DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF E&ME, NUST, RAWALPINDI

EC-350 Artificial Intelligence and Decision Support System

LAB MANUAL – 04

Course Instructor: Assoc Prof Dr Arsalan Shaukat

Lab Engineer: Kashaf Raheem

Student Name: **_AMINA QADEER____**

Degree/ Syndicate: **_____CE-42-A_______**

<u>LAB # 4: BREADTH FIRST SEARCH FOR GRAPH AND TREE TRAVERSAL</u>

<u>Lab Objective:</u>

- To implement breadth first search (BFS) algorithm for graph and tree traversals using python.

<u>Hardware/Software required:</u>

Hardware: Desktop/ Notebook Computer
Software Tool: Python 3.10.0

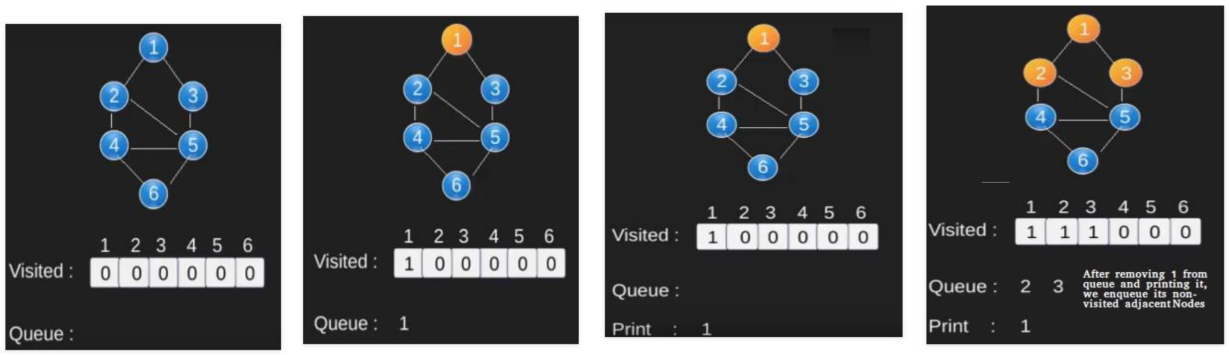
<u>Lab Description:</u>

Breadth first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root or some arbitrary node of a graph and explores the neighbor nodes first, before moving to the next level neighbors.

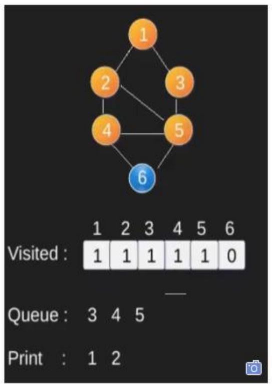
In this lab, we will implement BFS for both tree and graph traversals. Since BFS searches the sibling or neighboring nodes first before moving to the next child nodes so we will employ First In First Out (FIFO) structure in our implementation where sibling nodes will be added first before child nodes and since they are added first so they will be searched earlier as well.

- Start from a given or random node (call it **Current/Working** node)
- Write it in a **queue** (FIFO)
  <u>*Note: While writing in a queue (in case of graphs without cost), you have to write the elements in ascending order or in alphabetical order)*</u>
- Check its neighboring nodes and write them in a Queue while writing the Current Node in **output**.
- Update the Current Node, which is now the **First element** of Queue.
- Traverse till end

Example of BFS applied on graph:



After removing 1 from queue and printing it, we enqueue its non-visited adjacent Nodes

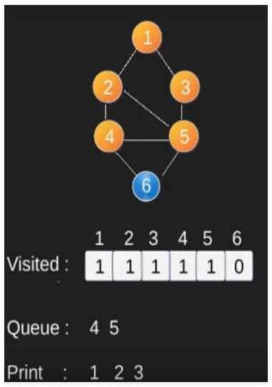


Visited :

1	2	3	4	5	6
1	1	1	1	1	0

Queue : 3 4 5

Print : 1 2

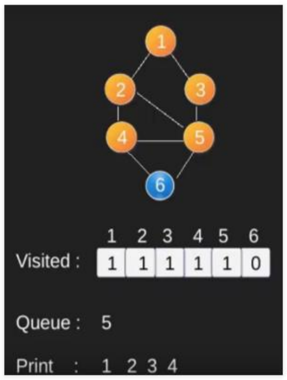


Visited :

1	2	3	4	5	6
1	1	1	1	1	0

Queue : 4 5

Print : 1 2 3

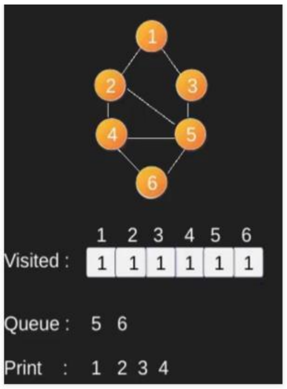


Visited :

1	2	3	4	5	6
1	1	1	1	1	0

Queue : 5

Print : 1 2 3 4

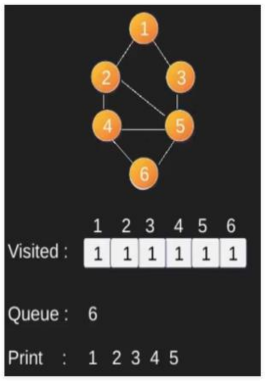


Visited :

1	2	3	4	5	6
1	1	1	1	1	1

Queue : 5 6

Print : 1 2 3 4

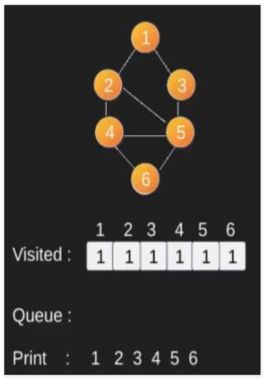


Visited :

1	2	3	4	5	6
1	1	1	1	1	1

Queue : 6

Print : 1 2 3 4 5



Visited :

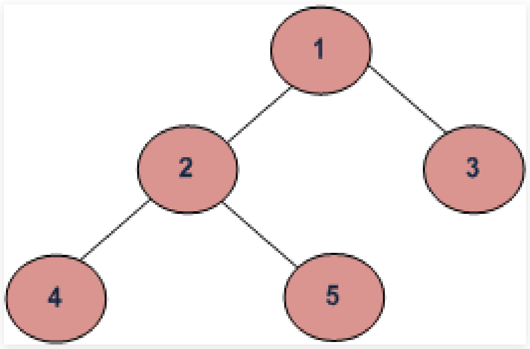
1	2	3	4	5	6
1	1	1	1	1	1

Queue :

Print : 1 2 3 4 5 6

Example of BFS on tree:

- In a tree, BFS is implemented Level wise and we start writing in a queue from left node



```
Breadth First Traversal : 1 2 3 4 5
```

Code for Tree:

Option 1:

```
graph={
    1:[2,3],
    2:[1,4,5],
    3:[1],
    4:[2],
    5:[2]
    }
```

Option 2:

```
class Tree():

    def __init__(self):

        self.left = None

        self.right = None

        self.data = None


root = Tree()

root.data = "root"

root.left = Tree()

root.left.data = "left"

root.right = Tree()

root.right.data = "right"
```

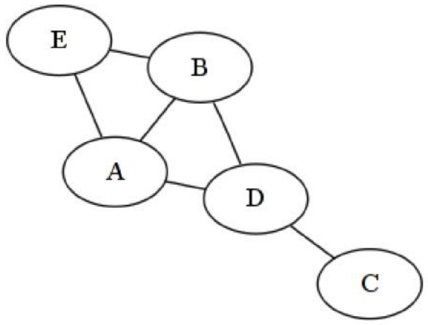
<u>Lab Tasks:</u>

1. **Implement a FIFO data structure in python.**

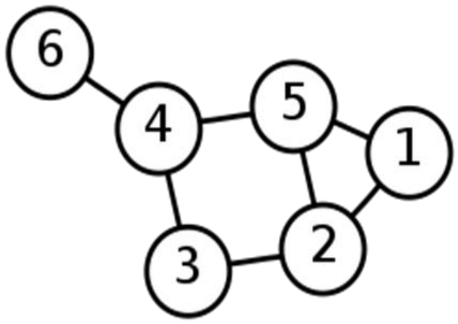
Code:

```
class FIFOQueue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            raise IndexError("The queue is empty.")

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)

# Example usage:
fifo = FIFOQueue()
print("Is the queue empty:", fifo.is_empty())

fifo.enqueue(5)
fifo.enqueue(10)
fifo.enqueue(15)
print("Queue size:", fifo.size())

print("Dequeued item:", fifo.dequeue())
print("Dequeued item:", fifo.dequeue())
print("Dequeued item:", fifo.dequeue())

print("Is the queue empty:", fifo.is_empty())
```

2. Use the implemented FIFO data structure to implement BFS for Graph 1 and Graph 2 in python. The starting nodes for Graph 1 and Graph2 are 'C' and '6', respectively.



Graph 1



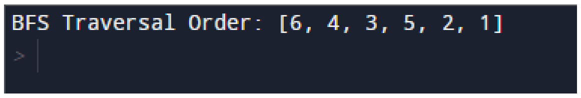
Graph 2

Code:

```
from collections import deque

graph = {
    6: [4],
    4: [6, 3, 5],
    3: [2, 4],
    2: [1, 3, 5],
    1: [2, 5],
    5: []
}

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    traversal_path = []

    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            visited.add(vertex)
            traversal_path.append(vertex)
            for node in graph[vertex]:
                if node not in visited and node
not in queue:
                    queue.append(node)

    return traversal_path
```

```
# Perform BFS on the graph with start
node 6

start_node = 6

result = bfs(graph, start_node)


# Print the result

print("BFS Traversal Order:", result)
```

output:



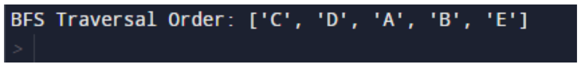
```
BFS Traversal Order: [6, 4, 3, 5, 2, 1]
>
```

Graph 2:

Code:

```
from collections import deque


graph = {

    'C': ['D'],

    'D': ['A', 'B', 'C'],

    'B': ['A', 'E', 'D'],

    'A': ['B', 'E', 'D'],

    'E': []

}


def bfs(graph, start):
    visited = set()

    queue = deque([start])

    traversal_path = []


    while queue:

        vertex = queue.popleft()

        if vertex not in visited:

            visited.add(vertex)

            traversal_path.append(vertex)

            for node in graph[vertex]:

                if node not in visited and node
not in queue:

                    queue.append(node)


    return traversal_path


# Perform BFS on the graph with start
node 'C'

start_node = 'C'

result = bfs(graph, start_node)


# Print the result

print("BFS Traversal Order:", result)
```

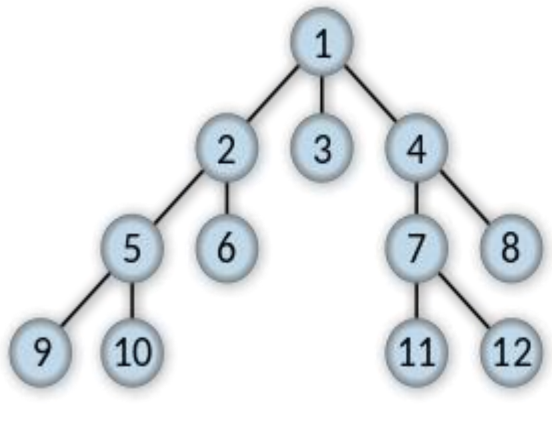
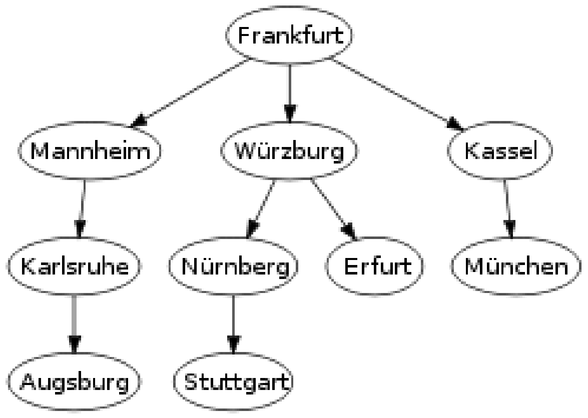
Output:



```
BFS Traversal Order: ['C', 'D', 'A', 'B', 'E']
>
```

3. Implement BFS for Tree 1 and 2 in python. The starting node is '1' for Tree 1 while the starting node is 'Frankfurt' for Tree 2.

Note: for creating Trees you can use the code given above

 **Tree 1**	 Frankfurt Mannheim Würzburg Kassel Karlsruhe Nürnberg Erfurt München Augsburg Stuttgart **Tree 2**

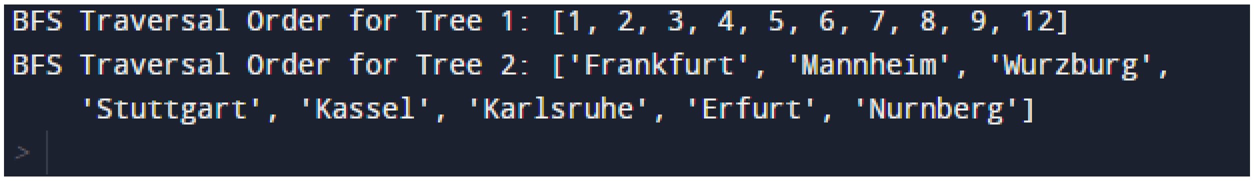
Code:

```
from collections import deque

class Tree:
    def __init__(self):
        self.left = None
        self.middle = None
        self.right = None
        self.data = None

# Tree 1
root1 = Tree()
```

```
root1.data = 1

root1.left = Tree()
root1.left.data = 2
root1.middle = Tree()
root1.middle.data = 3
root1.right = Tree()
root1.right.data = 4

root1.left.left = Tree()
root1.left.left.data = 5
root1.left.middle = Tree()
root1.left.middle.data = 6
root1.right.middle = Tree()
root1.right.middle.data = 7
root1.right.right = Tree()
root1.right.right.data = 8

root1.left.left.left = Tree()
root1.left.left.left.data = 9
root1.left.left.mid = Tree()
root1.left.left.mid.data = 10
root1.right.middle.mid = Tree()
root1.right.middle.mid.data = 11
root1.right.middle.right = Tree()
root1.right.middle.right.data = 12

# BFS for Tree 1
def bfs_tree_1(root):
    if root is None:
        return []

    visited = []
    queue = deque([root])
    while queue:
        node = queue.popleft()
        visited.append(node.data)
        if node.left:
            queue.append(node.left)
        if node.middle:
            queue.append(node.middle)
        if node.right:
            queue.append(node.right)
    return visited

# Perform BFS for Tree 1
```

```
tree_1_result = bfs_tree_1(root1)
print("BFS Traversal Order for Tree 1:", tree_1_result)

# Tree 2
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

# Tree 2 creation
root2 = TreeNode('Frankfurt')
node1 = TreeNode('Mannheim')
node2 = TreeNode('Wurzburg')
node3 = TreeNode('Stuttgart')
node4 = TreeNode('Kassel')
node5 = TreeNode('Karlsruhe')
node6 = TreeNode('Erfurt')
node7 = TreeNode('Nurnberg')

root2.children = [node1, node2, node3]
node1.children = [node4]
node2.children = [node5, node6]
node3.children = [node7]

# BFS for Tree 2
def bfs_tree_2(root):
    if root is None:
        return []

    visited = []
    queue = deque([root])
    while queue:
        node = queue.popleft()
        visited.append(node.value)
        for child in node.children:
            queue.append(child)
    return visited

# Perform BFS for Tree 2
tree_2_result = bfs_tree_2(root2)
print("BFS Traversal Order for Tree 2:", tree_2_result)
```

Output:



```
BFS Traversal Order for Tree 1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 12]
BFS Traversal Order for Tree 2: ['Frankfurt', 'Mannheim', 'Wurzburg',
    'Stuttgart', 'Kassel', 'Karlsruhe', 'Erfurt', 'Nurnberg']
>
```