# Task 1 & Task 2 Documentation

## Task 1: Breadth-First Search (BFS) Using a Queue

### Objective

To traverse a tree structure level by level using the Breadth-First Search (BFS) algorithm with the help of a queue data structure, and stop traversal when the goal node is found.

### Algorithm Explanation

Breadth-First Search (BFS) explores all nodes at the present depth level before moving on to the next level. A queue is used to store the nodes that need to be explored next. This ensures that nodes are processed in the same order they are discovered.

### Code

tree = {  
 'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': [],  
 'F': ['G'],  
 'G': []  
}  
  
def bfs\_with\_queue(start, goal):  
 visited = []  
 queue = [start]  
  
 while queue:  
 node = queue.pop(0)  
 if node not in visited:  
 visited.append(node)  
 if node == goal:  
 break  
 for child in tree[node]:  
 if child not in visited:  
 queue.append(child)  
 return visited  
  
print(bfs\_with\_queue('A', 'F'))

### Step-by-Step Explanation

* Tree Definition – The tree is represented as a dictionary where each node maps to a list of its child nodes.
* Function bfs\_with\_queue(start, goal) – Initializes visited and queue lists.
* While Loop – Runs until the queue is empty, processing nodes in FIFO order.
* Visiting Nodes – Adds node to visited and checks if it is the goal.
* Adding Children – Enqueues all unvisited child nodes.
* Return Statement – Returns the list of visited nodes.

### Output

['A', 'B', 'C', 'D', 'E', 'F']

## Task 2: Breadth-First Search (BFS) Without Using a Queue

### Objective

To perform BFS traversal of a tree without explicitly using a queue. Instead, two lists are used — one for the current level and one for the next level.

### Algorithm Explanation

This version avoids the traditional queue by manually maintaining current and next levels for traversal. It produces the same result as standard BFS.

### Code

tree = {  
 'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': [],  
 'F': ['G'],  
 'G': []  
}  
  
def bfs\_without\_queue(start, goal):  
 visited = []  
 current\_level = [start]  
  
 while current\_level:  
 next\_level = []  
 for node in current\_level:  
 if node not in visited:  
 visited.append(node)  
 if node == goal:  
 return visited  
 next\_level.extend(tree[node])  
 current\_level = next\_level  
 return visited  
  
print(bfs\_without\_queue('A', 'F'))

### Step-by-Step Explanation

* Tree Definition – Same dictionary structure as the first task.
* Function bfs\_without\_queue(start, goal) – Initializes visited and current\_level lists.
* While Loop – Continues until there are nodes to explore.
* Inner Loop – Visits nodes in current\_level, records children for next\_level.
* Goal Check – Returns when the goal node is found.
* Return Statement – Returns visited nodes after traversal.

### Output

['A', 'B', 'C', 'D', 'E', 'F']

## Comparison of Both Methods

|  |  |  |
| --- | --- | --- |
| Feature | BFS With Queue | BFS Without Queue |
| Data Structure Used | Queue (FIFO) | Two lists (current\_level, next\_level) |
| Implementation Style | Traditional BFS | Level-based BFS |
| Memory Usage | Moderate | Slightly higher due to extra lists |
| Traversal Order | Same | Same |
| Goal Check | Inside main loop | Inside inner loop |