```
#Defining the Graph (Adjacency List) (Based on Fig-1)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['D', 'G'],
    'D': ['C', 'F'],
    'E': ['H'],
    'F': [],
    'G': [],     #GOAL Node
    'H': []
}

print("Graph initialized successfully.")
```

```
Graph initialized successfully.
```

```
#BFS Function
def bfs(graph, start_node, goal_node):
    print(f"\n--- Starting BFS from {start_node} to search for {goal_node} ---")

    # Initialize: Set Open_list = [s]
    # BFS uses FIFO Queue
    open_list = [start_node]
    closed_list = set() # Keeps track of visited nodes

    step_count = 1

    # Loop while OPEN list is not empty
    while open_list:
        print(f"Step {step_count}: OPEN = {open_list}, CLOSED = {closed_list}")

        # Select: Remove the first element (Queue behavior: pop from index 0)
        current_node = open_list.pop(0)

        # Terminate: Check if we found the goal
        if current_node == goal_node:
            print(f"SUCCESS: Goal node '{goal_node}' found at Step {step_count}!")
            return True

        # Add to CLOSED list
        closed_list.add(current_node)

        # Expand: Generate successors
        if current_node in graph:
            children = graph[current_node]
            for child in children:
                # Only add if not in OPEN and not in CLOSED
                if child not in open_list and child not in closed_list:
                    # Add to the END of the list (Queue behavior)
                    open_list.append(child)

        step_count += 1

    print("FAILURE: Goal node not found.")
    return False
```

```
#Run BFS
#Searching for Goal node 'G' starting from 'A'
bfs(graph, 'A', 'G')
```

```
--- Starting BFS from A to search for G ---
Step 1: OPEN = ['A'], CLOSED = set()
Step 2: OPEN = ['B', 'C'], CLOSED = {'A'}
Step 3: OPEN = ['C', 'D', 'E'], CLOSED = {'B', 'A'}
Step 4: OPEN = ['D', 'E', 'G'], CLOSED = {'B', 'C', 'A'}
Step 5: OPEN = ['E', 'G', 'F'], CLOSED = {'D', 'B', 'C', 'A'}
Step 6: OPEN = ['G', 'F', 'H'], CLOSED = {'D', 'B', 'E', 'A', 'C'}
SUCCESS: Goal node 'G' found at Step 6!
True
```

```
#DFS Function
def dfs(graph, start_node, goal_node):
    print(f"\n--- Starting DFS from {start_node} to search for {goal_node} ---")

    # Initialize: Set OPEN = [s]
    # DFS uses LIFO Stack
    open_list = [start_node]
    closed_list = set()

    step_count = 1

    while open_list:
        print(f"Step {step_count}: OPEN = {open_list}, CLOSED = {closed_list}")

        # Select: Remove the LAST element (Stack behavior: pop())
        current_node = open_list.pop()

        # Terminate: Check if we found the goal
        if current_node == goal_node:
            print(f"SUCCESS: Goal node '{goal_node}' found at Step {step_count}!")
            return True

        # Add to CLOSED list
        closed_list.add(current_node)

        # Expand: Generate successors
        if current_node in graph:
            children = graph[current_node]

            # We reverse children before adding to stack.
            # This ensures the "left" child is popped first (LIFO order).
            # If we didn't reverse, 'E' would be popped before 'D' for node 'B'.
            for child in reversed(children):
                if child not in open_list and child not in closed_list:
                    open_list.append(child)

        step_count += 1

    print("FAILURE: Goal node not found.")
    return False
```

```
#Run DFS
#Searching for Goal node 'G' starting from 'A'
dfs(graph, 'A', 'G')
```

```
--- Starting DFS from A to search for G ---
Step 1: OPEN = ['A'], CLOSED = set()
Step 2: OPEN = ['C', 'B'], CLOSED = {'A'}
Step 3: OPEN = ['C', 'E', 'D'], CLOSED = {'B', 'A'}
Step 4: OPEN = ['C', 'E', 'F'], CLOSED = {'D', 'B', 'A'}
Step 5: OPEN = ['C', 'E'], CLOSED = {'D', 'F', 'B', 'A'}
Step 6: OPEN = ['C', 'H'], CLOSED = {'D', 'F', 'B', 'E', 'A'}
Step 7: OPEN = ['C'], CLOSED = {'D', 'F', 'B', 'E', 'A', 'H'}
Step 8: OPEN = ['G'], CLOSED = {'D', 'F', 'B', 'E', 'A', 'H', 'C'}
SUCCESS: Goal node 'G' found at Step 8!
True
```

ॐ श्री गणेशाय नमः

AIML Exp: 01

Date:

Post Lab Questions

A1 Uninformed: Blind search (brute-force) with no domain search knowledge.

Informed: Uses domain knowledge (heuristics) to search estimate closeness to the goal.

A2 Properties of Search Algorithms:

① Completeness: Guranteed to find a sol^n or not

② Optimality: Finds the best/shortest the sol^n or not.

③ Time complexity: Time taken ($O(b^d)$ or $O(b^m)$).

④ Space complexity: Memory required.

A3 ~~What~~ BFS is a blind search that explores the tree layer-by-layer (shallowest nodes first). It uses a Queue (FIFO) data structure.

A4 DFS is a blind search that explores a path to the deepest node before backtracking. It uses a stack (LIFO) data structure.

A5

Date :

AS

	Feature	BFS	DFS
①	Data Structure	Queue (FIFO)	Stack (LIFO)
②	Traversal	Level - by - level	Deepest Path 1ST
③	Memory	High (Exponential)	Low (linear)
④	Optimal	Yes	No

Conclusion : We implement BFS & DFS algorithms. BFS uses a Queue, explores layer - by - layer & gurantees the shortest path but requires high memory. DFS uses stack, explores deep path first, and is memory efficient but doesn't gurantee the shortest path.