Lab 08: Implementing Breadth First Search and Depth First Search in Python

Objectives:

1. To understand and implement a **tree structure** using the `treelib` library in Python.
2. To learn how to create and use a **queue** using Python lists.
3. To implement the **Breadth-First Search (BFS)** algorithm to search nodes in a tree.

a. Implementing a Tree using `treelib`

Theory:

A **tree** is a hierarchical data structure consisting of nodes.
Each node contains data and links to its child nodes. The **topmost node** is called the **root**.

`treelib` is a Python library that allows you to easily create, visualize, and manage tree structures.

```
# Installing treelib
!pip install treelib
```

```
Collecting treelib
  Downloading treelib-1.8.0-py3-none-any.whl.metadata (3.3 kB)
Requirement already satisfied: six>=1.13.0 in /usr/local/lib/python3.12/dist-packages (from treelib) (1.17.0)
Downloading treelib-1.8.0-py3-none-any.whl (30 kB)
Installing collected packages: treelib
Successfully installed treelib-1.8.0
```

```
# Import Tree class from treelib
from treelib import Node, Tree

# Step 1: Create an empty tree
tree = Tree()

# Step 2: Create root node
tree.create_node("Ahmed", "A")  # Root node (name or tag, identifier)

# Step 3: Add child nodes
tree.create_node("Bilal", "B", parent="A")
tree.create_node("Cathy", "C", parent="A")
tree.create_node("Ducky", "D", parent="B")
tree.create_node("Einstine", "E", parent="B")
tree.create_node("Fawad", "F", parent="C")

# Step 4: Display the tree structure
tree.show()
```

```
Ahmed
├── Bilal
│   ├── Ducky
│   └── Einstine
└── Cathy
    └── Fawad
```

b. Implementing Queue in Python

Theory:

A **queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle.
The element added first is removed first, just like people standing in a line.

```

# Step 1: Create an empty list to represent the queue
queue = []

# Step 2: Add elements to the queue (enqueue)
queue.append('A')
queue.append('B')
queue.append('C')
print("Queue after enqueueing:", queue)

# Step 3: Remove element from the front (dequeue)
removed_item = queue.pop(0)
print("Removed item:", removed_item)
print("Queue after dequeueing:", queue)

# Step 4: Add one more element
queue.append('D')
print("Queue after adding 'D':", queue)
```

```
Queue after enqueueing: ['A', 'B', 'C']
Removed item: A
Queue after dequeueing: ['B', 'C']
Queue after adding 'D': ['B', 'C', 'D']
```

c. Implementing BFS (Breadth-First Search) in Python

Theory:

The **Breadth-First Search (BFS)** algorithm is used to **traverse** or **search** through a tree or graph.
It explores **nodes level by level**, starting from the root node, using a **queue** to keep track of the next node to visit.

Steps:

1. Start from the root node.
2. Visit all its children before moving to the next level.
3. Use a queue to manage nodes to be visited.

```
# Implementing BFS using treelib

from treelib import Node, Tree

# Step 1: Create a tree
tree = Tree()
tree.create_node("A", "A")  # Root
tree.create_node("B", "B", parent="A")
tree.create_node("C", "C", parent="A")
tree.create_node("D", "D", parent="B")
tree.create_node("E", "E", parent="B")
tree.create_node("F", "F", parent="C")

# Step 2: Define BFS function
def bfs_search(tree, target):
    # Initialize queue with root node
    queue = [tree.get_node("A")]

    # Step 3: Start BFS
    while queue:
        node = queue.pop(0)  # Dequeue front node

        # Step 4: Check if current node is target
        if node.tag == target:
            print(f"Node '{target}' found!")
            return True

        # Step 5: Add child nodes to queue
        for child in tree.children(node.identifier):
            queue.append(child)

    print(f"Node '{target}' not found.")
    return False

# Step 6: Call the BFS function
bfs_search(tree, 'A')
```

```
Node 'A' found!
True
```

Lab Task for Submission:

- Task 1: Use the same code and develop code to implement DFS in Python (Hint: use stack)
- Task 2: Extend the Queue Implementation (in code of BFS) - Add a function to check if the queue is empty before performing operation.
- Task 3: Search for Multiple Nodes in BFS Question: Modify your BFS implementation to search for multiple target nodes in a tree or graph.
    - Instead of stopping when the first target is found, continue the search for all target nodes in the structure.
    - Print the paths taken to reach each target node. Hint: Use a list to store multiple target nodes and modify the BFS function to check for each node.

```
Start coding or generate with AI.
```