ASSIGNMENT-11.4

TASK-1:

prompt: Implement a **Stack** class in Python with the following operations: push(), pop(), peek(), and is_empty().

Code with output:

```
from collections import deque
class DequeStack:
    def __init__(self):
        self.items = deque()
    def push(self, item):
        """Adds an item to the top of the stack."""
        self.items.append(item)
    def pop(self):
        """Removes and returns the item from the top of the stack."""
        if not self.is_empty():
            return self.items.pop()
        else:
            return "Stack is empty"
    def peek(self):
        """Returns the item at the top of the stack without removing it."""
        if not self.is_empty():
            return self.items[-1]
        else:
            return "Stack is empty"
    def is_empty(self):
        """Checks if the stack is empty."""
        return len(self.items) == 0
# Example usage:
deque_stack = DequeStack()
print(f"Is the deque stack empty? {deque_stack.is_empty()}")
deque_stack.push(100)
deque_stack.push(200)
deque_stack.push(300)
print(f"Is the deque stack empty? {deque_stack.is_empty()}")
print(f"Top element: {deque_stack.peek()}")
print(f"Popped element: {deque_stack.pop()}")
print(f"Top element after pop: {deque_stack.peek()}")
print(f"Popped element: {deque_stack.pop()}")
print(f"Popped element: {deque_stack.pop()}")
print(f"Popped element from empty deque stack: {deque_stack.pop()}")
```

```
Is the deque stack empty? True
Is the deque stack empty? False
Top element: 300
Popped element: 300
Top element after pop: 200
Popped element: 200
Popped element: 100
Popped element from empty deque stack: Stack is empty
```

EXPLANATION:

- `from collections import deque`: This line imports the `deque` class from the `collections` module. A deque (double-ended queue) is a list-like container with fast appends and pops from either end.
- `class DequeStack:`: This defines the `DequeStack` class.
- `def __init__(self):`: This is the constructor. It initializes an empty `deque` called `self.items` to store the stack elements.
- `def push(self, item):`: This method adds an `item` to the top of the stack using `self.items.append(item)`. Since a deque allows fast appends to the right, this is an efficient way to push onto the stack.
- `def pop(self, item):`: This method removes and returns the item from the top of the stack. It first checks if the stack is empty using `self.is_empty()`. If not empty, it uses `self.items.pop()` to remove and return the last element (which is the top of the stack). If the stack is empty, it returns "Stack is empty".
- `def peek(self):`: This method returns the item at the top of the stack without removing it. It checks if the stack is empty and, if not, returns the last element using `self.items[-1]`.
- `def is_empty(self):`: This method checks if the stack is empty by checking if the length of `self.items` is 0.
- **Example usage**: The code then demonstrates how to create a `DequeStack` object, push elements onto it, check if it's empty, peek at the top element, and pop elements off the stack.

This implementation leverages the `deque`'s efficient append and pop operations from the right end, making it a suitable choice for implementing a stack.

TASK-2:

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

Code with output:

```python
from collections import deque
class DequeQueue:
    def __init__(self):
        self.items = deque()
    def enqueue(self, item):
        """Adds an item to the rear of the queue."""
        self.items.append(item) # Efficiently adds to the right end
    def dequeue(self):
        """Removes and returns the item from the front of the queue."""
        if not self.is_empty():
            return self.items.popleft() # Efficiently removes from the left end
        else:
            return "Queue is empty"
    def is_empty(self):
        """Checks if the queue is empty."""
        return len(self.items) == 0
# Example usage:
deque_queue = DequeQueue()
print(f"Is the deque queue empty? {deque_queue.is_empty()}")
deque_queue.enqueue(10)
deque_queue.enqueue(20)
deque_queue.enqueue(30)
print(f"Is the deque queue empty? {deque_queue.is_empty()}")
print(f"Dequeued element: {deque_queue.dequeue()}")
print(f"Dequeued element: {deque_queue.dequeue()}")
print(f"Dequeued element: {deque_queue.dequeue()}")
print(f"Dequeued element from empty deque queue: {deque_queue.dequeue()}")
```

```
Is the deque queue empty? True
Is the deque queue empty? False
Dequeued element: 10
Dequeued element: 20
Dequeued element: 30
Dequeued element from empty deque queue: Queue is empty
```

EXPLANATION:

- **`from collections import deque`**: This line imports the `deque` class. A `deque` (double-ended queue) is optimized for adding and removing elements from both ends quickly, which is perfect for a queue.
- **`class DequeQueue:`**: This line starts the definition of our custom queue class.
- **`def __init__(self):`**: This is the constructor. When you create a `DequeQueue` object, this method runs. It initializes an empty `deque` called `self.items` to hold the elements of the queue.
- **`def enqueue(self, item):`**: This method adds an `item` to the back (or rear) of the queue. `self.items.append(item)` is used, which efficiently adds the item to the right side of the `deque`.
- **`def dequeue(self):`**: This method removes and returns the item from the front of the queue. It first checks if the queue is empty using `self.is_empty()`. If it's not empty, `self.items.popleft()` is used. This efficiently removes and returns the leftmost element of the `deque`, which is the front of the queue. If the queue is empty, it returns "Queue is empty".
- **`def is_empty(self):`**: This method checks if the queue is empty by returning `True` if the length of the `self.items` deque is 0, and `False` otherwise.
- **Example usage**: The lines after the class definition demonstrate how to create a `DequeQueue` object, add elements with `enqueue`, check if it's empty with `is_empty`, and remove elements with `dequeue`.

TASK-3:

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

Code with output:

```
# Test cases for LinkedList

# Test case 1: Insert into an empty list and traverse
print("--- Test Case 1: Insert into empty list ---")
my_list = LinkedList()
my_list.insert_at_end(10)
my_list.traverse() # Expected output: Linked List: 10 -> None

# Test case 2: Insert multiple elements and traverse
print("\n--- Test Case 2: Insert multiple elements ---")
my_list.insert_at_end(20)
my_list.insert_at_end(30)
my_list.traverse() # Expected output: Linked List: 10 -> 20 -> 30 -> None

# Test case 3: Delete a value that exists (middle)
print("\n--- Test Case 3: Delete existing value (middle) ---")
my_list.delete_value(20)
my_list.traverse() # Expected output: Linked List: 10 -> 30 -> None

# Test case 4: Delete a value that does not exist
print("\n--- Test Case 4: Delete non-existing value ---")
my_list.delete_value(99) # Expected output: Value 99 not found in the list.
my_list.traverse() # Expected output: Linked List: 10 -> 30 -> None (list remains unchanged)

# Test case 5: Delete the head node
print("\n--- Test Case 5: Delete the head node ---")
my_list.delete_value(10)
my_list.traverse() # Expected output: Linked List: 30 -> None
```

```python
# Test case 6: Delete the last node
print("\n--- Test Case 6: Delete the last node ---")
my_list.delete_value(30)
my_list.traverse() # Expected output: The list is empty.

# Test case 7: Delete from an empty list
print("\n--- Test Case 7: Delete from an empty list ---")
my_list.delete_value(5) # Expected output: Value 5 not found in the list.
my_list.traverse() # Expected output: The list is empty.

# Test case 8: Insert after deleting all elements
print("\n--- Test Case 8: Insert after deleting all elements ---")
my_list.insert_at_end(50)
my_list.traverse() # Expected output: Linked List: 50 -> None
```

```
--- Test Case 1: Insert into empty list ---
Linked List:
10None

--- Test Case 2: Insert multiple elements ---
Linked List:
10 -> 20 -> 30None

--- Test Case 3: Delete existing value (middle) ---
Linked List:
10 -> 30None

--- Test Case 4: Delete non-existing value ---
Value 99 not found in the list.
Linked List:
10 -> 30None

--- Test Case 5: Delete the head node ---
Linked List:
30None
```

```
--- Test Case 6: Delete the last node ---
The list is empty.

--- Test Case 7: Delete from an empty list ---
Value 5 not found in the list.
The list is empty.

--- Test Case 8: Insert after deleting all elements ---
Linked List:
50None
```

EXPLANATION:

- **Test Case 1: Insert into empty list**: This case creates an empty linked list and inserts a single element (10). It then traverses the list to show that the element was added correctly as the head.
- **Test Case 2: Insert multiple elements**: This builds on the previous case by inserting two more elements (20 and 30) to the end of the list and then traverses to show the updated list.
- **Test Case 3: Delete existing value (middle)**: This case tests deleting an element (20) that is in the middle of the list. The traversal after deletion shows that 20 has been removed and the list is correctly linked.
- **Test Case 4: Delete non-existing value**: This case attempts to delete a value (99) that is not in the list. The expected output confirms that the value was not found, and the traversal shows that the list remains unchanged.
- **Test Case 5: Delete the head node**: This case tests deleting the first element (10) of the list. The traversal shows that the next element (30) has become the new head.
- **Test Case 6: Delete the last node**: This case tests deleting the last element (30) of the list. The traversal shows that the list is now empty.
- **Test Case 7: Delete from an empty list**: This case attempts to delete a value (5) from an already empty list. The expected output confirms that the value was not found, and the traversal shows that the list remains empty.
- **Test Case 8: Insert after deleting all elements**: This case tests inserting an element (50) into a list that was previously emptied. The traversal shows that the element is correctly inserted as the new head.

TASK-4:

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

Code with output:

```python
# 1. Create a list of integers
test_data = [50, 30, 70, 20, 40, 60, 80]

# 2. Instantiate a BinarySearchTree object
bst = BinarySearchTree()

# 3. Insert each integer into the BST
for num in test_data:
    bst.insert(num)

# 4. Call inorder_traversal() and print the result
print("Inorder Traversal:")
print(bst.inorder_traversal())

# 5. Search for a present element
present_element = 40
print(f"\nSearching for {present_element}: {bst.search(present_element)}")

# 6. Search for an absent element
absent_element = 99
print(f"Searching for {absent_element}: {bst.search(absent_element)}")
```

```
Inorder Traversal:
[20, 30, 40, 50, 60, 70, 80]

Searching for 40: True
Searching for 99: False
```

EXPLANATION:

1. `test_data = [50, 30, 70, 20, 40, 60, 80]`: This line creates a list of integers that will be used to build the binary search tree.
2. `bst = BinarySearchTree()`: This line creates an instance of the `BinarySearchTree` class, initializing an empty tree.
3. `for num in test_data: bst.insert(num)`: This loop iterates through each number in the `test_data` list and inserts it into the binary search tree using the `insert()` method. The `insert()` method ensures that each number is placed in the correct position to maintain the BST property (smaller values in the left subtree, larger values in the right subtree).
4. `print("Inorder Traversal:") print(bst.inorder_traversal())`: This calls the `inorder_traversal()` method on the `bst` object. Inorder traversal visits the nodes of a BST in a way that results in the elements being returned in sorted order. The output will be the sorted list of the numbers inserted into the tree.
5. `present_element = 40 print(f"\nSearching for {present_element}: {bst.search(present_element)}")`: This section tests the `search()` method with an element (40) that is known to be present in the `test_data`. The output will be `True` because 40 was inserted into the tree.
6. `absent_element = 99 print(f"Searching for {absent_element}: {bst.search(absent_element)}")`: This section tests the `search()` method with an element (99) that is not in the `test_data`. The output will be `False` because 99 was not inserted into the tree.

TASK-5:

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

Code with output:

```
# Represents the graph using an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

from collections import deque

def bfs(graph, start_node):
    """
    Performs a Breadth-First Search on a graph starting from a given node.

    Args:
        graph: A dictionary representing the graph using an adjacency list.
        start_node: The node to start the traversal from.
    """
    # Initialize a set to keep track of visited nodes
    visited = set()
    # Initialize a queue for nodes to visit (using a deque for efficient appending and popping)
    queue = deque([start_node])

    print(f"BFS starting from {start_node}:")
    # Continue the loop as long as the queue is not empty
    while queue:
        # Dequeue a node from the front of the queue
        current_node = queue.popleft()

        # If the current node has not been visited
        if current_node not in visited:
            # Mark the current node as visited
            visited.add(current_node)
```

```python
        # Process the current node (e.g., print it)
        print(current_node, end=" ")

        # Enqueue all unvisited neighbors of the current node
        for neighbor in graph.get(current_node, []):
            if neighbor not in visited:
                queue.append(neighbor)
    print("\n")

def dfs_recursive(graph, start_node, visited=None):
    """
    Performs a recursive Depth-First Search on a graph starting from a given node.

    Args:
        graph: A dictionary representing the graph using an adjacency list.
        start_node: The node to start the traversal from.
        visited: A set to keep track of visited nodes (used for recursion).
    """
    # Initialize the visited set if it's the initial call
    if visited is None:
        visited = set()

    # Mark the current node as visited
    visited.add(start_node)
    # Process the current node (e.g., print it)
    print(start_node, end=" ")

    # Recursively visit all unvisited neighbors
    for neighbor in graph.get(start_node, []):
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

def dfs_iterative(graph, start_node):
    """
    Performs an iterative Depth-First Search on a graph starting from a given node.

    Args:
```

```python
        start_node: The node to start the traversal from.
    """
    # Initialize a set to keep track of visited nodes
    visited = set()
    # Initialize a stack for nodes to visit (using a list)
    stack = [start_node]

    print(f"Iterative DFS starting from {start_node}:")
    # Continue the loop as long as the stack is not empty
    while stack:
        # Pop a node from the top of the stack
        current_node = stack.pop()

        # If the current node has not been visited
        if current_node not in visited:
            # Mark the current node as visited
            visited.add(current_node)
            # Process the current node (e.g., print it)
            print(current_node, end=" ")

            # Push all unvisited neighbors onto the stack
            # Process neighbors in reverse order to match recursive output order for a simple graph
            for neighbor in reversed(graph.get(current_node, [])):
                if neighbor not in visited:
                    stack.append(neighbor)
    print("\n")

# Example Usage (you can modify the start_node)
print("Testing BFS:")
bfs(graph, 'A')

print("Testing Recursive DFS:")
dfs_recursive(graph, 'A')
print("\n") # Add a newline for better formatting after recursive DFS
```


```python
print("\n") # Add a newline for better formatting after recursive DFS
```


```python
print("Testing Iterative DFS:")
dfs_iterative(graph, 'A')
```

```
Testing BFS:
BFS starting from A:
A B C D E F

Testing Recursive DFS:
A B D E F C

Testing Iterative DFS:
Iterative DFS starting from A:
A B D E F C
```

EXPLANATION:

1. **Graph Representation:**
    - `graph = { ... }`: This dictionary represents the graph using an **adjacency list**. The keys of the dictionary are the nodes of the graph (e.g., 'A', 'B', 'C'), and the value for each key is a list of the nodes that are directly connected to that node (its neighbors). For example, `'A': ['B', 'C']` means node 'A' is connected to nodes 'B' and 'C'. This is an undirected graph because if 'A' is in 'B''s list, 'B' is also in 'A''s list.
2. **Breadth-First Search (BFS):**
    - `def bfs(graph, start_node):`: This function performs a BFS traversal starting from `start_node`.
    - `visited = set()`: A set is used to keep track of nodes that have already been visited to avoid processing them multiple times and prevent infinite loops in graphs with cycles.
    - `queue = deque([start_node])`: A `deque` (double-ended queue) from the `collections` module is used as the queue for BFS. It's efficient for adding and removing elements from both ends. The `start_node` is initially added to the queue.
    - `while queue:`: The loop continues as long as there are nodes in the queue to visit.
    - `current_node = queue.popleft()`: The node at the front of the queue is removed and processed.
    - `if current_node not in visited:`: Checks if the current node has already been visited.
    - `visited.add(current_node)`: If not visited, the node is marked as visited.
    - `print(current_node, end=" ")`: The current node is printed (this is where you would typically process the node).
    - `for neighbor in graph.get(current_node, []):`: It iterates through all the neighbors of the `current_node`. `graph.get(current_node, [])` safely retrieves the list of neighbors, returning an empty list if the node has no neighbors.
    - `if neighbor not in visited:`: If a neighbor hasn't been visited, it's added to the queue to be processed later.
    - `print("\n")`: Adds a newline for formatting after the BFS traversal is complete.
3. **Depth-First Search (DFS - Recursive):**
    - `def dfs_recursive(graph, start_node, visited=None):`: This is the recursive implementation of DFS. It also takes a `visited` set, which is initialized in the first call.
    - `if visited is None: visited = set()`: Initializes the `visited` set if it's the initial call.

- `if visited is None: visited = set()`: Initializes the `visited` set if it's the initial call.
    - `visited.add(start_node)`: Marks the current node as visited.
    - `print(start_node, end=" ")`: Processes the current node (prints it).
    - `for neighbor in graph.get(start_node, []):`: Iterates through the neighbors.
    - `if neighbor not in visited:`: If a neighbor hasn't been visited, the `dfs_recursive` function is called on that neighbor, effectively exploring that branch of the graph.
4. **Depth-First Search (DFS - Iterative):**
    - `def dfs_iterative(graph, start_node):`: This is the iterative implementation of DFS using an explicit stack.
    - `visited = set()`: A set to track visited nodes.
    - `stack = [start_node]`: A list is used as a stack, with the `start_node` initially pushed onto it.
    - `while stack:`: The loop continues as long as there are nodes in the stack.
    - `current_node = stack.pop()`: The node at the top of the stack is removed and processed.
    - `if current_node not in visited:`: Checks if the node has been visited.
    - `visited.add(current_node)`: Marks the node as visited.
    - `print(current_node, end=" ")`: Processes the node (prints it).
    - `for neighbor in reversed(graph.get(current_node, [])):`: It iterates through the neighbors in *reverse* order. This is done so that when the neighbors are pushed onto the stack, they are popped off in the same order as the recursive DFS would visit them (for a simple graph).
    - `if neighbor not in visited:`: If a neighbor hasn't been visited, it's pushed onto the stack.
    - `print("\n")`: Adds a newline for formatting.
5. **Example Usage:**
    - The final part of the code demonstrates how to call the `bfs`, `dfs_recursive`, and `dfs_iterative` functions with the defined `graph` and a starting node ('A').

is code provides a clear implementation of graph representation and two common traversal algorithms