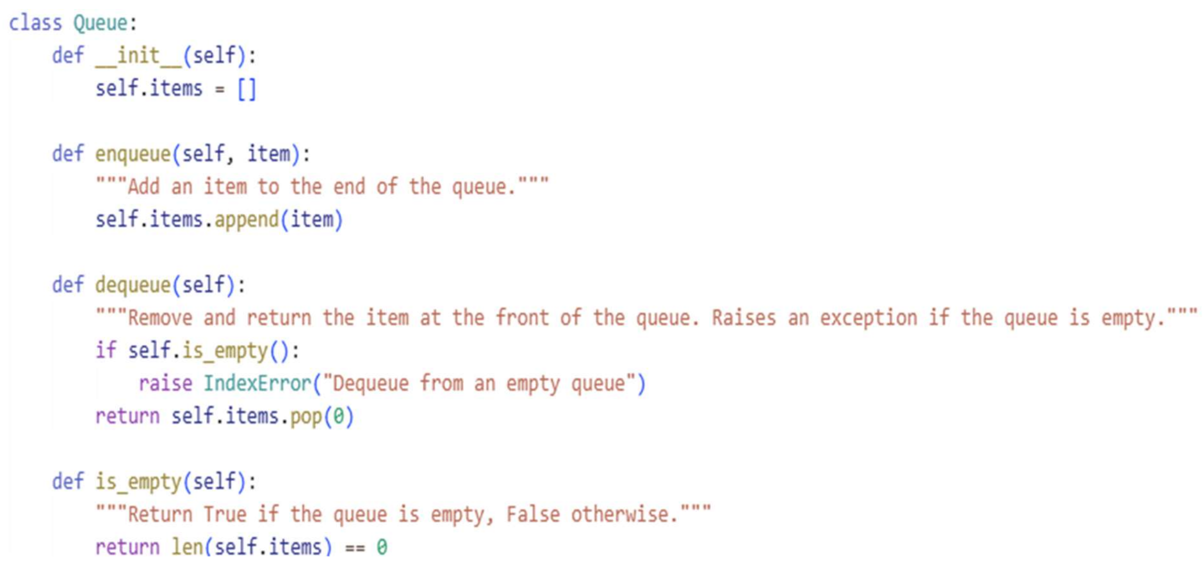
This code creates a simple Stack class in Python to help with undo operations, like those you see in text editors. A stack works by following the Last In, First Out (LIFO) rule, which means the last thing you add is the first thing you take out. This is perfect for undoing actions because you want to reverse your most recent step first. The Stack class has methods to add an action (push), remove the last action (pop), look at the last action without removing it (peek), and check if the stack is empty (is_empty). The code also shows how to use the stack with some easy examples, making it clear how undoing actions works step by step. Using a stack makes it easy to keep track of what you did last and undo things in the right order.

Task 2: Queue for Customer Service Requests (FIFO)

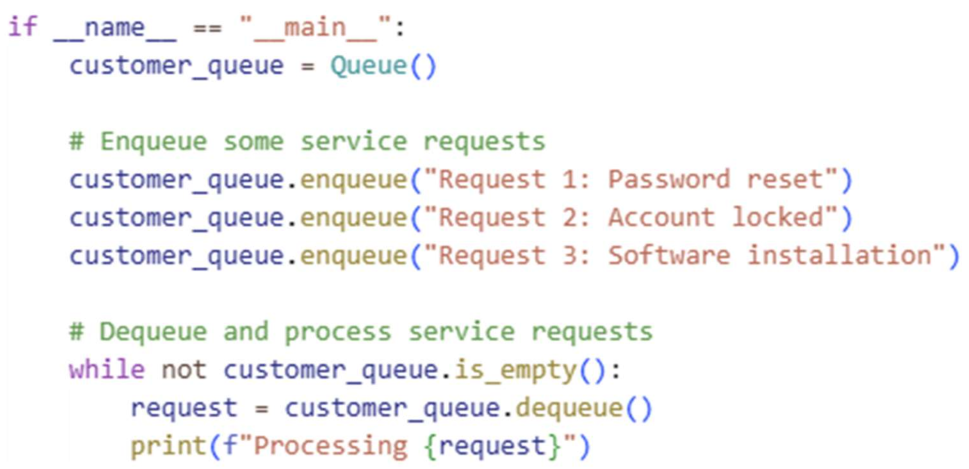
Prompt:

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,dequeue,is_empty methods.

Code:



```python
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        """Add an item to the end of the queue."""
        self.items.append(item)

    def dequeue(self):
        """Remove and return the item at the front of the queue. Raises an exception if the queue is empty."""
        if self.is_empty():
            raise IndexError("Dequeue from an empty queue")
        return self.items.pop(0)

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

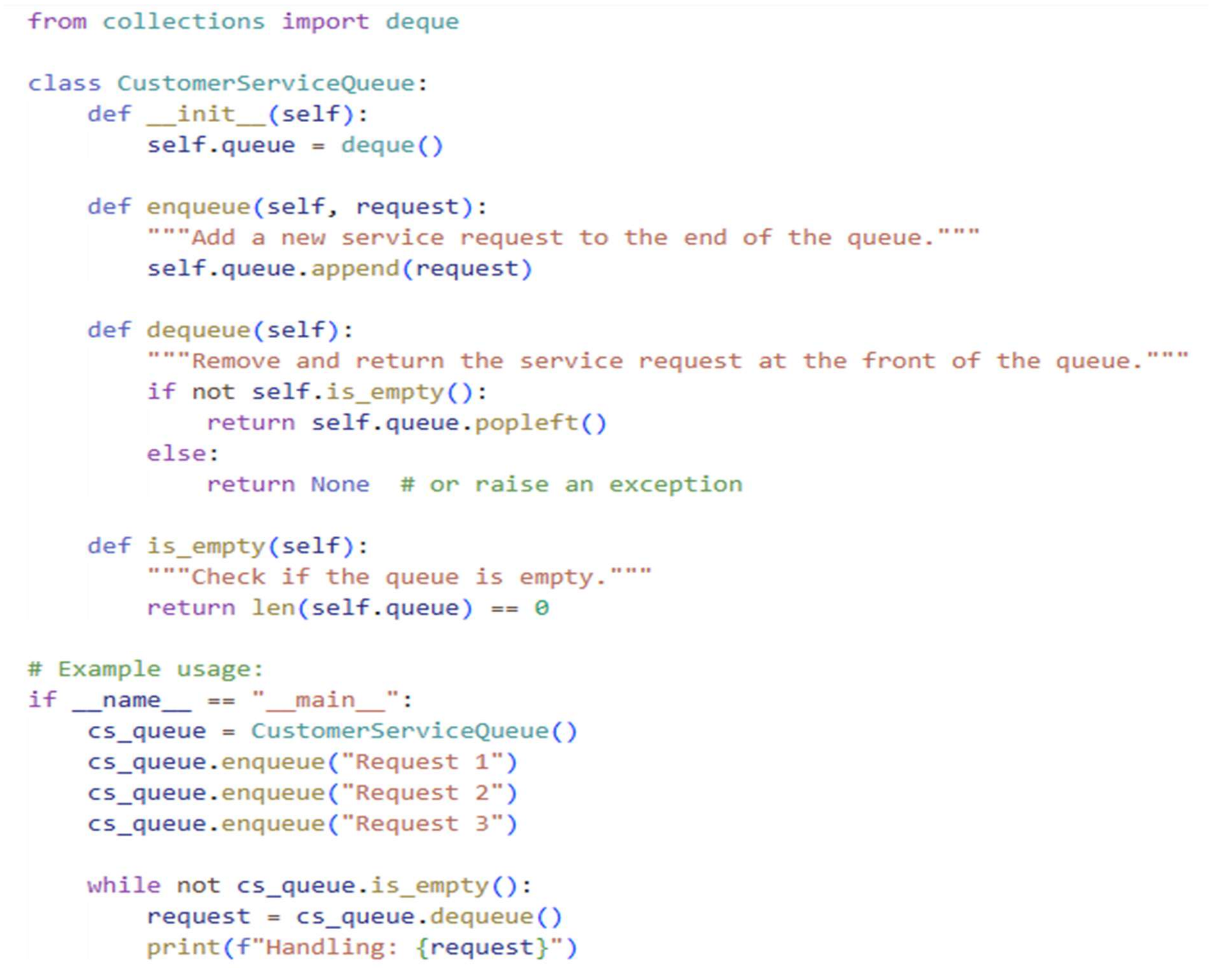


```python
if __name__ == "__main__":
    customer_queue = Queue()

    # Enqueue some service requests
    customer_queue.enqueue("Request 1: Password reset")
    customer_queue.enqueue("Request 2: Account locked")
    customer_queue.enqueue("Request 3: Software installation")

    # Dequeue and process service requests
    while not customer_queue.is_empty():
        request = customer_queue.dequeue()
        print(f"Processing {request}")
```

Deque-based optimized queue:

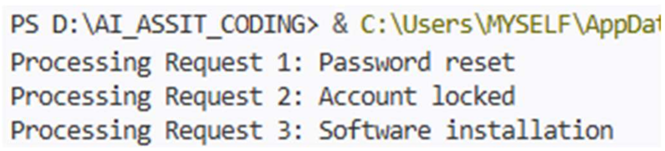
Review how using a list to create a queue affects performance. Suggest a better solution by using collections.deque, which works faster for queue operations. Explain why deque is more efficient, and write improved code that uses collections.deque to implement the queue.

Code:



```python
from collections import deque

class CustomerServiceQueue:
    def __init__(self):
        self.queue = deque()

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

    def dequeue(self):
        """Remove and return the service request at the front of the queue."""
        if not self.is_empty():
            return self.queue.popleft()
        else:
            return None  # or raise an exception

    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.queue) == 0

# Example usage:
if __name__ == "__main__":
    cs_queue = CustomerServiceQueue()
    cs_queue.enqueue("Request 1")
    cs_queue.enqueue("Request 2")
    cs_queue.enqueue("Request 3")

    while not cs_queue.is_empty():
        request = cs_queue.dequeue()
        print(f"Handling: {request}")
```

Output:



```
PS D:\AI_ASSIT_CODING> & C:\Users\MYSELF\AppDat
Processing Request 1: Password reset
Processing Request 2: Account locked
Processing Request 3: Software installation
```

Explanation:

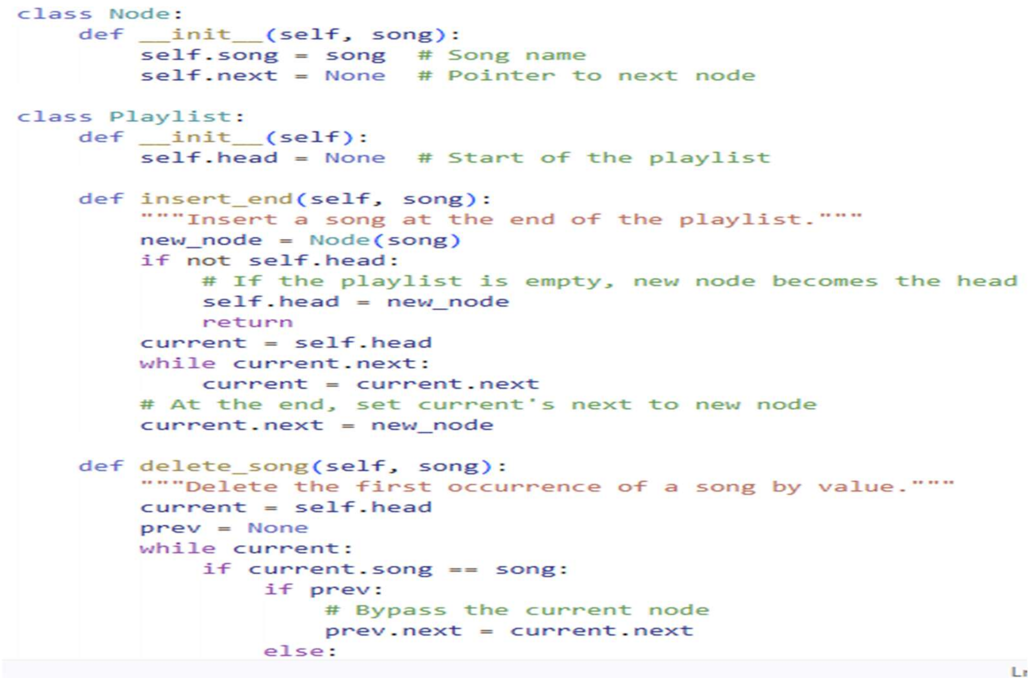
The queue made with a list takes **O(n)** time to remove an item from the front because all the other items have to shift one position forward. As the queue gets bigger, this makes it slower and less efficient. On the other hand, using collections.deque takes **O(1)** time to add or remove items from either end. This means the speed stays the same no matter how large the queue becomes.Using deque is especially important in real-world programs that need to handle many tasks quickly, like web servers or task scheduling systems, where fast and consistent performance matters.

Question 3: Singly Linked List for Dynamic Playlist Management

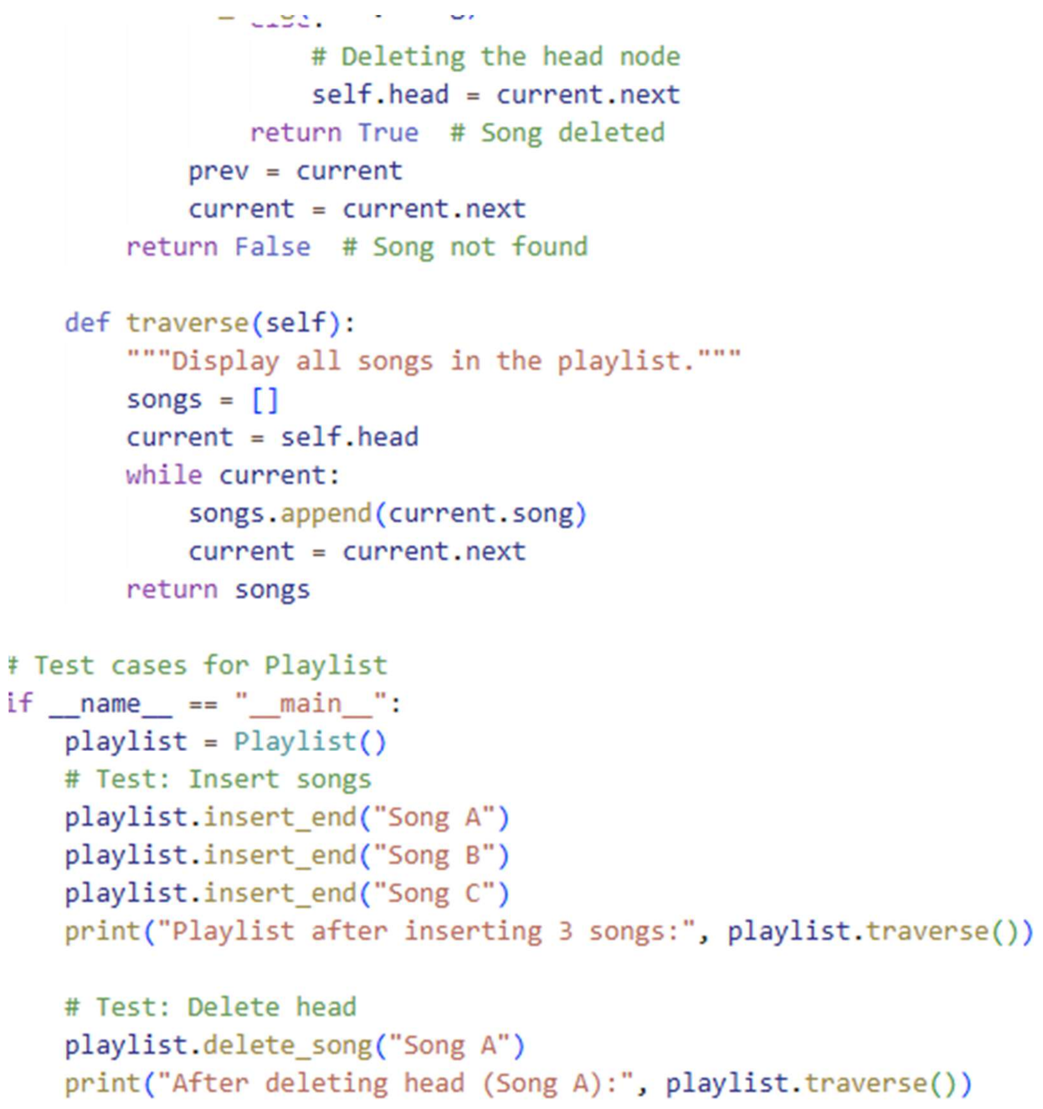
Prompt:

Write a Python program that creates a Singly Linked List for managing a dynamic playlist. The linked list should include methods to insert a song at the end, delete a song by its value, and traverse the playlist to display all songs. Add clear inline comments that explain how pointers are being updated, especially in the tricky parts of insertion and deletion. The program should also include test cases to verify that all operations work correctly, along with edge case scenarios. Make sure the final implementation is complete and fully functional.

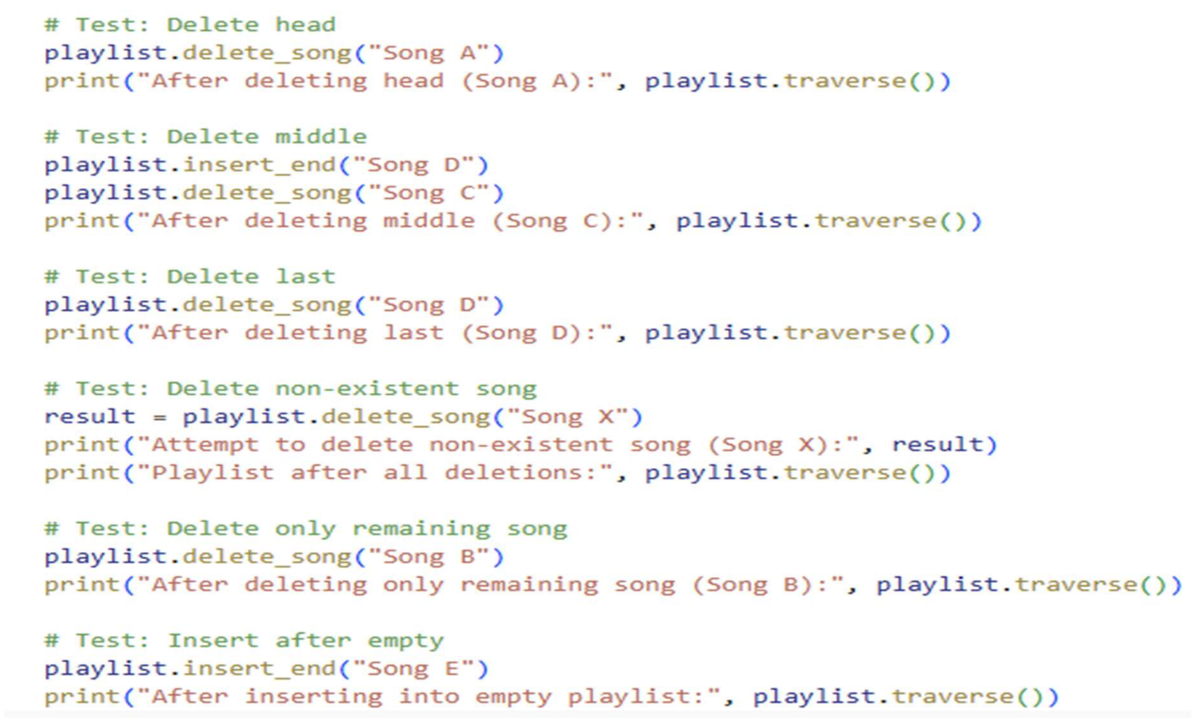
Code:



```python
class Node:
    def __init__(self, song):
        self.song = song  # Song name
        self.next = None  # Pointer to next node

class Playlist:
    def __init__(self):
        self.head = None  # Start of the playlist

    def insert_end(self, song):
        """Insert a song at the end of the playlist."""
        new_node = Node(song)
        if not self.head:
            # If the playlist is empty, new node becomes the head
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        # At the end, set current's next to new node
        current.next = new_node

    def delete_song(self, song):
        """Delete the first occurrence of a song by value."""
        current = self.head
        prev = None
        while current:
            if current.song == song:
                if prev:
                    # Bypass the current node
                    prev.next = current.next
                else:
```

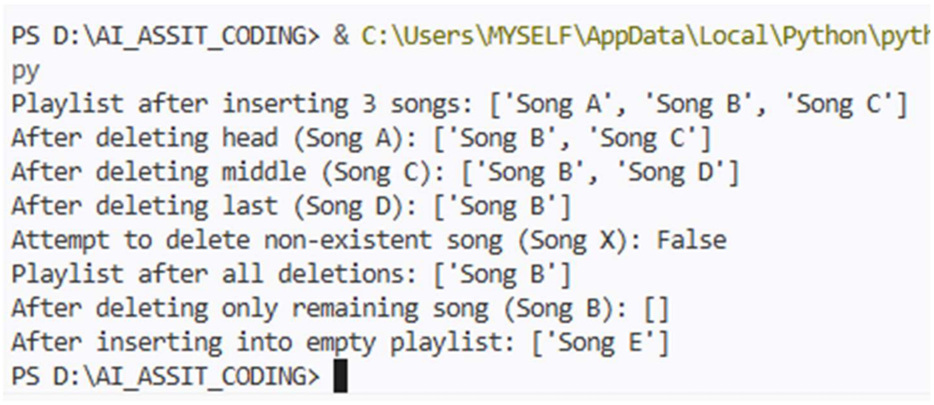


```python
                    # Deleting the head node
                    self.head = current.next
                return True  # Song deleted
            prev = current
            current = current.next
        return False  # Song not found

    def traverse(self):
        """Display all songs in the playlist."""
        songs = []
        current = self.head
        while current:
            songs.append(current.song)
            current = current.next
        return songs

# Test cases for Playlist
if __name__ == "__main__":
    playlist = Playlist()
    # Test: Insert songs
    playlist.insert_end("Song A")
    playlist.insert_end("Song B")
    playlist.insert_end("Song C")
    print("Playlist after inserting 3 songs:", playlist.traverse())

    # Test: Delete head
    playlist.delete_song("Song A")
    print("After deleting head (Song A):", playlist.traverse())
```



```python
    # Test: Delete head
    playlist.delete_song("Song A")
    print("After deleting head (Song A):", playlist.traverse())

    # Test: Delete middle
    playlist.insert_end("Song D")
    playlist.delete_song("Song C")
    print("After deleting middle (Song C):", playlist.traverse())

    # Test: Delete last
    playlist.delete_song("Song D")
    print("After deleting last (Song D):", playlist.traverse())

    # Test: Delete non-existent song
    result = playlist.delete_song("Song X")
    print("Attempt to delete non-existent song (Song X):", result)
    print("Playlist after all deletions:", playlist.traverse())

    # Test: Delete only remaining song
    playlist.delete_song("Song B")
    print("After deleting only remaining song (Song B):", playlist.traverse())

    # Test: Insert after empty
    playlist.insert_end("Song E")
    print("After inserting into empty playlist:", playlist.traverse())
```

Output:



```
PS D:\AI_ASSIT_CODING> & C:\Users\MYSELF\AppData\Local\Python\pyt
py
Playlist after inserting 3 songs: ['Song A', 'Song B', 'Song C']
After deleting head (Song A): ['Song B', 'Song C']
After deleting middle (Song C): ['Song B', 'Song D']
After deleting last (Song D): ['Song B']
Attempt to delete non-existent song (Song X): False
Playlist after all deletions: ['Song B']
After deleting only remaining song (Song B): []
After inserting into empty playlist: ['Song E']
PS D:\AI_ASSIT_CODING> █
```

Explanation:

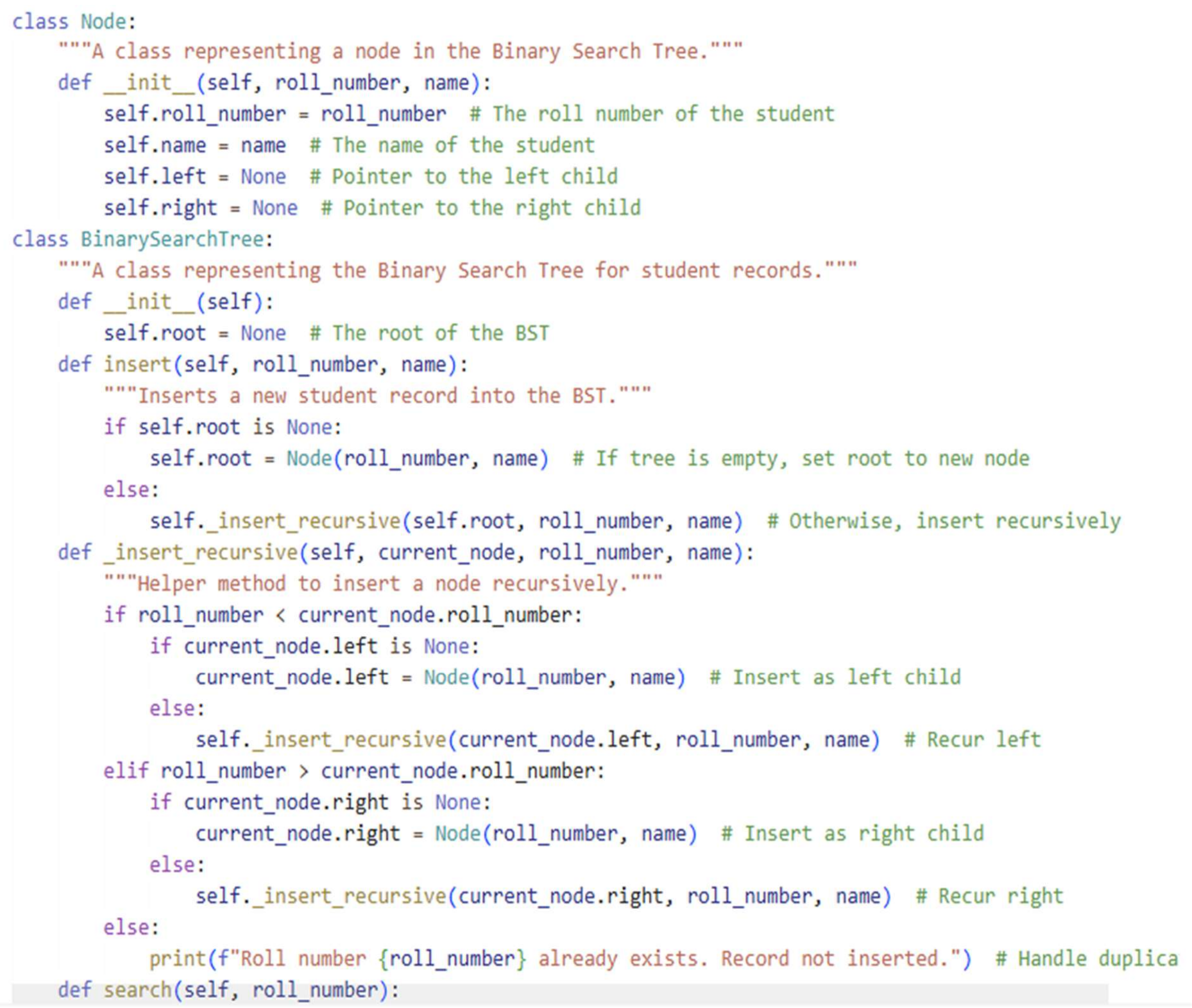
The function is_palindrome(sentence) checks if a given sentence is a palindrome by first cleaning the input. It removes all spaces and punctuation marks, and converts the string to lowercase to ensure that the palindrome check is case-insensitive and ignores non-alphanumeric characters. Then, it compares the cleaned sentence with its reverse to determine if it is a palindrome. This approach allows the function to accurately identify palindromic sentences while ignoring irrelevant characters, making it robust for various inputs.

Question 4: Binary Search Tree for Fast Record Lookup

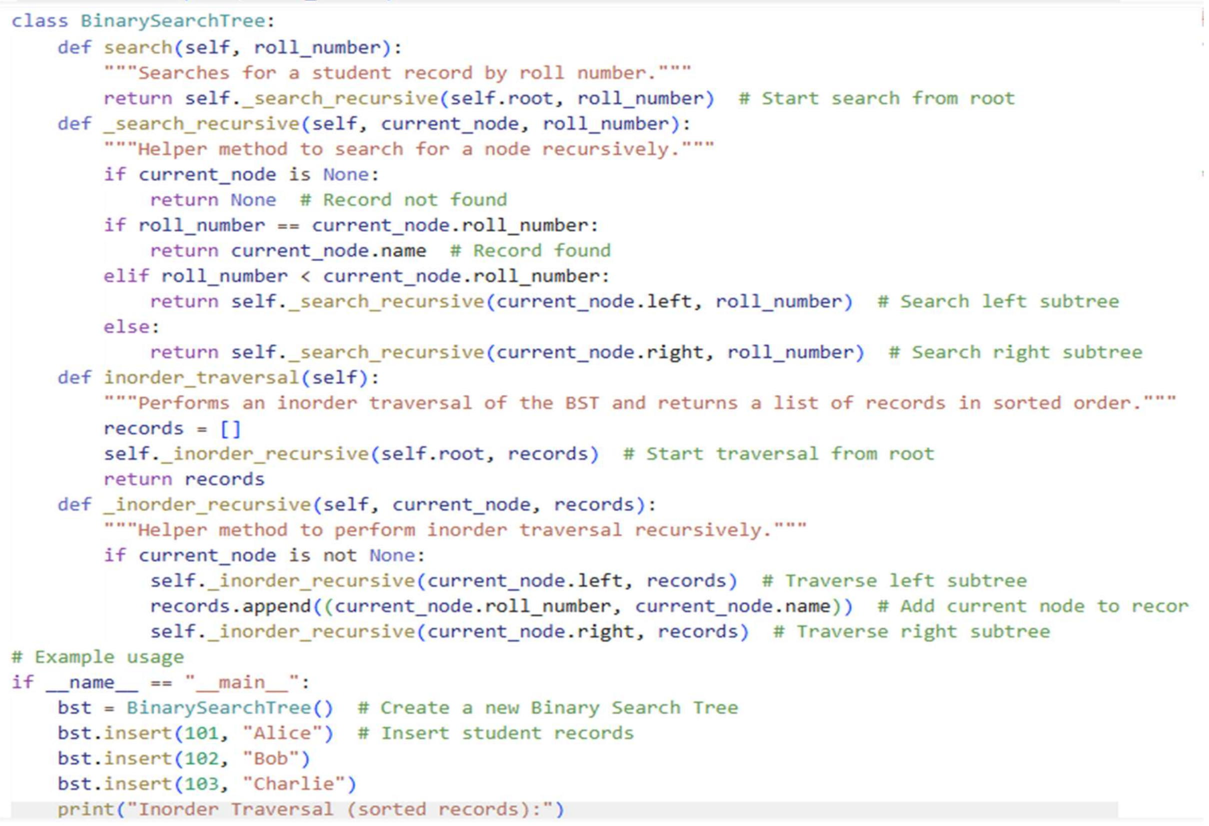
Prompt:

Write a Python program that implements a Binary Search Tree (BST) for fast record lookup. Create a student record system where records can be quickly searched using a roll number. Include functions to insert a record, search for a record, and perform an inorder traversal to display the records in sorted order. Add clear and meaningful comments throughout the code. Make sure the program is well-structured, clean, and correctly handles insert and search operations with appropriate output.

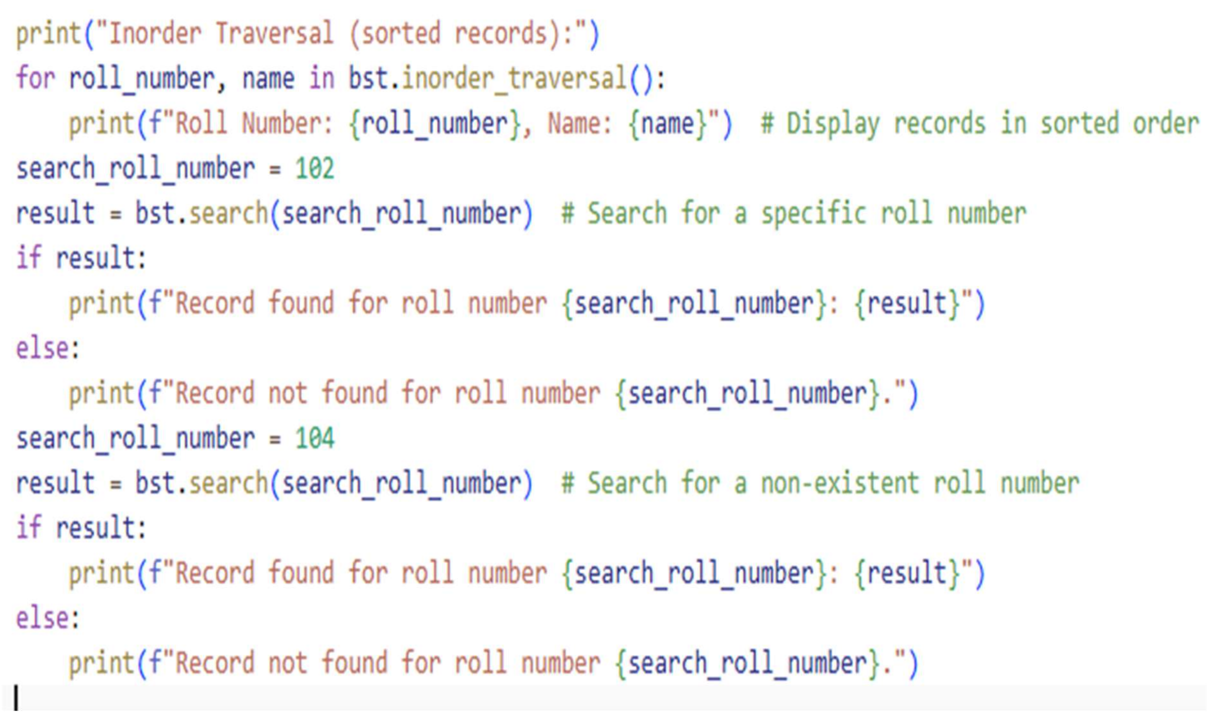
Code:



```python
class Node:
    """A class representing a node in the Binary Search Tree."""
    def __init__(self, roll_number, name):
        self.roll_number = roll_number  # The roll number of the student
        self.name = name  # The name of the student
        self.left = None  # Pointer to the left child
        self.right = None  # Pointer to the right child
class BinarySearchTree:
    """A class representing the Binary Search Tree for student records."""
    def __init__(self):
        self.root = None  # The root of the BST
    def insert(self, roll_number, name):
        """Inserts a new student record into the BST."""
        if self.root is None:
            self.root = Node(roll_number, name)  # If tree is empty, set root to new node
        else:
            self._insert_recursive(self.root, roll_number, name)  # Otherwise, insert recursively
    def _insert_recursive(self, current_node, roll_number, name):
        """Helper method to insert a node recursively."""
        if roll_number < current_node.roll_number:
            if current_node.left is None:
                current_node.left = Node(roll_number, name)  # Insert as left child
            else:
                self._insert_recursive(current_node.left, roll_number, name)  # Recur left
        elif roll_number > current_node.roll_number:
            if current_node.right is None:
                current_node.right = Node(roll_number, name)  # Insert as right child
            else:
                self._insert_recursive(current_node.right, roll_number, name)  # Recur right
        else:
            print(f"Roll number {roll_number} already exists. Record not inserted.")  # Handle duplica
    def search(self, roll_number):
```

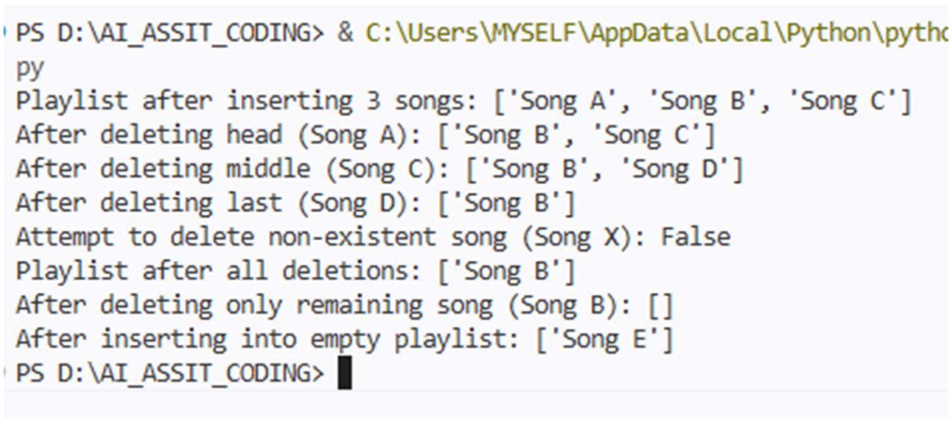


```python
class BinarySearchTree:
    def search(self, roll_number):
        """Searches for a student record by roll number."""
        return self._search_recursive(self.root, roll_number)  # Start search from root
    def _search_recursive(self, current_node, roll_number):
        """Helper method to search for a node recursively."""
        if current_node is None:
            return None  # Record not found
        if roll_number == current_node.roll_number:
            return current_node.name  # Record found
        elif roll_number < current_node.roll_number:
            return self._search_recursive(current_node.left, roll_number)  # Search left subtree
        else:
            return self._search_recursive(current_node.right, roll_number)  # Search right subtree
    def inorder_traversal(self):
        """Performs an inorder traversal of the BST and returns a list of records in sorted order."""
        records = []
        self._inorder_recursive(self.root, records)  # Start traversal from root
        return records
    def _inorder_recursive(self, current_node, records):
        """Helper method to perform inorder traversal recursively."""
        if current_node is not None:
            self._inorder_recursive(current_node.left, records)  # Traverse left subtree
            records.append((current_node.roll_number, current_node.name))  # Add current node to recor
            self._inorder_recursive(current_node.right, records)  # Traverse right subtree
# Example usage
if __name__ == "__main__":
    bst = BinarySearchTree()  # Create a new Binary Search Tree
    bst.insert(101, "Alice")  # Insert student records
    bst.insert(102, "Bob")
    bst.insert(103, "Charlie")
    print("Inorder Traversal (sorted records):")
```



```
print("Inorder Traversal (sorted records):")
for roll_number, name in bst.inorder_traversal():
    print(f"Roll Number: {roll_number}, Name: {name}")  # Display records in sorted order
search_roll_number = 102
result = bst.search(search_roll_number)  # Search for a specific roll number
if result:
    print(f"Record found for roll number {search_roll_number}: {result}")
else:
    print(f"Record not found for roll number {search_roll_number}.")
search_roll_number = 104
result = bst.search(search_roll_number)  # Search for a non-existent roll number
if result:
    print(f"Record found for roll number {search_roll_number}: {result}")
else:
    print(f"Record not found for roll number {search_roll_number}.")
```

Output:



```
PS D:\AI_ASSIT_CODING> & C:\Users\MYSELF\AppData\Local\Python\pytho
py
Playlist after inserting 3 songs: ['Song A', 'Song B', 'Song C']
After deleting head (Song A): ['Song B', 'Song C']
After deleting middle (Song C): ['Song B', 'Song D']
After deleting last (Song D): ['Song B']
Attempt to delete non-existent song (Song X): False
Playlist after all deletions: ['Song B']
After deleting only remaining song (Song B): []
After inserting into empty playlist: ['Song E']
PS D:\AI_ASSIT_CODING>
```

Explanation:

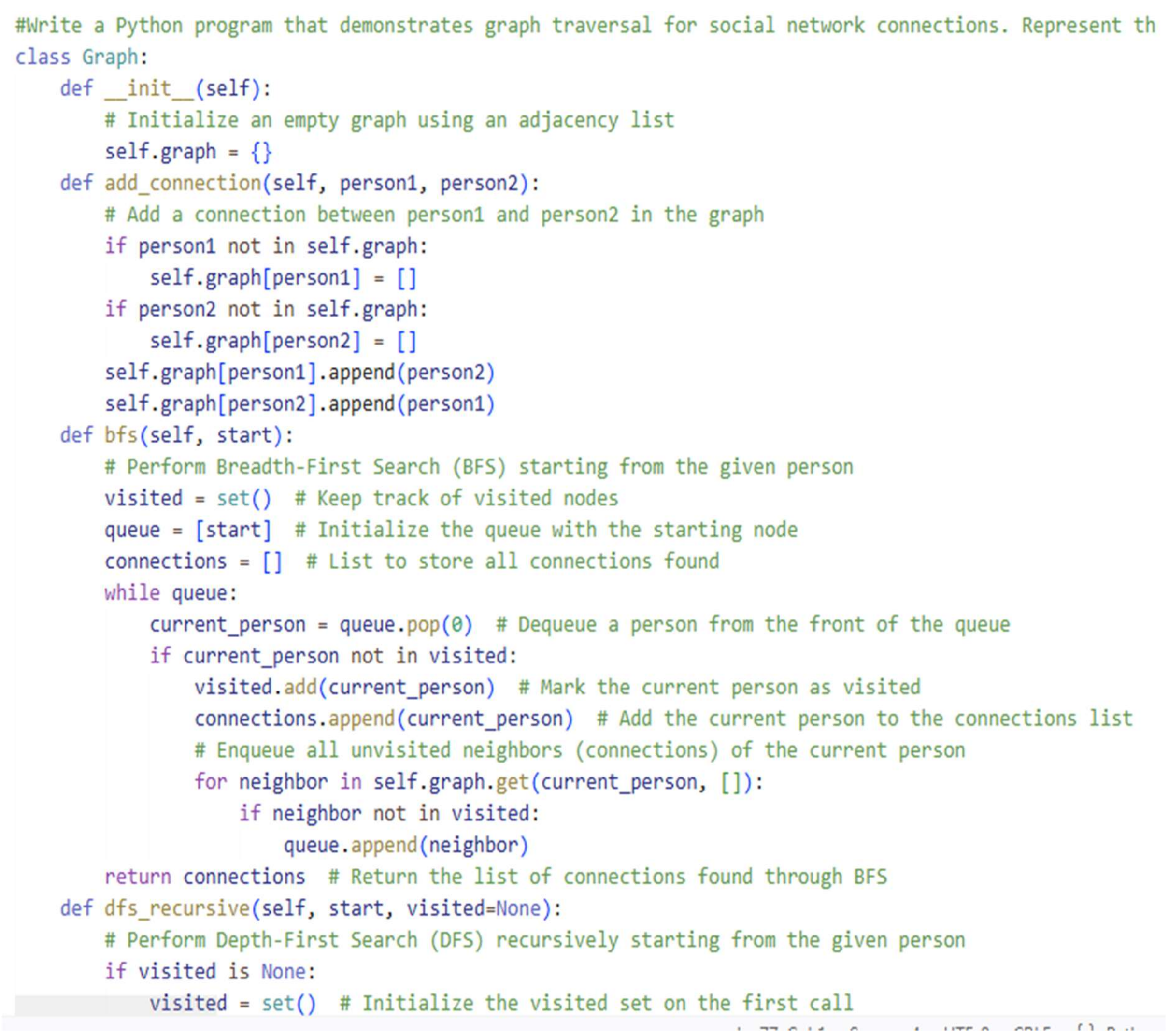
A Binary Search Tree (BST) makes searching faster than a linear search by arranging data in a structured, hierarchical way. In a BST, each node can have up to two children. The left child holds values smaller than the parent node, and the right child holds values greater than the parent node.Because of this structure, a balanced BST can search in **O(log n)** time, since each comparison reduces the search space by half. On the other hand, a linear search takes **O(n)** time because it may need to check every element one by one.The best performance of a BST happens when the tree is perfectly balanced. However, if the tree becomes skewed (for example, when all nodes are placed on one side), its performance drops to **O(n)**, which is the same as a linear search.

Question 5: Graph Traversal for Social Network Connections

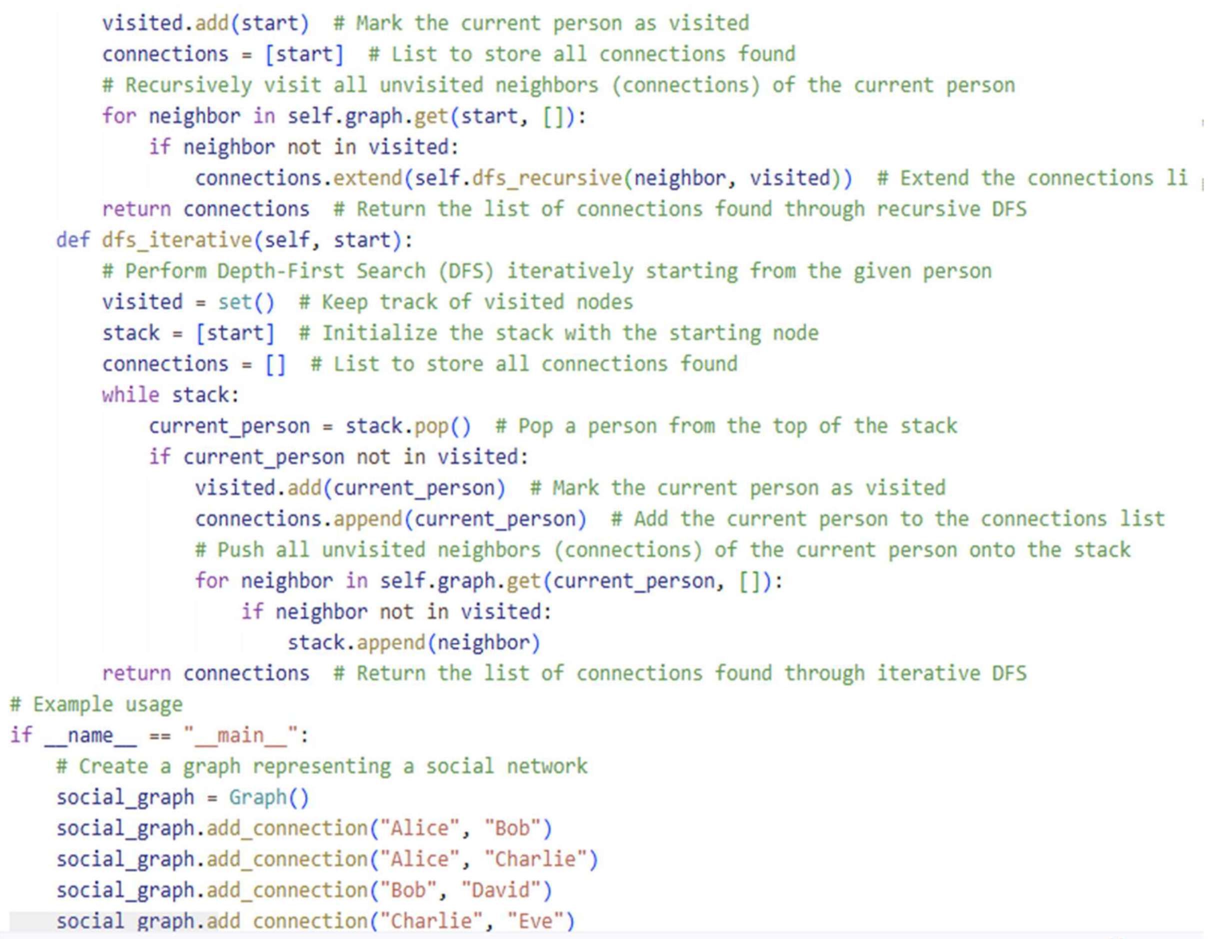
Prompt:

Write a Python program that demonstrates graph traversal for social network connections. Represent the social network as a graph where each node represents a person and each edge represents a connection between people. Implement both Depth-First Search (DFS) and Breadth-First Search (BFS) to explore the graph and find all connections for a given person. Include clear inline comments explaining each step of the traversal process. Also compare the recursive and iterative approaches to DFS. Ensure the implementation is clean, well-structured, efficient, and easy to understand.

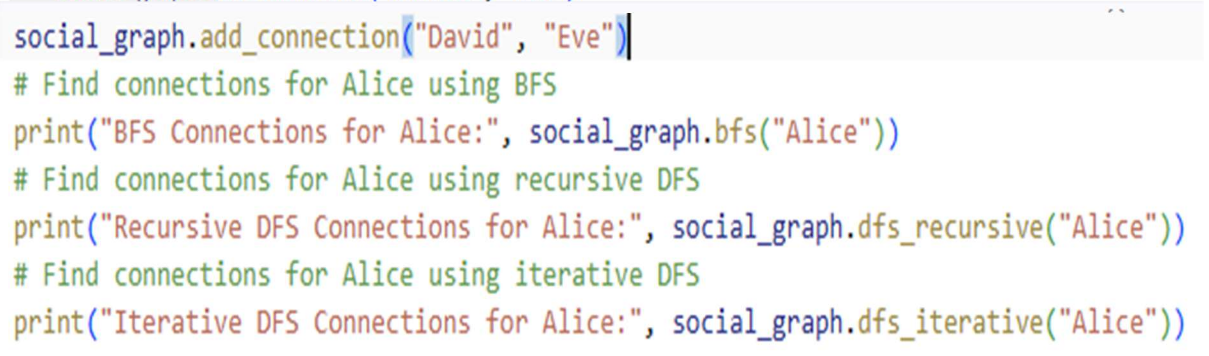
Code:



```python
#Write a Python program that demonstrates graph traversal for social network connections. Represent th
class Graph:
    def __init__(self):
        # Initialize an empty graph using an adjacency list
        self.graph = {}
    def add_connection(self, person1, person2):
        # Add a connection between person1 and person2 in the graph
        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 bfs(self, start):
        # Perform Breadth-First Search (BFS) starting from the given person
        visited = set()  # Keep track of visited nodes
        queue = [start]  # Initialize the queue with the starting node
        connections = []  # List to store all connections found
        while queue:
            current_person = queue.pop(0)  # Dequeue a person from the front of the queue
            if current_person not in visited:
                visited.add(current_person)  # Mark the current person as visited
                connections.append(current_person)  # Add the current person to the connections list
                # Enqueue all unvisited neighbors (connections) of the current person
                for neighbor in self.graph.get(current_person, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
        return connections  # Return the list of connections found through BFS
    def dfs_recursive(self, start, visited=None):
        # Perform Depth-First Search (DFS) recursively starting from the given person
        if visited is None:
            visited = set()  # Initialize the visited set on the first call
```

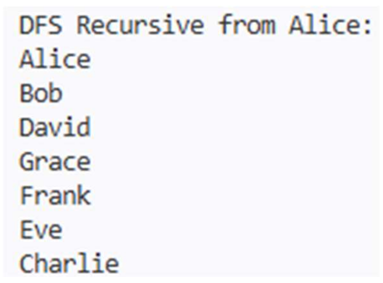


```python
        visited.add(start)  # Mark the current person as visited
        connections = [start]  # List to store all connections found
        # Recursively visit all unvisited neighbors (connections) of the current person
        for neighbor in self.graph.get(start, []):
            if neighbor not in visited:
                connections.extend(self.dfs_recursive(neighbor, visited))  # Extend the connections li
        return connections  # Return the list of connections found through recursive DFS
    def dfs_iterative(self, start):
        # Perform Depth-First Search (DFS) iteratively starting from the given person
        visited = set()  # Keep track of visited nodes
        stack = [start]  # Initialize the stack with the starting node
        connections = []  # List to store all connections found
        while stack:
            current_person = stack.pop()  # Pop a person from the top of the stack
            if current_person not in visited:
                visited.add(current_person)  # Mark the current person as visited
                connections.append(current_person)  # Add the current person to the connections list
                # Push all unvisited neighbors (connections) of the current person onto the stack
                for neighbor in self.graph.get(current_person, []):
                    if neighbor not in visited:
                        stack.append(neighbor)
        return connections  # Return the list of connections found through iterative DFS
# Example usage
if __name__ == "__main__":
    # Create a graph representing a social network
    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")
```

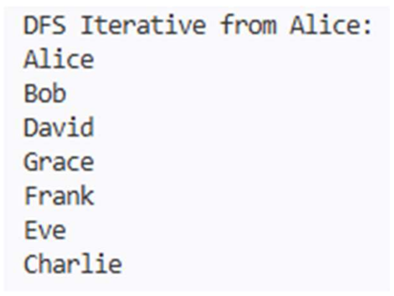


```python
social_graph.add_connection("David", "Eve")
# Find connections for Alice using BFS
print("BFS Connections for Alice:", social_graph.bfs("Alice"))
# Find connections for Alice using recursive DFS
print("Recursive DFS Connections for Alice:", social_graph.dfs_recursive("Alice"))
# Find connections for Alice using iterative DFS
print("Iterative DFS Connections for Alice:", social_graph.dfs_iterative("Alice"))
```

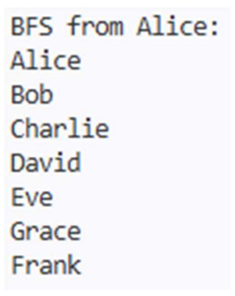
Output:



```
DFS Recursive from Alice:
Alice
Bob
David
Grace
Frank
Eve
Charlie
```



```
DFS Iterative from Alice:
Alice
Bob
David
Grace
Frank
Eve
Charlie
```



```
BFS from Alice:
Alice
Bob
Charlie
David
Eve
Grace
Frank
```

Explanation:

Depth-First Search (DFS) goes as deep as possible along one branch before coming back and trying other paths. This makes it useful for problems like maze solving or pathfinding, where you need to explore every possible route.

Breadth-First Search (BFS), in contrast, explores all neighboring nodes at the current level before moving to the next level. Because of this step-by-step approach, BFS is better for finding the shortest path in unweighted graphs or for level-order traversal in trees.

The recursive version of DFS is usually simpler and easier to read, while the iterative version can use memory more efficiently, especially when working with very deep graphs.