# HAMDA HUSSAIN(059)

# TASK:06(DOCUMENTATION)

# Breadth First (BFS) Task

This document includes two implementations of the Breadth First Search (BFS) algorithm:  
1. BFS without Queue & without Node Class  
2. BFS with Queue & Node Class  
  
Both versions are implemented in Python and explained in detail below.

## 1. BFS Without Queue & Without Node Class

In this version, BFS is implemented using only a simple list structure and adjacency list representation. No separate queue or Node class is used. A list behaves like a queue by maintaining an index pointer.

### Code:

# BFS Traversal Without Using Queue & Node Class  
  
def bfs\_without\_queue(graph, start):  
 visited = []  
 bfs\_order = []  
  
 # We'll use a simple list to behave like a queue  
 temp = [start]  
 index = 0  
  
 while index < len(temp):  
 vertex = temp[index]  
 index += 1  
  
 if vertex not in visited:  
 visited.append(vertex)  
 bfs\_order.append(vertex)  
  
 for neighbour in graph[vertex]:  
 if neighbour not in visited:  
 temp.append(neighbour)  
   
 return bfs\_order  
  
  
# Example Graph (Adjacency List)  
graph = {  
 'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': ['F'],  
 'F': []  
}  
  
# Calling function  
result = bfs\_without\_queue(graph, 'A')  
print("BFS Traversal (Without Queue & Node):", result)

### Output:

BFS Traversal (Without Queue & Node): ['A', 'B', 'C', 'D', 'E', 'F']

### Explanation:

• A simple adjacency list is used to represent the graph.  
• The 'temp' list stores nodes to be visited.  
• The 'index' variable works like a queue pointer.  
• Each node’s unvisited neighbours are added to the list.  
• Traversal continues until all reachable nodes are visited.

## 2. BFS With Queue & Node Class

In this version, BFS is implemented using Object-Oriented Programming (OOP). A Node class is used to represent each graph node, and Python's built-in deque is used as a queue.

### Code:

# BFS Traversal Using Queue & Node Class  
from collections import deque  
  
class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.neighbours = []  
  
 def add\_neighbour(self, node):  
 self.neighbours.append(node)  
  
  
def bfs\_with\_queue(start\_node):  
 visited = []  
 queue = deque([start\_node])  
  
 while queue:  
 current = queue.popleft()  
  
 if current not in visited:  
 visited.append(current)  
 print(current.value, end=" ")  
  
 for n in current.neighbours:  
 if n not in visited:  
 queue.append(n)  
  
  
# --- Creating Nodes ---  
A = Node('A')  
B = Node('B')  
C = Node('C')  
D = Node('D')  
E = Node('E')  
F = Node('F')  
  
# --- Connecting Nodes (Graph edges) ---  
A.add\_neighbour(B)  
A.add\_neighbour(C)  
B.add\_neighbour(D)  
B.add\_neighbour(E)  
C.add\_neighbour(F)  
E.add\_neighbour(F)  
  
# --- Run BFS ---  
print("BFS Traversal (With Queue & Node):", end=" ")  
bfs\_with\_queue(A)

### Output:

BFS Traversal (With Queue & Node): A B C D E F

### Explanation:

• Each vertex is represented as an object of the Node class.  
• The deque is used as a queue for efficient FIFO operations.  
• Nodes are visited level by level starting from the root node.  
• The program prints the BFS traversal order directly.