SR UNIVERSITY
AI ASSISTED CODING

Name:B.Hasini
2503A51L13

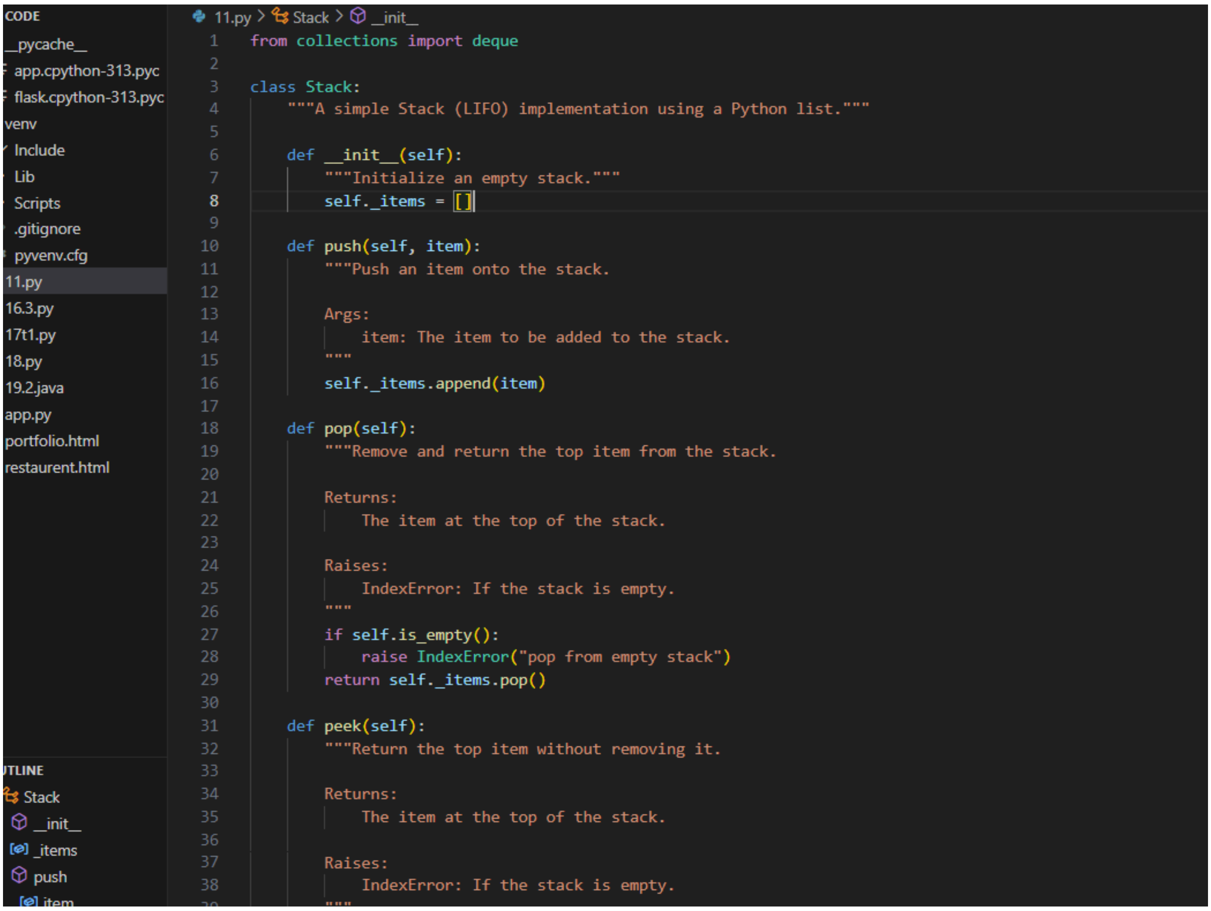
Lab 11 – Data Structures with AI: Implementing Fundamental Structures
Lab Objectives

- Use AI to assist in designing and implementing fundamental data structures in Python.
- Learn how to prompt AI for structure creation, optimization, and documentation.
- Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, Graphs, and Hash Tables.
- Enhance code quality with AI-generated comments and performance suggestions.

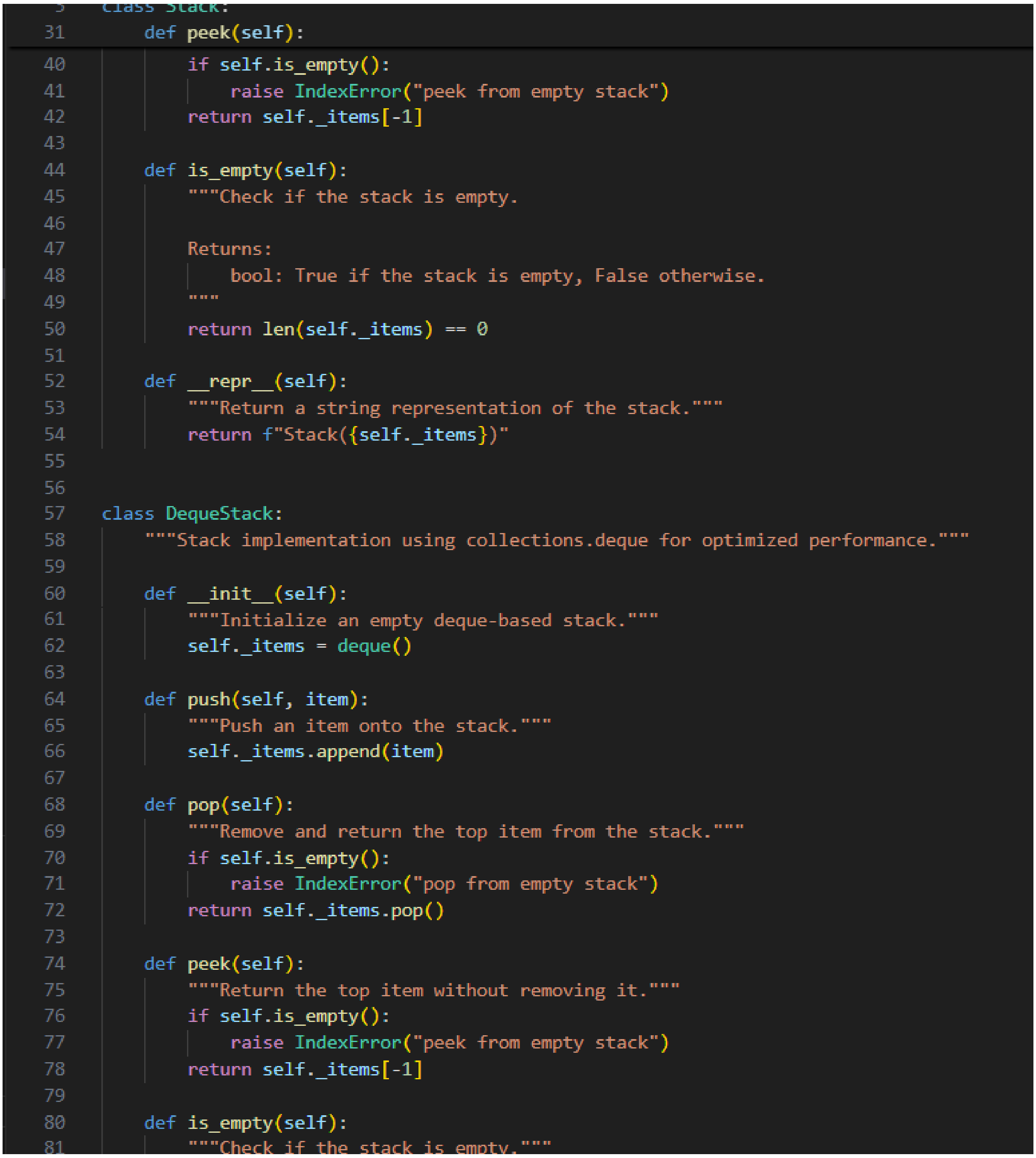
Task 1: Implementing a Stack (LIFO)

- **Task**: Use AI to help implement a **Stack** class in Python with the following operations: push(), pop(), peek(), and is_empty().

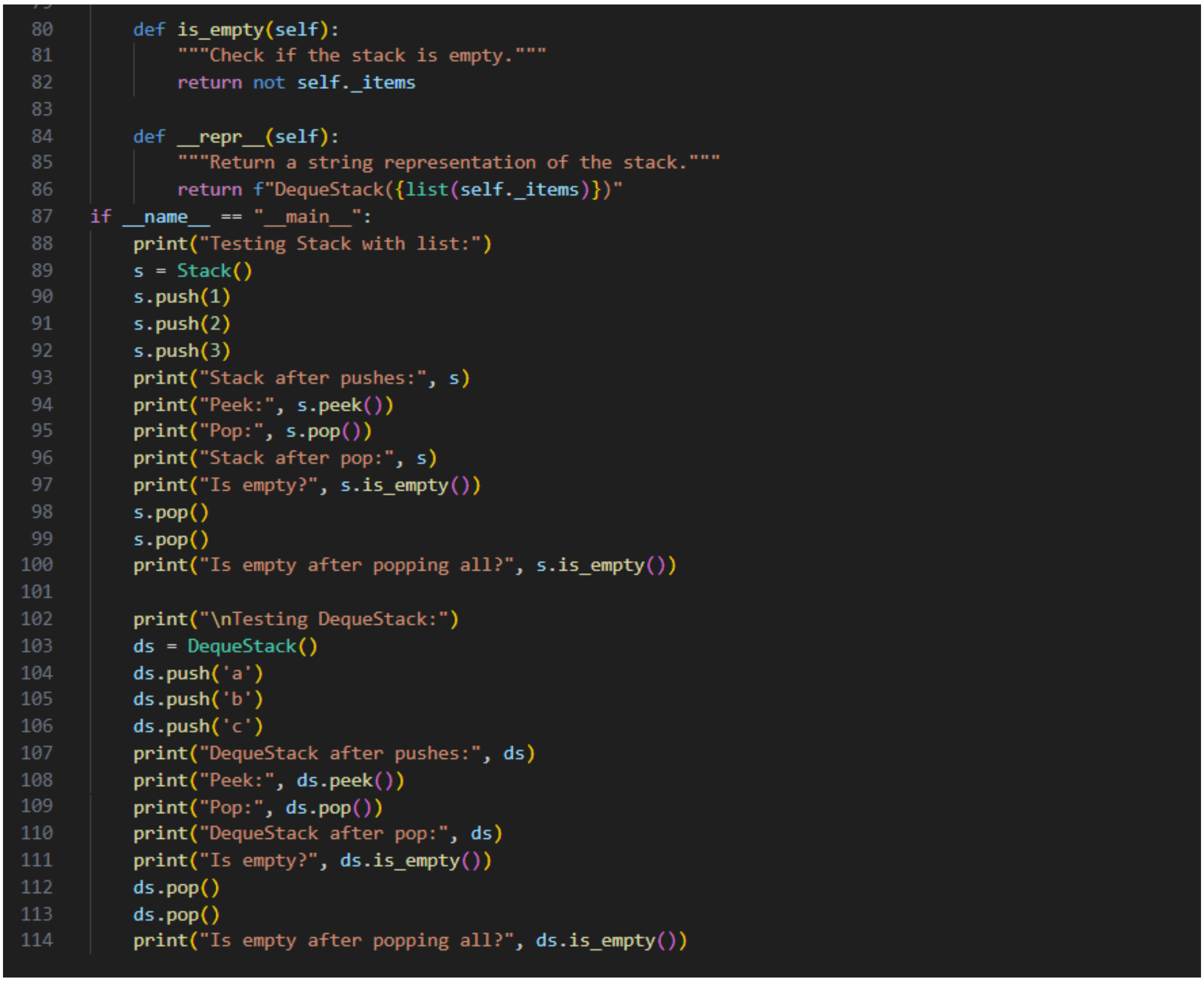
CODE:



```python
from collections import deque

class Stack:
    """A simple Stack (LIFO) implementation using a Python list."""

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

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

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

    def pop(self):
        """Remove and return the top item from the stack.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """Return the top item without removing it.

        Returns:
            The item at the top of the stack.

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

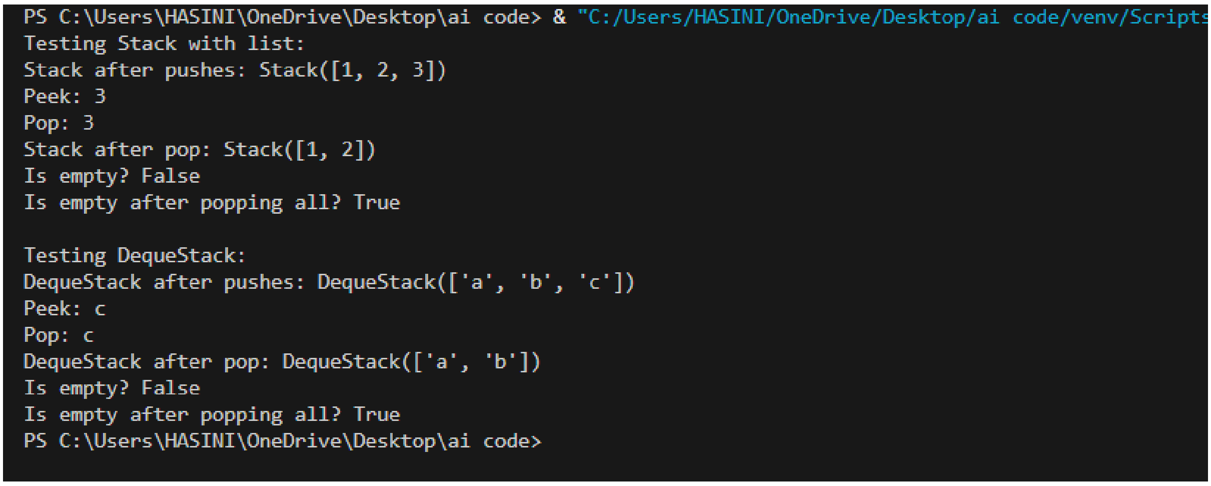


```python
class Stack:
    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]

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

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

    def __repr__(self):
        """Return a string representation of the stack."""
        return f"Stack({self._items})"


class DequeStack:
    """Stack implementation using collections.deque for optimized performance."""

    def __init__(self):
        """Initialize an empty deque-based stack."""
        self._items = deque()

    def push(self, item):
        """Push an item onto the stack."""
        self._items.append(item)

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

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

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



```python
    def is_empty(self):
        """Check if the stack is empty."""
        return not self._items

    def __repr__(self):
        """Return a string representation of the stack."""
        return f"DequeStack({list(self._items)})"
if __name__ == "__main__":
    print("Testing Stack with list:")
    s = Stack()
    s.push(1)
    s.push(2)
    s.push(3)
    print("Stack after pushes:", s)
    print("Peek:", s.peek())
    print("Pop:", s.pop())
    print("Stack after pop:", s)
    print("Is empty?", s.is_empty())
    s.pop()
    s.pop()
    print("Is empty after popping all?", s.is_empty())

    print("\nTesting DequeStack:")
    ds = DequeStack()
    ds.push('a')
    ds.push('b')
    ds.push('c')
    print("DequeStack after pushes:", ds)
    print("Peek:", ds.peek())
    print("Pop:", ds.pop())
    print("DequeStack after pop:", ds)
    print("Is empty?", ds.is_empty())
    ds.pop()
    ds.pop()
    print("Is empty after popping all?", ds.is_empty())
```

OUTPUT:



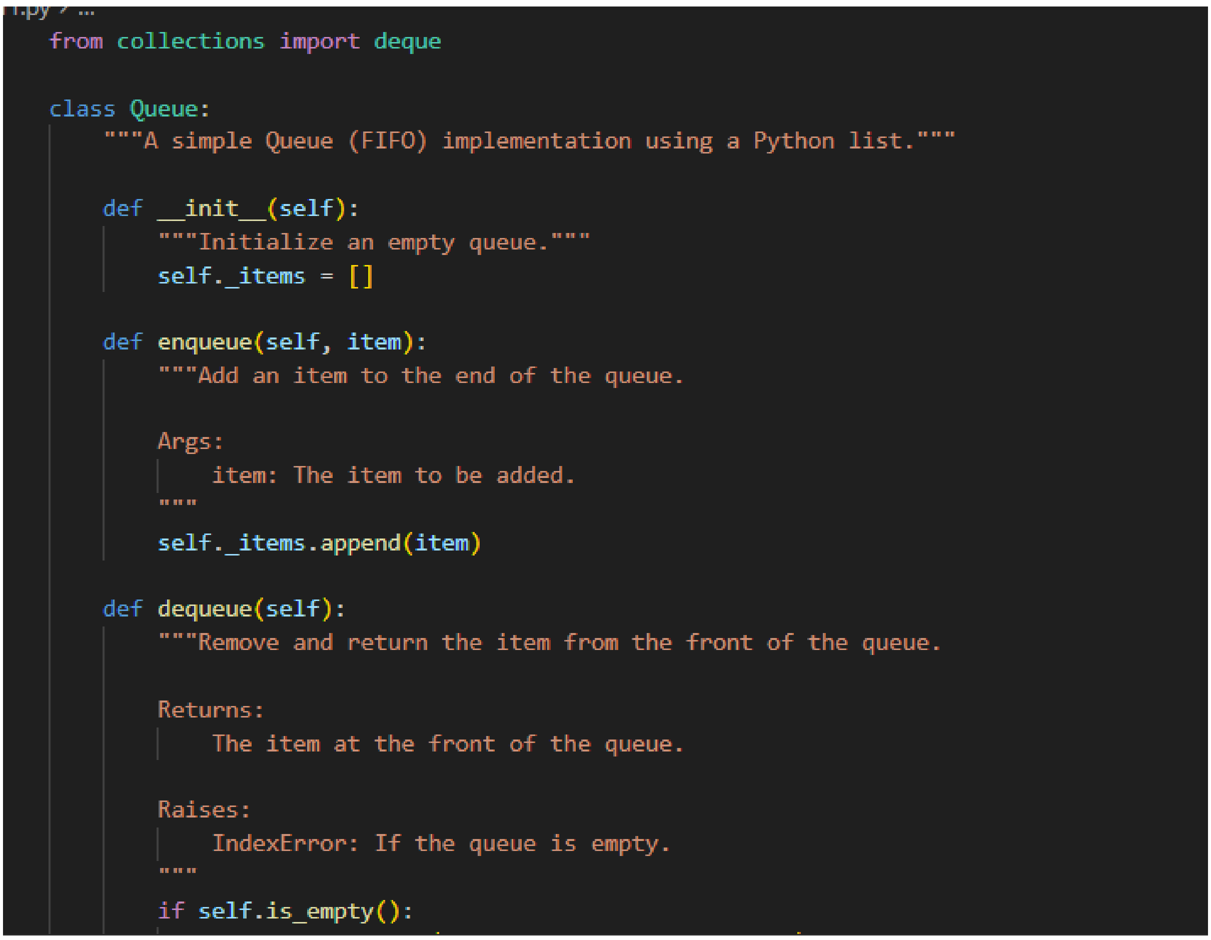
```
PS C:\Users\HASINI\OneDrive\Desktop\ai code> & "C:/Users/HASINI/OneDrive/Desktop/ai code/venv/Scripts
Testing Stack with list:
Stack after pushes: Stack([1, 2, 3])
Peek: 3
Pop: 3
Stack after pop: Stack([1, 2])
Is empty? False
Is empty after popping all? True

Testing DequeStack:
DequeStack after pushes: DequeStack(['a', 'b', 'c'])
Peek: c
Pop: c
DequeStack after pop: DequeStack(['a', 'b'])
Is empty? False
Is empty after popping all? True
PS C:\Users\HASINI\OneDrive\Desktop\ai code>
```

Observations:

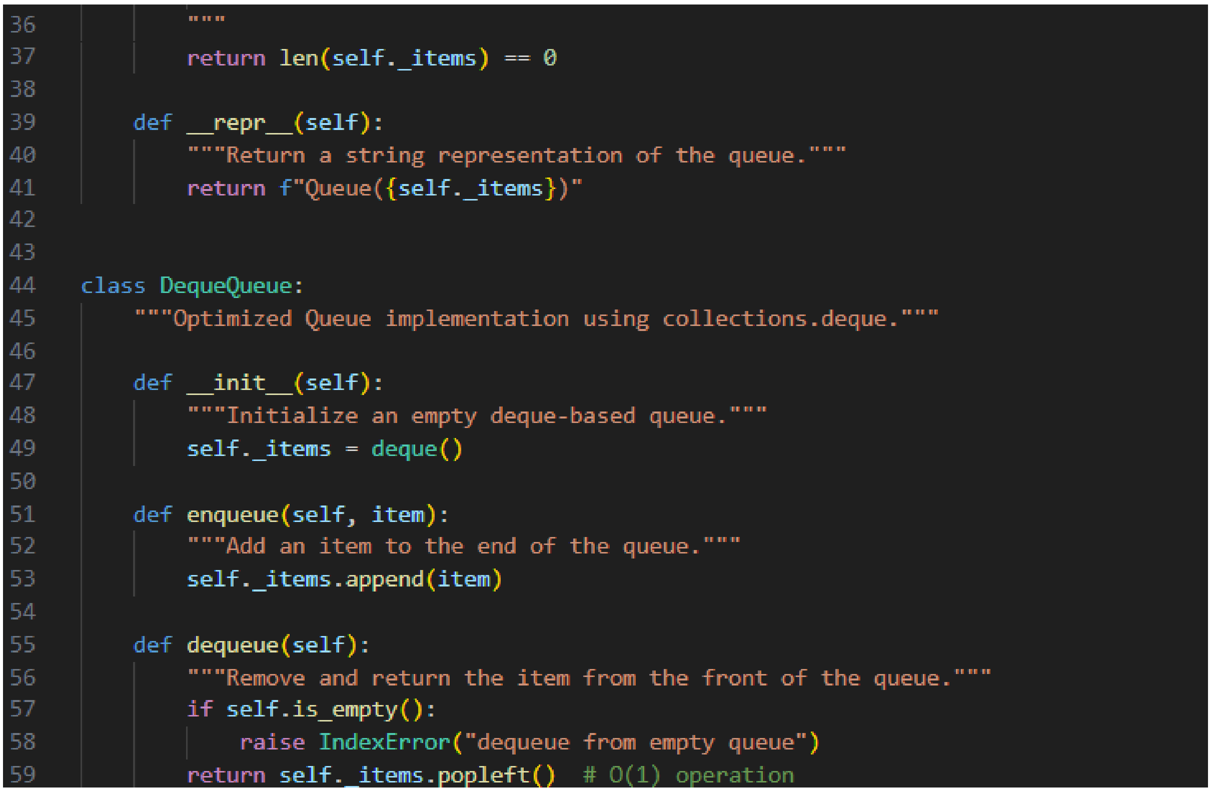
Task 2: Queue Implementation with Performance Review

- **Task**: Implement a **Queue** with enqueue(), dequeue(), and is_empty() methods.

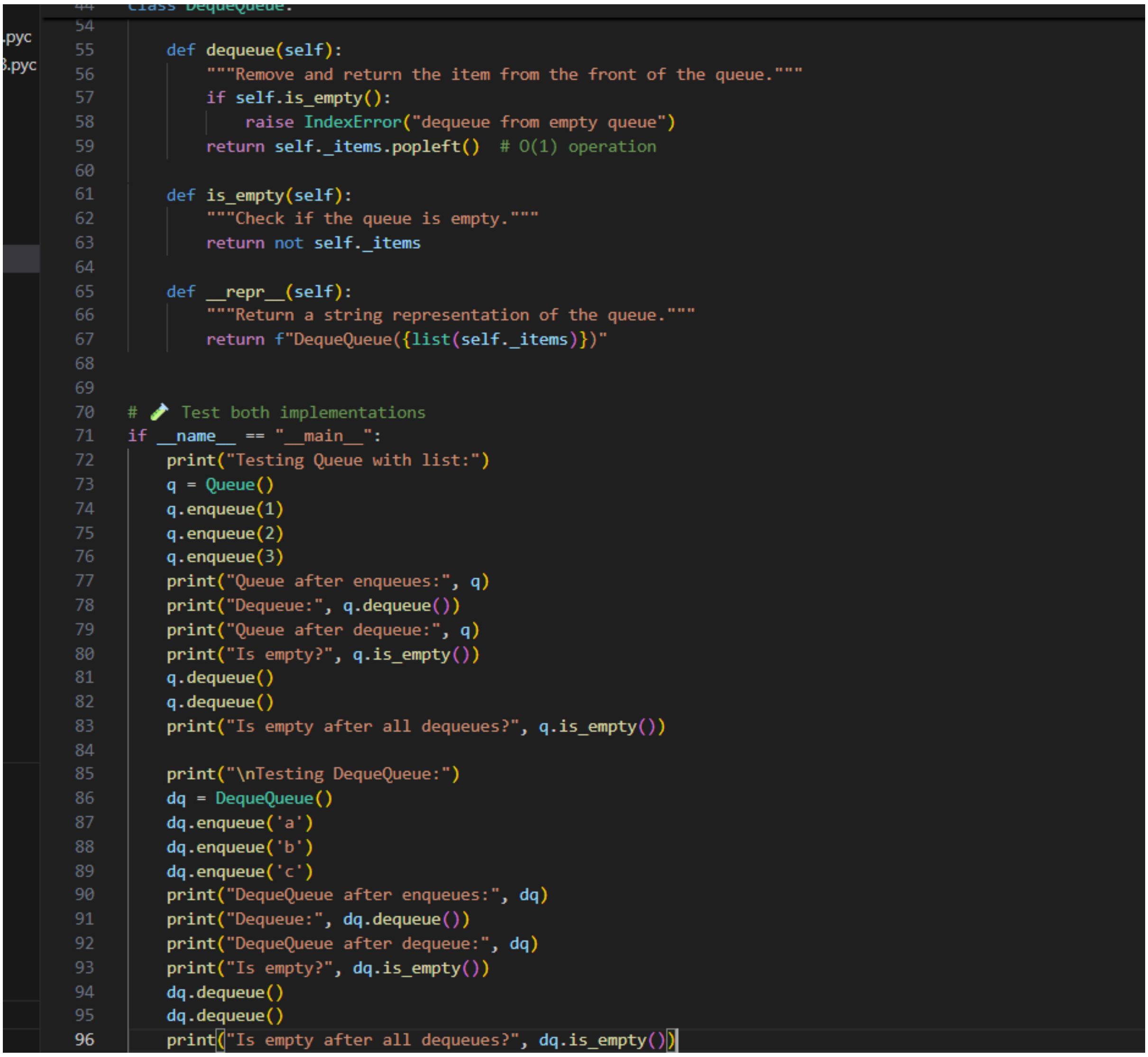
  CODE:



```python
from collections import deque

class Queue:
    """A simple Queue (FIFO) implementation using a Python list."""

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

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

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

    def dequeue(self):
        """Remove and return the item from the front of the queue.

        Returns:
            The item at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
```

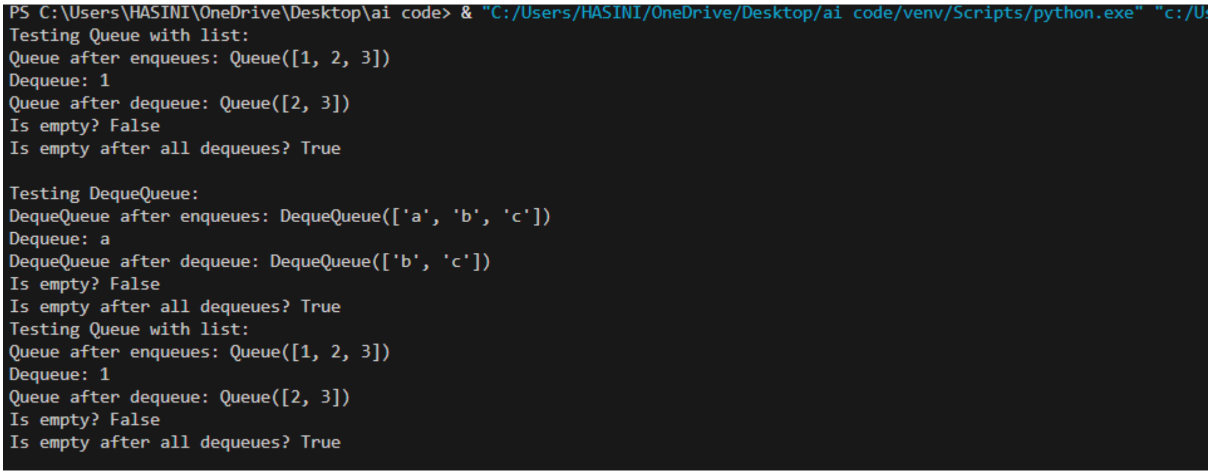


```python
        """
        return len(self._items) == 0

    def __repr__(self):
        """Return a string representation of the queue."""
        return f"Queue({self._items})"


class DequeQueue:
    """Optimized Queue implementation using collections.deque."""

    def __init__(self):
        """Initialize an empty deque-based queue."""
        self._items = deque()

    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 from the front of the queue."""
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.popleft()  # O(1) operation
```



```python
class DequeQueue:

    def dequeue(self):
        """Remove and return the item from the front of the queue."""
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.popleft()  # O(1) operation

    def is_empty(self):
        """Check if the queue is empty."""
        return not self._items

    def __repr__(self):
        """Return a string representation of the queue."""
        return f"DequeQueue({list(self._items)})"


# 🧪 Test both implementations
if __name__ == "__main__":
    print("Testing Queue with list:")
    q = Queue()
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)
    print("Queue after enqueues:", q)
    print("Dequeue:", q.dequeue())
    print("Queue after dequeue:", q)
    print("Is empty?", q.is_empty())
    q.dequeue()
    q.dequeue()
    print("Is empty after all dequeues?", q.is_empty())

    print("\nTesting DequeQueue:")
    dq = DequeQueue()
    dq.enqueue('a')
    dq.enqueue('b')
    dq.enqueue('c')
    print("DequeQueue after enqueues:", dq)
    print("Dequeue:", dq.dequeue())
    print("DequeQueue after dequeue:", dq)
    print("Is empty?", dq.is_empty())
    dq.dequeue()
    dq.dequeue()
    print("Is empty after all dequeues?", dq.is_empty())
```

OUTPUT:



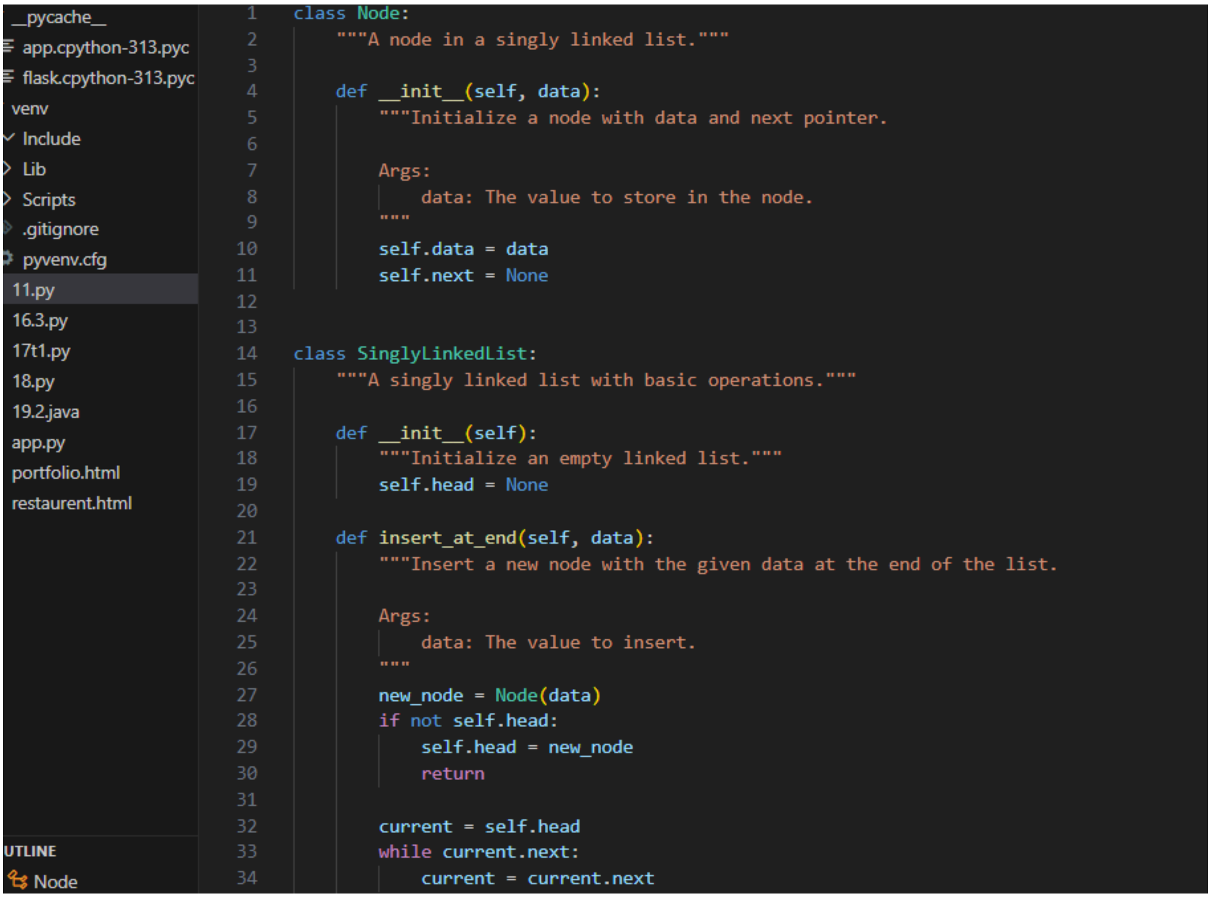
```
PS C:\Users\HASINI\OneDrive\Desktop\ai code> & "C:/Users/HASINI/OneDrive/Desktop/ai code/venv/Scripts/python.exe" "c:/U
Testing Queue with list:
Queue after enqueues: Queue([1, 2, 3])
Dequeue: 1
Queue after dequeue: Queue([2, 3])
Is empty? False
Is empty after all dequeues? True

Testing DequeQueue:
DequeQueue after enqueues: DequeQueue(['a', 'b', 'c'])
Dequeue: a
DequeQueue after dequeue: DequeQueue(['b', 'c'])
Is empty? False
Is empty after all dequeues? True
Testing Queue with list:
Queue after enqueues: Queue([1, 2, 3])
Dequeue: 1
Queue after dequeue: Queue([2, 3])
Is empty? False
Is empty after all dequeues? True
```

Observations:

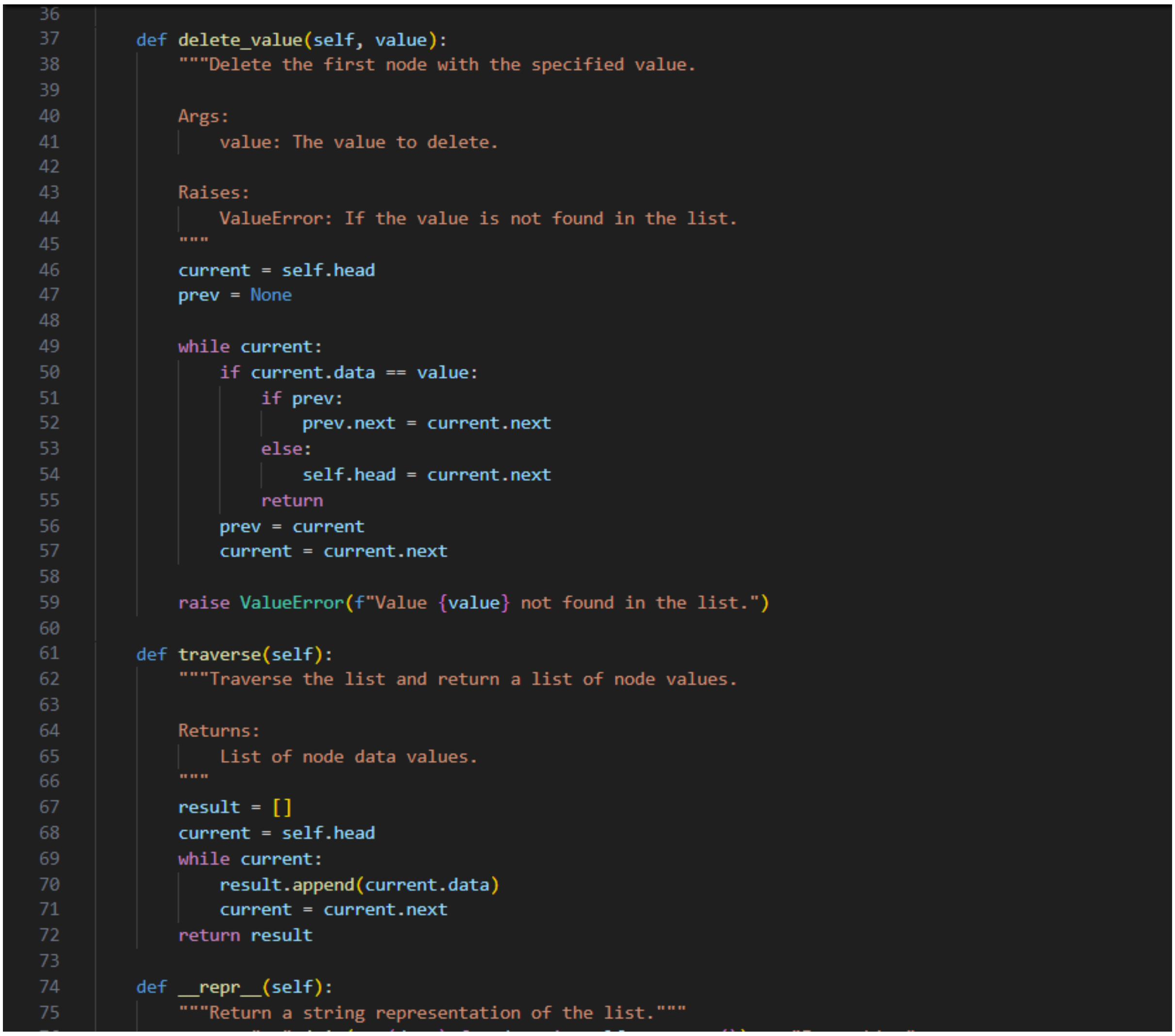
Task 3: Singly Linked List with Traversal

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

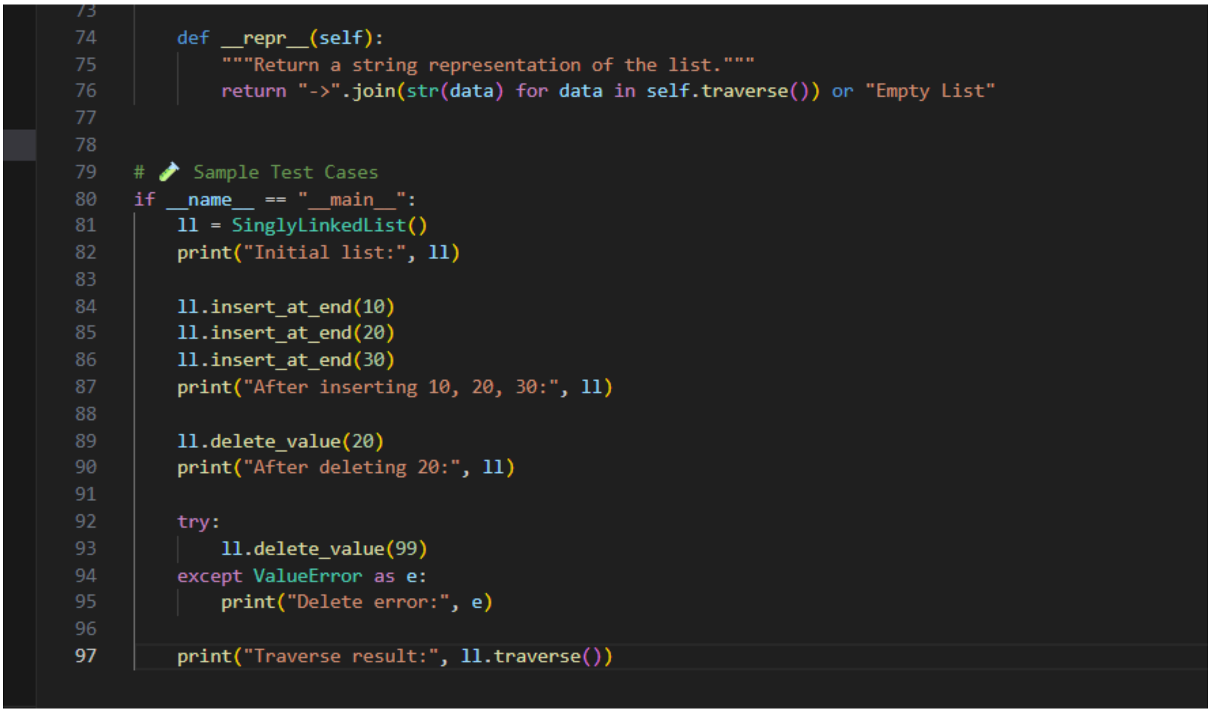
CODE:



```python
class Node:
    """A node in a singly linked list."""

    def __init__(self, data):
        """Initialize a node with data and next pointer.

        Args:
            data: The value to store in the node.
        """
        self.data = data
        self.next = None


class SinglyLinkedList:
    """A singly linked list with basic operations."""

    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None

    def insert_at_end(self, data):
        """Insert a new node with the given data at the end of the list.

        Args:
            data: The value to insert.
        """
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return

        current = self.head
        while current.next:
            current = current.next
```

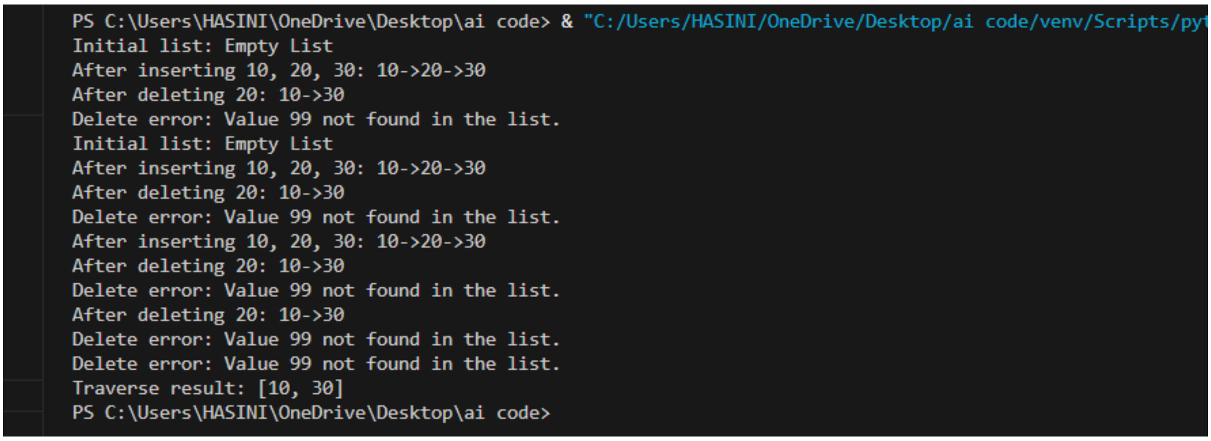


```python
    def delete_value(self, value):
        """Delete the first node with the specified value.

        Args:
            value: The value to delete.

        Raises:
            ValueError: If the value is not found in the list.
        """
        current = self.head
        prev = None

        while current:
            if current.data == value:
                if prev:
                    prev.next = current.next
                else:
                    self.head = current.next
                return
            prev = current
            current = current.next

        raise ValueError(f"Value {value} not found in the list.")

    def traverse(self):
        """Traverse the list and return a list of node values.

        Returns:
            List of node data values.
        """
        result = []
        current = self.head
        while current:
            result.append(current.data)
            current = current.next
        return result

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



```python
    def __repr__(self):
        """Return a string representation of the list."""
        return "->".join(str(data) for data in self.traverse()) or "Empty List"


# 🧪 Sample Test Cases
if __name__ == "__main__":
    ll = SinglyLinkedList()
    print("Initial list:", ll)

    ll.insert_at_end(10)
    ll.insert_at_end(20)
    ll.insert_at_end(30)
    print("After inserting 10, 20, 30:", ll)

    ll.delete_value(20)
    print("After deleting 20:", ll)

    try:
        ll.delete_value(99)
    except ValueError as e:
        print("Delete error:", e)

    print("Traverse result:", ll.traverse())
```

OUTPUT:



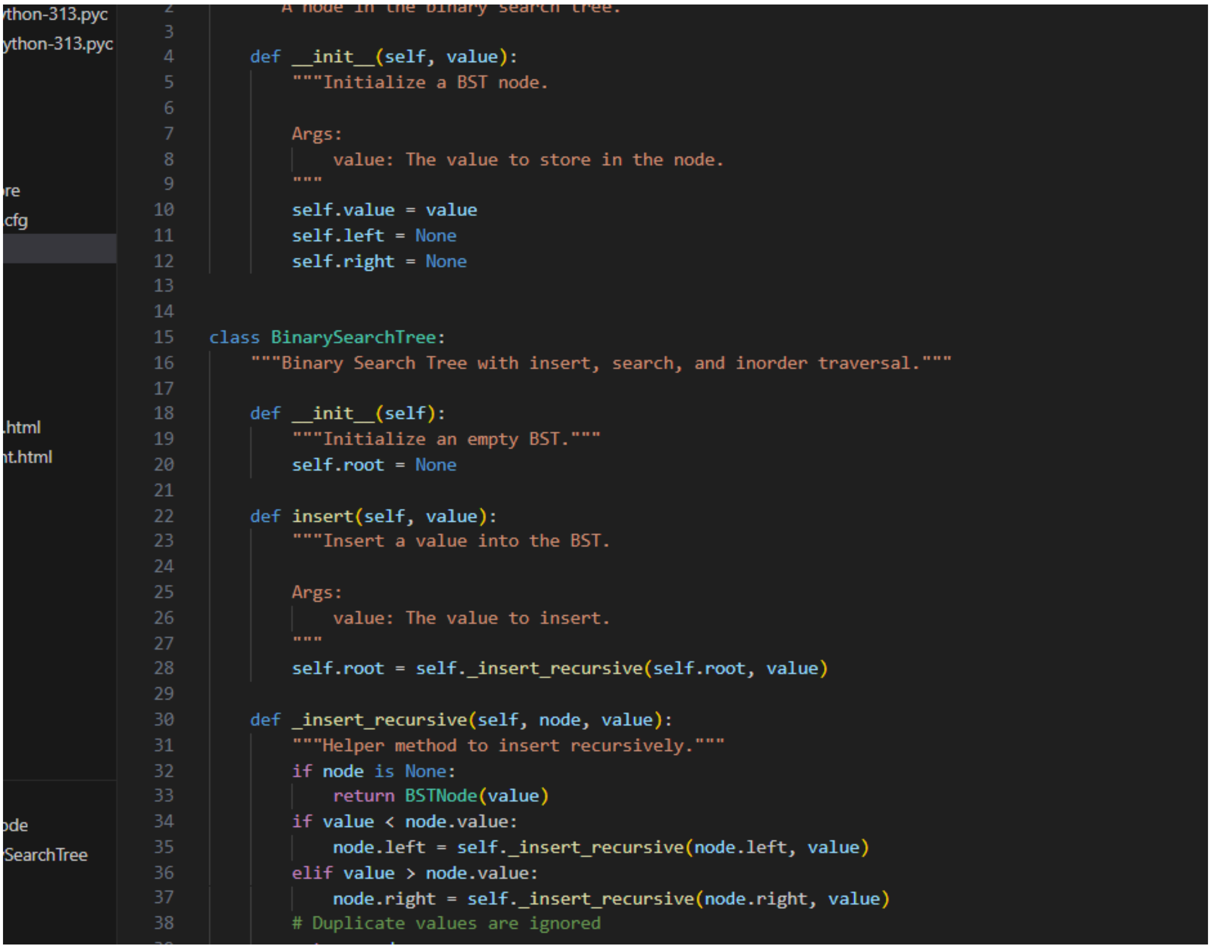
```
PS C:\Users\HASINI\OneDrive\Desktop\ai code> & "C:/Users/HASINI/OneDrive/Desktop/ai code/venv/Scripts/pyt
Initial list: Empty List
After inserting 10, 20, 30: 10->20->30
After deleting 20: 10->30
Delete error: Value 99 not found in the list.
Initial list: Empty List
After inserting 10, 20, 30: 10->20->30
After deleting 20: 10->30
Delete error: Value 99 not found in the list.
After inserting 10, 20, 30: 10->20->30
After deleting 20: 10->30
Delete error: Value 99 not found in the list.
After deleting 20: 10->30
Delete error: Value 99 not found in the list.
Delete error: Value 99 not found in the list.
Traverse result: [10, 30]
PS C:\Users\HASINI\OneDrive\Desktop\ai code>
```

Observations:

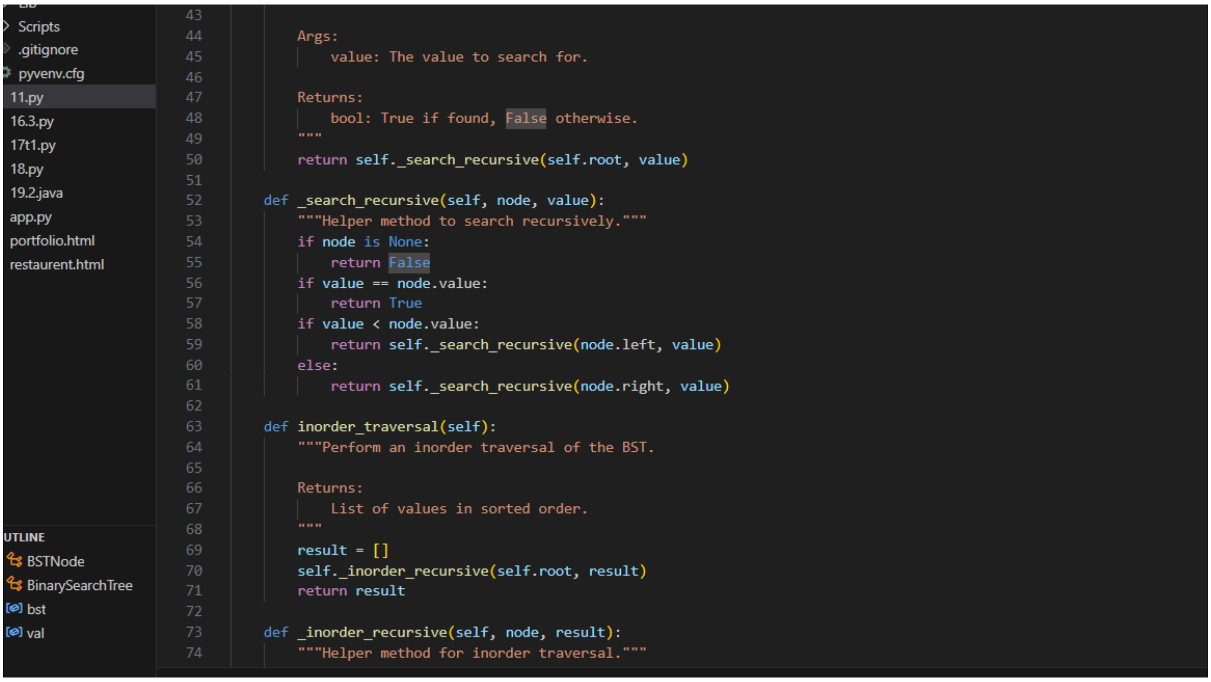
Task 4: Binary Search Tree (BST)

- **Task**: Implement a **Binary Search Tree** with methods for insert(), search(), and inorder_traversal().

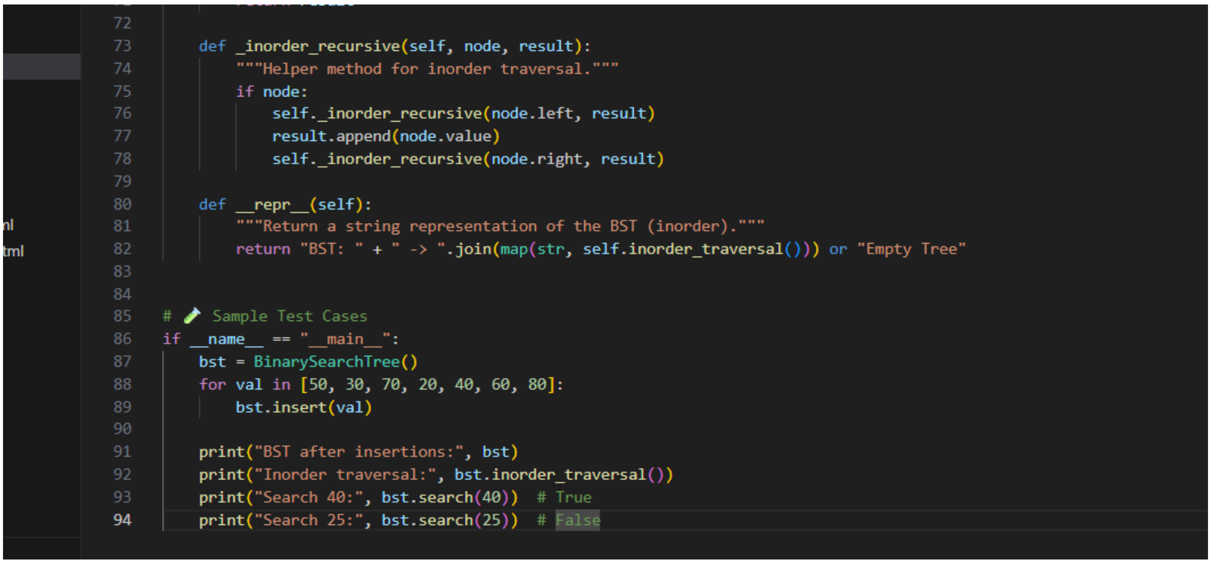
CODE:



```
        A node in the binary search tree.

    def __init__(self, value):
        """Initialize a BST node.

        Args:
            value: The value to store in the node.
        """
        self.value = value
        self.left = None
        self.right = None


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

    def __init__(self):
        """Initialize an empty BST."""
        self.root = None

    def insert(self, value):
        """Insert a value into the BST.

        Args:
            value: The value to insert.
        """
        self.root = self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        """Helper method to insert recursively."""
        if node is None:
            return BSTNode(value)
        if value < node.value:
            node.left = self._insert_recursive(node.left, value)
        elif value > node.value:
            node.right = self._insert_recursive(node.right, value)
        # Duplicate values are ignored
```

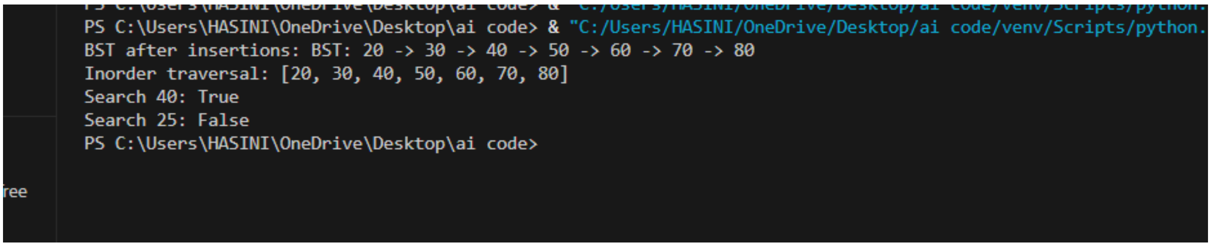


```

        Args:
            value: The value to search for.

        Returns:
            bool: True if found, False otherwise.
        """
        return self._search_recursive(self.root, value)

    def _search_recursive(self, node, value):
        """Helper method to search recursively."""
        if node is None:
            return False
        if value == node.value:
            return True
        if value < node.value:
            return self._search_recursive(node.left, value)
        else:
            return self._search_recursive(node.right, value)

    def inorder_traversal(self):
        """Perform an inorder traversal of the BST.

        Returns:
            List of values in sorted order.
        """
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        """Helper method for inorder traversal."""
```



```

    def _inorder_recursive(self, node, result):
        """Helper method for inorder traversal."""
        if node:
            self._inorder_recursive(node.left, result)
            result.append(node.value)
            self._inorder_recursive(node.right, result)

    def __repr__(self):
        """Return a string representation of the BST (inorder)."""
        return "BST: " + " -> ".join(map(str, self.inorder_traversal())) or "Empty Tree"


# ✏️ Sample Test Cases
if __name__ == "__main__":
    bst = BinarySearchTree()
    for val in [50, 30, 70, 20, 40, 60, 80]:
        bst.insert(val)

    print("BST after insertions:", bst)
    print("Inorder traversal:", bst.inorder_traversal())
    print("Search 40:", bst.search(40))  # True
    print("Search 25:", bst.search(25))  # False
```

OUTPUT:



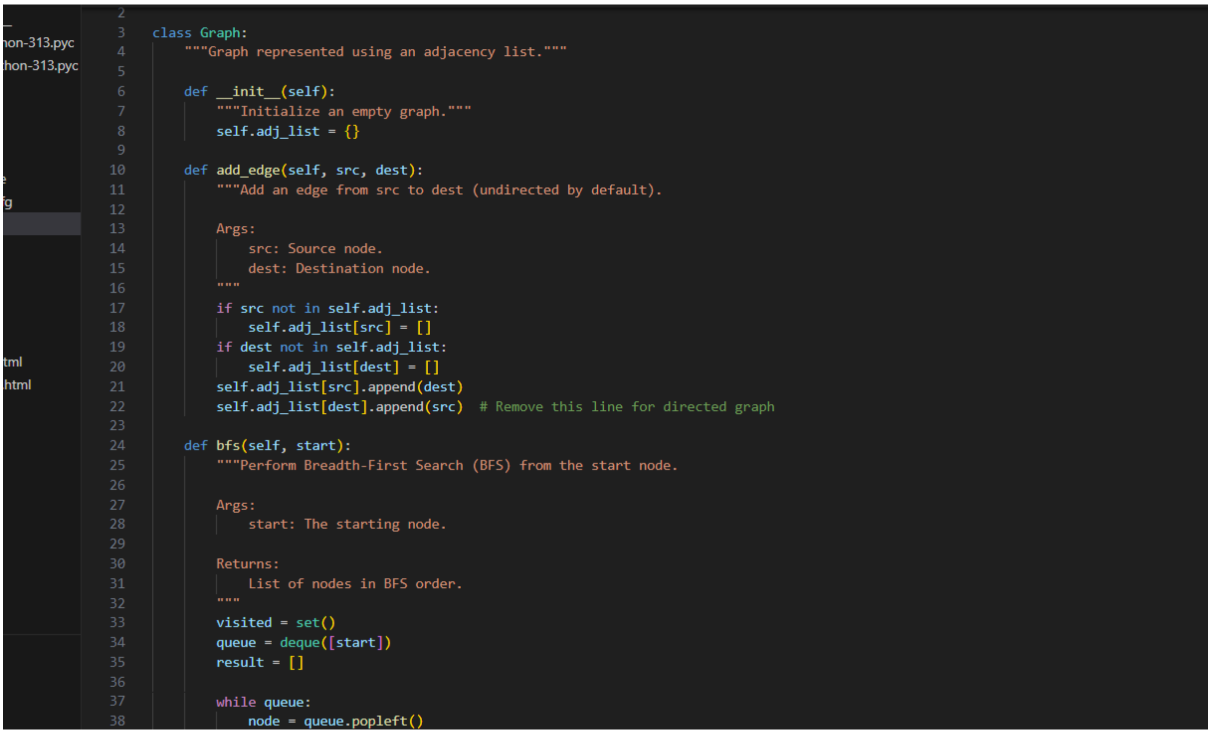
```
PS C:\Users\HASINI\OneDrive\Desktop\ai code> & "C:/Users/HASINI/OneDrive/Desktop/ai code/venv/Scripts/python.
BST after insertions: BST: 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 80
Inorder traversal: [20, 30, 40, 50, 60, 70, 80]
Search 40: True
Search 25: False
PS C:\Users\HASINI\OneDrive\Desktop\ai code>
```

Observations:

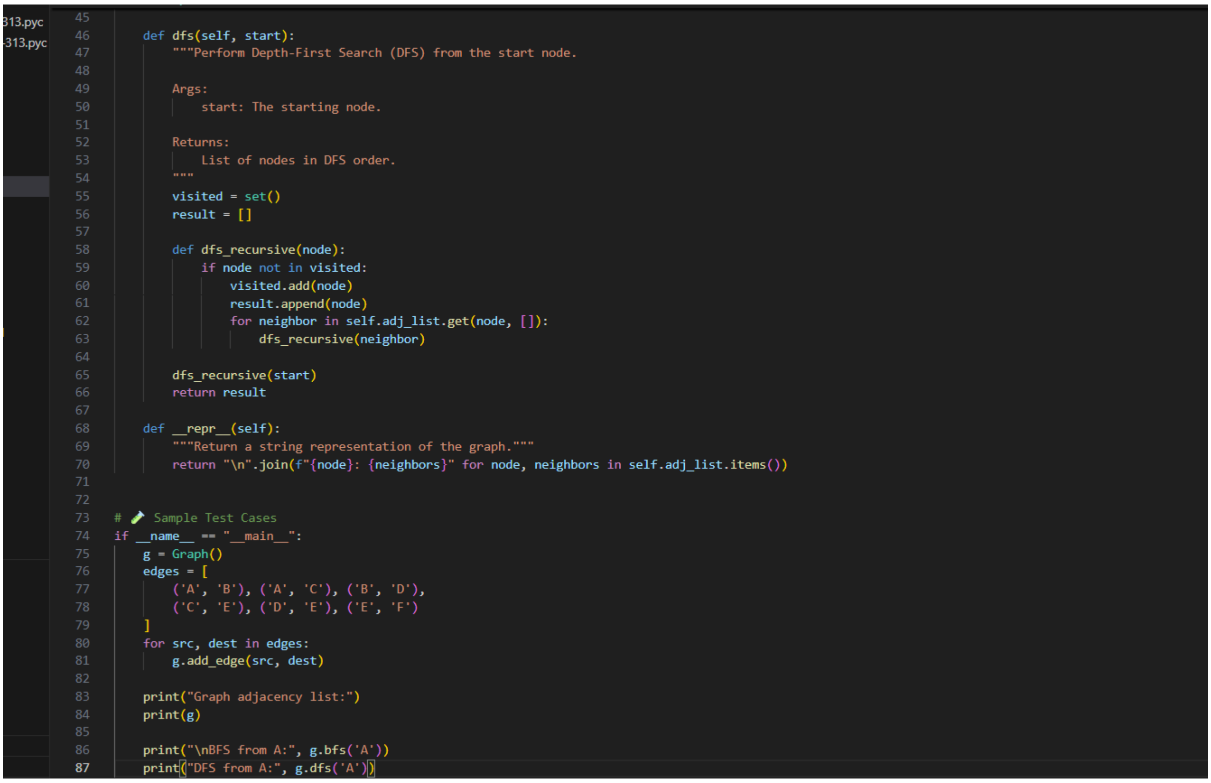
Task 5: Graph Representation and BFS/DFS Traversal

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

CODE:

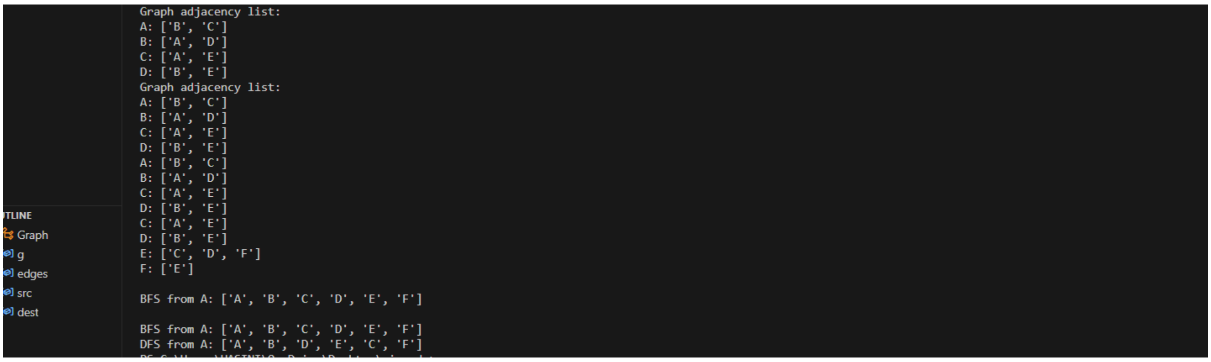


```python

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

    def __init__(self):
        """Initialize an empty graph."""
        self.adj_list = {}

    def add_edge(self, src, dest):
        """Add an edge from src to dest (undirected by default).

        Args:
            src: Source node.
            dest: Destination node.
        """
        if src not in self.adj_list:
            self.adj_list[src] = []
        if dest not in self.adj_list:
            self.adj_list[dest] = []
        self.adj_list[src].append(dest)
        self.adj_list[dest].append(src)  # Remove this line for directed graph

    def bfs(self, start):
        """Perform Breadth-First Search (BFS) from the start node.

        Args:
            start: The starting node.

        Returns:
            List of nodes in BFS order.
        """
        visited = set()
        queue = deque([start])
        result = []

        while queue:
            node = queue.popleft()
```



```python

    def dfs(self, start):
        """Perform Depth-First Search (DFS) from the start node.

        Args:
            start: The starting node.

        Returns:
            List of nodes in DFS order.
        """
        visited = set()
        result = []

        def dfs_recursive(node):
            if node not in visited:
                visited.add(node)
                result.append(node)
                for neighbor in self.adj_list.get(node, []):
                    dfs_recursive(neighbor)

        dfs_recursive(start)
        return result

    def __repr__(self):
        """Return a string representation of the graph."""
        return "\n".join(f"{node}: {neighbors}" for node, neighbors in self.adj_list.items())


# 🧪 Sample Test Cases
if __name__ == "__main__":
    g = Graph()
    edges = [
        ('A', 'B'), ('A', 'C'), ('B', 'D'),
        ('C', 'E'), ('D', 'E'), ('E', 'F')
    ]
    for src, dest in edges:
        g.add_edge(src, dest)

    print("Graph adjacency list:")
    print(g)

    print("\nBFS from A:", g.bfs('A'))
    print("DFS from A:", g.dfs('A'))
```

OUTPUT:



```
Graph adjacency list:
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'E']
D: ['B', 'E']
Graph adjacency list:
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'E']
D: ['B', 'E']
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'E']
D: ['B', 'E']
C: ['A', 'E']
D: ['B', 'E']
E: ['C', 'D', 'F']
F: ['E']

BFS from A: ['A', 'B', 'C', 'D', 'E', 'F']

BFS from A: ['A', 'B', 'C', 'D', 'E', 'F']
DFS from A: ['A', 'B', 'D', 'E', 'C', 'F']
```

Observation:

- Uses dict for adjacency list — efficient and readable
- BFS uses deque for O(1) pops
- DFS uses recursion — elegant for small graphs
- For large graphs, consider iterative DFS to avoid recursion depth issues