AI ASSISTED CODING

NAME: E.HAMSITHA

ENROLL NUMBER: 2403A52361

BATCH NUMBER :13

Lab assignment-11.4

Task 1: Implementing a Stack (LIFO)

- **Task**: Use AI to help implement a **Stack** class in Python with the following operations: push(), pop(), peek(), and is_empty().
- **Instructions**:
    - Ask AI to generate code skeleton with docstrings.
    - Test stack operations using sample data.
    - Request AI to suggest optimizations or alternative implementations (e.g., using collections.deque).

Code:

[1]
✓ 0s

```python
class Stack:
    """
    A simple Stack implementation using a Python list.
    """

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

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

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

    def pop(self):
        """
        Removes and returns the top item from the stack.

        Returns:
            The top item from the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if not self.is_empty():
            return self._items.pop()
        else:
            raise IndexError("pop from empty stack")
```

```python
    def peek(self):
        """
        Returns the top item from the stack without removing it.

        Returns:
            The top item from the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if not self.is_empty():
            return self._items[-1]
        else:
            raise IndexError("peek from empty stack")

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

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

[2]
✓ 0s

```python
# 1. Create an instance of the Stack class.
my_stack = Stack()

# 2. Push a few sample data elements onto the stack using the push() method.
my_stack.push(10)
my_stack.push(20)
my_stack.push(30)

# 3. Check if the stack is empty using is_empty().
print(f"Is the stack empty? {my_stack.is_empty()}")

# 4. Peek at the top element of the stack using peek() and print the result.
try:
    print(f"Top element (peek): {my_stack.peek()}")
except IndexError as e:
    print(f"Error peeking: {e}")

# 5. Pop an element from the stack using pop() and print the popped element.
try:
    popped_item = my_stack.pop()
    print(f"Popped element: {popped_item}")
except IndexError as e:
    print(f"Error popping: {e}")

# 6. Peek at the top element again to see if it has changed.
try:
    print(f"Top element after pop (peek): {my_stack.peek()}")
except IndexError as e:
    print(f"Error peeking after pop: {e}")

# 7. Pop the remaining elements from the stack.
while not my_stack.is_empty():
    try:
        print(f"Popping remaining: {my_stack.pop()}")
```

[2]
✓ 0s

```
while not my_stack.is_empty():
    try:
        print(f"Popping remaining: {my_stack.pop()}")
    except IndexError as e:
        print(f"Error popping remaining: {e}")

# 8. Check if the stack is empty again.
print(f"Is the stack empty now? {my_stack.is_empty()}")

# 9. Attempt to pop or peek from the empty stack to observe the error handling.
try:
    print(f"Attempting to peek from empty stack: {my_stack.peek()}")
except IndexError as e:
    print(f"Error as expected when peeking from empty stack: {e}")

try:
    print(f"Attempting to pop from empty stack: {my_stack.pop()}")
except IndexError as e:
    print(f"Error as expected when popping from empty stack: {e}")
```

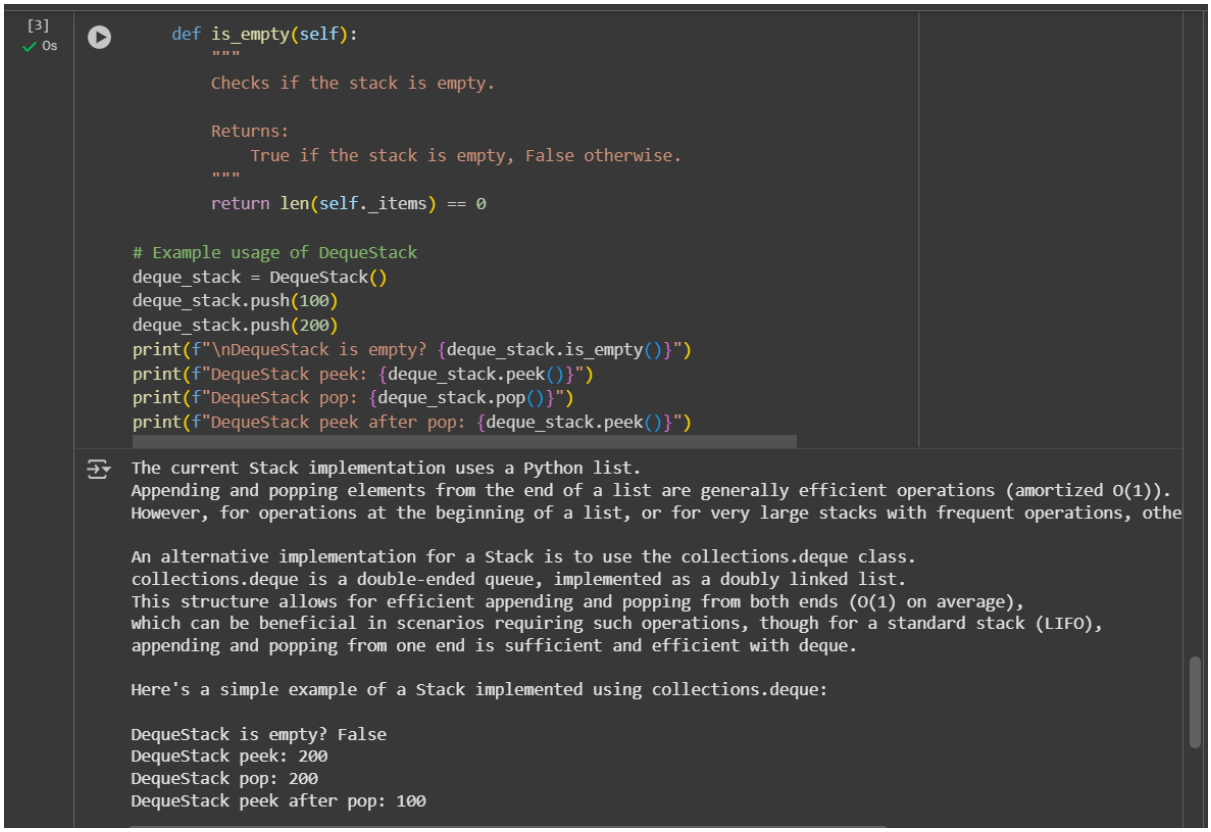
```
Is the stack empty? False
Top element (peek): 30
Popped element: 30
Top element after pop (peek): 20
Popping remaining: 20
Popping remaining: 10
Is the stack empty now? True
Error as expected when peeking from empty stack: peek from empty stack
Error as expected when popping from empty stack: pop from empty stack
```

[3]
✓ 0s

```
import collections

# 1. Explain that while the current list-based implementation works, appending and popping from the end of
print("The current Stack implementation uses a Python list.")
print("Appending and popping elements from the end of a list are generally efficient operations (amortized
print("However, for operations at the beginning of a list, or for very large stacks with frequent operatio

# 2. Suggest using collections.deque as an alternative for potentially better performance.
print("\nAn alternative implementation for a Stack is to use the collections.deque class.")

# 3. Briefly explain why collections.deque can be more efficient for append/pop operations.
print("collections.deque is a double-ended queue, implemented as a doubly linked list.")
print("This structure allows for efficient appending and popping from both ends (O(1) on average),")
print("which can be beneficial in scenarios requiring such operations, though for a standard stack (LIFO),
print("appending and popping from one end is sufficient and efficient with deque.")

# 4. Provide a simple code example showing how to implement a stack using collections.deque.
print("\nHere's a simple example of a Stack implemented using collections.deque:")

class DequeStack:
    """
    A Stack implementation using collections.deque.
    """
    def __init__(self):
        """
        Initializes an empty stack using a deque.
        """
        self._items = collections.deque()

    def push(self, item):
        """
        Adds an item to the top of the stack.
```

[3]
✓ 0s

```
        Args:
            item: The item to be added to the stack.
        """
        self._items.append(item) # Append to the right side

    def pop(self):
        """
        Removes and returns the top item from the stack.

        Returns:
            The top item from the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if not self.is_empty():
            return self._items.pop() # Pop from the right side
        else:
            raise IndexError("pop from empty stack")

    def peek(self):
        """
        Returns the top item from the stack without removing it.

        Returns:
            The top item from the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if not self.is_empty():
            return self._items[-1] # Peek at the rightmost element
        else:
            raise IndexError("peek from empty stack")
```



Summary:

Data Analysis Key Findings

* A Python class `Stack` was successfully implemented using a standard Python list (`self._items`) to store elements.

* The `push()` method appends elements to the end of the list, `pop()` removes and returns the last element, and `peek()` returns the last element without removing it.

* The `is_empty()` method correctly checks if the internal list is empty.

* Error handling for `pop()` and `peek()` on an empty stack was implemented using `IndexError`.

* Testing confirmed that `push()`, `pop()`, `peek()`, and `is_empty()` methods function as expected, including the error handling for empty stacks.

* An alternative `DequeStack` class using `collections.deque` was presented, demonstrating an alternative implementation that also provides efficient O(1) operations for stack functionality.

Insights or Next Steps

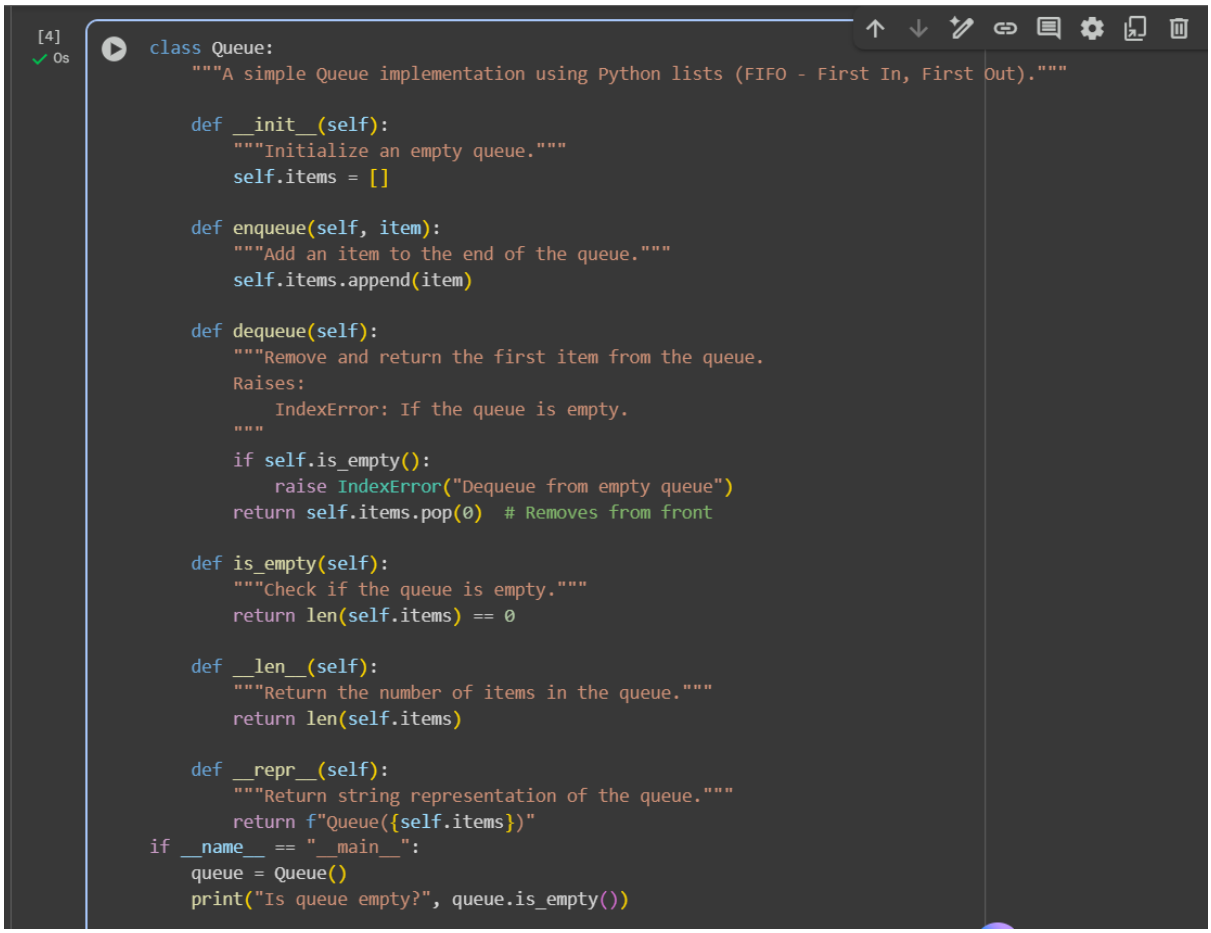
* The list-based stack implementation is generally efficient for typical stack operations.

* For scenarios requiring high performance with very large stacks or potential extensions involving operations on both ends, `collections.deque` offers a potentially more performant alternative.

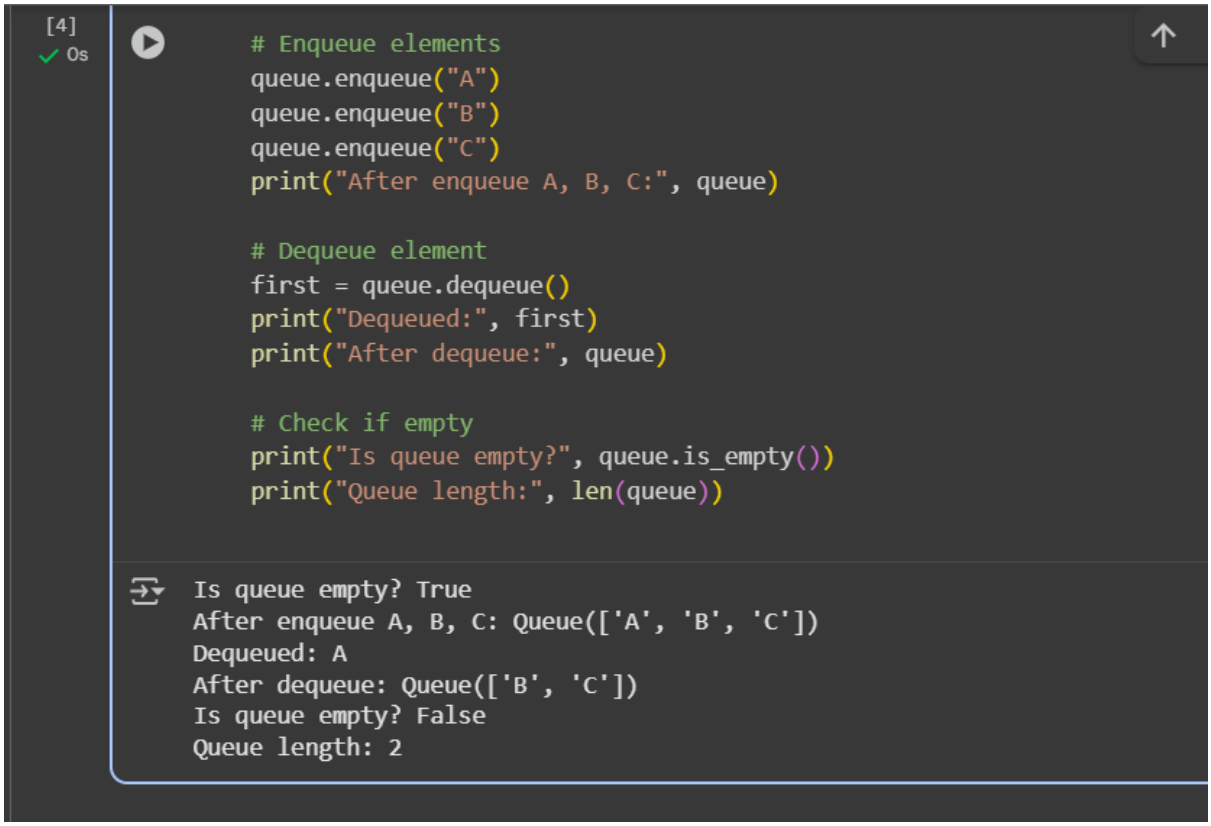
Task 2: Queue Implementation with Performance Review

- Task: Implement a Queue with enqueue(), dequeue(), and is_empty() methods.
- Instructions:
  - First, implement using Python lists.
  - Then, ask AI to review performance and suggest a more efficient implementation (using collections.deque).

code:



```
[4]
✓ 0s

class Queue:
    """A simple Queue implementation using Python lists (FIFO - First In, First Out)."""

    def __init__(self):
        """Initialize an empty queue."""
        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 first item from the queue.
        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)  # Removes from front

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

    def __len__(self):
        """Return the number of items in the queue."""
        return len(self.items)

    def __repr__(self):
        """Return string representation of the queue."""
        return f"Queue({self.items})"
if __name__ == "__main__":
    queue = Queue()
    print("Is queue empty?", queue.is_empty())
```



Task 3: Singly Linked List with Traversal

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

o Start with a simple class-based implementation (Node, LinkedList).

o Use AI to generate inline comments explaining pointer updates (which are non-trivial).

o Ask AI to suggest test cases to validate all operations.

Code:

[6]
✓ 0s

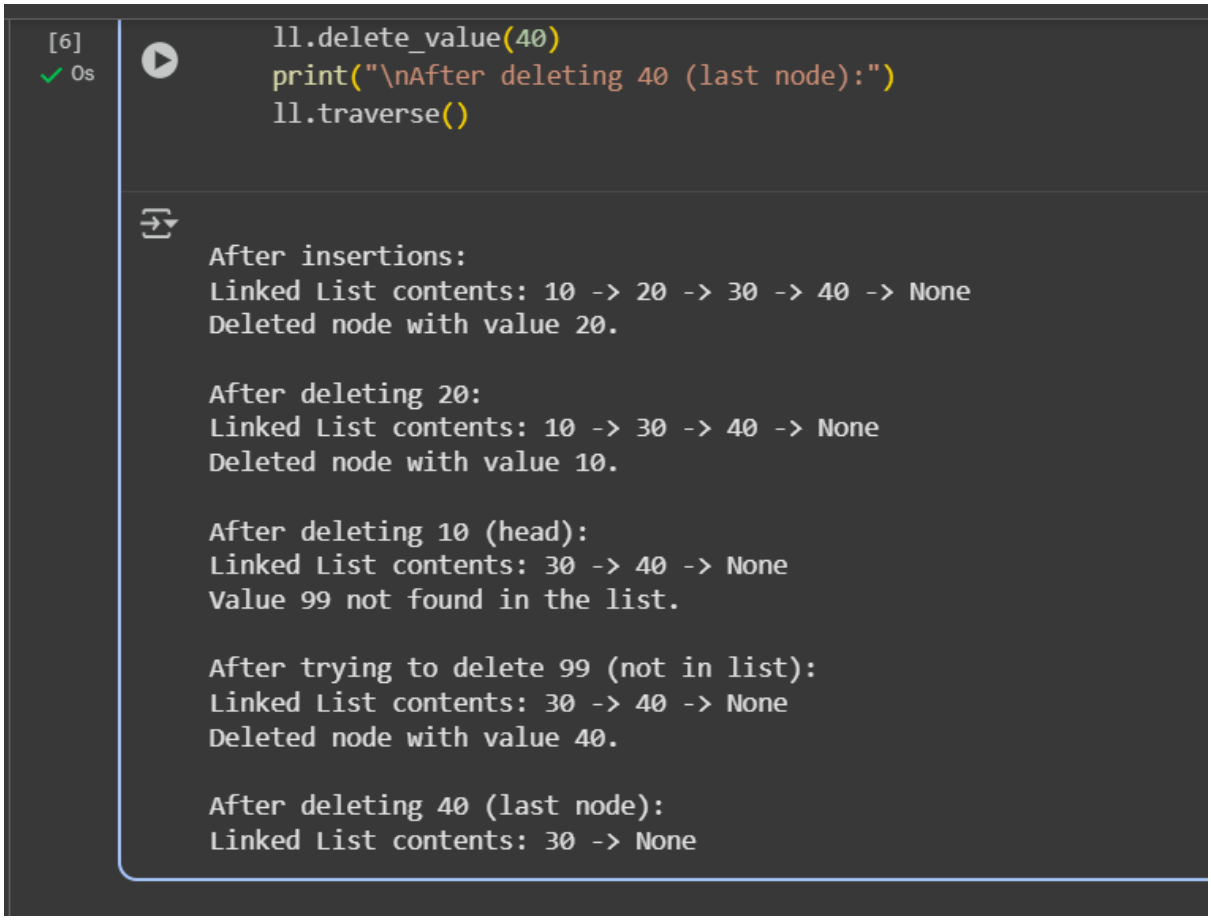
```python
class Node:
    """A Node in a singly linked list."""
    def __init__(self, data):
        self.data = data      # Store the data
        self.next = None      # Pointer to the next node (default: None)


class LinkedList:
    """A simple Singly Linked List implementation."""

    def __init__(self):
        self.head = None  # Initially, the list is empty (no head node)

    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 the list is empty, the new node becomes the head
        if self.head is None:
            self.head = new_node
            return

        # Otherwise, traverse to the last node
        current = self.head
        while current.next:
            current = current.next

        # Update the last node's next pointer to point to the new node
        current.next = new_node  # <-- This "links" the new node at the end

    def delete_value(self, value):
        """Delete the first occurrence of a node with the given value."""
        current = self.head
        previous = None
```

[6]
✓ 0s

```
        # Traverse the list to find the node to delete
        while current and current.data != value:
            previous = current          # Move 'previous' forward
            current = current.next      # Move 'current' forward

        # Value not found
        if current is None:
            print(f"Value {value} not found in the list.")
            return

        # Case 1: Deleting the head node
        if previous is None:
            # Move head pointer to the next node
            self.head = current.next
        else:
            # Skip the current node by linking previous node to next
            previous.next = current.next  # <-- Bypass the node to delete

        print(f"Deleted node with value {value}.")

    def traverse(self):
        """Traverse the linked list and print all node values."""
        current = self.head
        if not current:
            print("The list is empty.")
            return

        print("Linked List contents:", end=" ")
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")  # End of list indicator

    def __repr__(self):
        """Return string representation of the list."""
```

[6]
✓ 0s

```
            nodes = []
            current = self.head
            while current:
                nodes.append(str(current.data))
                current = current.next
            return " -> ".join(nodes) + " -> None"
    if __name__ == "__main__":
        ll = LinkedList()

        # Insert elements
        ll.insert_at_end(10)
        ll.insert_at_end(20)
        ll.insert_at_end(30)
        ll.insert_at_end(40)

        print("\nAfter insertions:")
        ll.traverse()

        # Delete a middle element
        ll.delete_value(20)
        print("\nAfter deleting 20:")
        ll.traverse()

        # Delete the head element
        ll.delete_value(10)
        print("\nAfter deleting 10 (head):")
        ll.traverse()

        # Delete a non-existent element
        ll.delete_value(99)
        print("\nAfter trying to delete 99 (not in list):")
        ll.traverse()

        # Delete the last element
        ll.delete_value(40)
        print("\nAfter deleting 40 (last node):")
```



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.
- **Expected Output**:

- [1] ✓ 0s

```python
# Binary Search Tree Implementation in Python

class Node:
    """A Node in the Binary Search Tree."""
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


class BST:
    """Binary Search Tree with insert, search, and inorder traversal methods."""

    def __init__(self):
        self.root = None

    def insert(self, key):
        """
        Insert a new key into the BST.
        If the tree is empty, the new node becomes the root.
        Otherwise, it recursively finds the correct position.
        """
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert(self.root, key)

    def _insert(self, current_node, key):
        if key < current_node.key:
            if current_node.left is None:
                current_node.left = Node(key)
            else:
                self._insert(current_node.left, key)
```

- [1] ✓ 0s

```python
                self._insert(current_node.left, key)
        elif key > current_node.key:
            if current_node.right is None:
                current_node.right = Node(key)
            else:
                self._insert(current_node.right, key)
        # Duplicate keys are ignored

    def search(self, key):
        """
        Search for a key in the BST.
        Returns True if found, else False.
        """
        return self._search(self.root, key)

    def _search(self, current_node, key):
        if current_node is None:
            return False
        if key == current_node.key:
            return True
        elif key < current_node.key:
            return self._search(current_node.left, key)
        else:
            return self._search(current_node.right, key)

    def inorder_traversal(self):
        """
        Perform an inorder traversal of the BST.
        Returns a sorted list of all keys.
        """
        elements = []
        self._inorder_traversal(self.root, elements)
        return elements
```

```
[1]
✓ 0s
```

```python
    def _inorder_traversal(self, current_node, elements):
        if current_node:
            self._inorder_traversal(current_node.left, elements)
            elements.append(current_node.key)
            self._inorder_traversal(current_node.right, elements)


# ------------------ TESTING ------------------

if __name__ == "__main__":
    bst = BST()
    numbers = [50, 30, 70, 20, 40, 60, 80]

    for num in numbers:
        bst.insert(num)

    print("Inorder Traversal (Sorted Elements):", bst.inorder_traversal())

    # Test search for present and absent elements
    test_values = [40, 25, 70, 90]
    for val in test_values:
        result = bst.search(val)
        print(f"Search for {val}: {'Found ✅' if result else 'Not Found ❌'}")
```

```
Inorder Traversal (Sorted Elements): [20, 30, 40, 50, 60, 70, 80]
Search for 40: Found ✅
Search for 25: Not Found ❌
Search for 70: Found ✅
Search for 90: Not Found ❌
```

Task 5: Graph Representation and BFS/DFS Traversal

- **Task**: Implement a **Graph** using an adjacency list, with traversal methods BFS() and DFS().
- **Instructions**:
  - Start with an adjacency list dictionary.
  - Ask AI to generate BFS and DFS implementations with inline comments.
  - Compare recursive vs iterative DFS if suggested by AI.

[2]
✓ 0s

```python
# Graph Representation and Traversal in Python

from collections import deque

class Graph:
    """Graph represented using an adjacency list."""

    def __init__(self):
        # adjacency list as a dictionary {node: [neighbors]}
        self.adj_list = {}

    def add_edge(self, u, v):
        """
        Add an edge between node u and v.
        For an undirected graph, add both directions.
        """
        if u not in self.adj_list:
            self.adj_list[u] = []
        if v not in self.adj_list:
            self.adj_list[v] = []

        self.adj_list[u].append(v)
        self.adj_list[v].append(u)

    def bfs(self, start):
        """
        Breadth-First Search (BFS) traversal starting from 'start' node.
        Uses a queue to visit nodes level by level.
        """
        visited = set()
        queue = deque([start])
        traversal_order = []
```

[2]
✓ 0s

```python
        while queue:
            vertex = queue.popleft()  # remove from front of queue
            if vertex not in visited:
                visited.add(vertex)
                traversal_order.append(vertex)
                # Add all unvisited neighbors to the queue
                for neighbor in self.adj_list[vertex]:
                    if neighbor not in visited:
                        queue.append(neighbor)
        return traversal_order

    def dfs_recursive(self, start, visited=None):
        """
        Depth-First Search (DFS) traversal using recursion.
        Visits nodes by going as deep as possible along each branch.
        """
        if visited is None:
            visited = set()
        visited.add(start)
        traversal_order = [start]

        for neighbor in self.adj_list[start]:
            if neighbor not in visited:
                traversal_order.extend(self.dfs_recursive(neighbor, visited))
        return traversal_order

    def dfs_iterative(self, start):
        """
        Depth-First Search (DFS) traversal using an explicit stack.
        Iterative version avoids recursion stack overflow for large graphs.
        """
        visited = set()
        stack = [start]
        traversal_order = []
```

[2]
✓ 0s

```
        traversal_order = []

        while stack:
            vertex = stack.pop()  # remove last element (LIFO)
            if vertex not in visited:
                visited.add(vertex)
                traversal_order.append(vertex)
                # Add neighbors in reverse order for consistent output
                for neighbor in reversed(self.adj_list[vertex]):
                    if neighbor not in visited:
                        stack.append(neighbor)
        return traversal_order


# ------------------- TESTING -------------------

if __name__ == "__main__":
    # Create a sample graph
    g = Graph()
    edges = [
        ('A', 'B'), ('A', 'C'),
        ('B', 'D'), ('B', 'E'),
        ('C', 'F'), ('E', 'F')
    ]

    for u, v in edges:
        g.add_edge(u, v)

    print("Adjacency List Representation:")
    for node, neighbors in g.adj_list.items():
        print(f"{node}: {neighbors}")

    # BFS Traversal
    print("\nBFS Traversal starting from A:", g.bfs('A'))
```

[2]
✓ 0s

```
    # BFS Traversal
    print("\nBFS Traversal starting from A:", g.bfs('A'))

    # DFS (Recursive)
    print("DFS (Recursive) starting from A:", g.dfs_recursive('A'))

    # DFS (Iterative)
    print("DFS (Iterative) starting from A:", g.dfs_iterative('A'))

    # Comparison
    print("\n✅ Both DFS methods visit all nodes, but order may differ slightly due to recursion vs st
```

```
Adjacency List Representation:
A: ['B', 'C']
B: ['A', 'D', 'E']
C: ['A', 'F']
D: ['B']
E: ['B', 'F']
F: ['C', 'E']

BFS Traversal starting from A: ['A', 'B', 'C', 'D', 'E', 'F']
DFS (Recursive) starting from A: ['A', 'B', 'D', 'E', 'F', 'C']
DFS (Iterative) starting from A: ['A', 'B', 'D', 'E', 'F', 'C']

✅ Both DFS methods visit all nodes, but order may differ slightly due to recursion vs stack behavior.
```