

PROMPT 1 ----->

```
import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

record Graph (Map<Integer, List<Integer>> adjacentList, int numberOfNodes) {

    public Graph (int numberOfNodes) {
        this(new HashMap<>(), numberOfNodes);
        for (int node = 0; node < numberOfNodes; node++)
            this.adjacentList.put(node, new ArrayList<>());
    }

    public void addEdge (int node1, int node2) {
        this.adjacentList.get(node1).add(node2);
        this.adjacentList.get(node2).add(node1);
    }

    public List<Integer> traverseByBFS (int sourceNode) {
        List<Integer> result = new ArrayList<>();

        boolean[] visited = new boolean[this.numberOfNodes]; // Keeps track of visited node(s).
        Queue<Integer> nodeQueue = new LinkedList<>();
        nodeQueue.add(sourceNode); // Adding the source node to the freshly created queue.

        while (!nodeQueue.isEmpty()) {
            int node = nodeQueue.remove();
            result.add(node);
            visited[node] = true;
            List<Integer> neighbourNodeList = this.adjacentList.get(node);
            for (int neighbourNode : neighbourNodeList) {
                if (visited[neighbourNode]) continue;
            }
        }
    }
}
```

```

        nodeQueue.add(neighbourNode);
    }
}

return result;
}

public void traverseByDFS (int node, boolean[] visited, List<Integer> result) /* DFS helper function. */ {
    // NOTE - Only pass unvisited node.
    result.add(node);
    visited[node] = true;
    List<Integer> neighbourNodeList = this.adjacentList.get(node);
    for (int neighbourNode : neighbourNodeList) {
        if (visited[neighbourNode]) continue;
        this.traverseByDFS(neighbourNode, visited, result);
    }
}

public List<Integer> traverseByDFS (int sourceNode) /* DFS calling function. */ {
    List<Integer> result = new ArrayList<>();
    boolean[] visited = new boolean[this.numberOfNodes]; // Keeps track of visited node(s).
    this.traverseByDFS(sourceNode, visited, result);
    return result;
}
}

```

/* Above is the code for traverseing through the graph using BFS (Bredth First Search) and DFS (Depth First Search) approach. The functions 'traverseBFS()' and 'traverseDFS()' represents traversing through BFS and DFS approach respectively. Can you please check and confirm the correctness of the functions? I presonally believe there is an error in BFS approach and the problem is the node queue might conatin duplicate visited / unvisited nodes. */

DeepSeek AI Response 1 ----->

Let me analyze both the BFS and DFS implementations in your `Graph` class.

BFS Implementation Analysis:

The current BFS implementation has a potential issue with duplicate nodes in the queue. Here's why:

1. When you visit a node, you mark it as visited **after** removing it from the queue.
2. However, you add all its neighbors to the queue without checking if they're already in the queue.
3. This means a node could be added to the queue multiple times before it's actually processed.

Here's how to fix it:

```
```java
public List<Integer> traverseByBFS(int sourceNode) {
 List<Integer> result = new ArrayList<>();
 boolean[] visited = new boolean[this.numberOfNodes];
 Queue<Integer> nodeQueue = new LinkedList<>();

 // Mark source as visited when adding to queue
 nodeQueue.add(sourceNode);
 visited[sourceNode] = true;

 while (!nodeQueue.isEmpty()) {
 int node = nodeQueue.remove();
 result.add(node);
 List<Integer> neighbourNodeList = this.adjacentList.get(node);
 for (int neighbourNode : neighbourNodeList) {
 if (!visited[neighbourNode]) {
 visited[neighbourNode] = true; // Mark visited when adding to queue
 nodeQueue.add(neighbourNode);
 }
 }
 }
 return result;
}
```
```

Key changes:

1. Mark nodes as visited when they're added to the queue, not when they're removed
2. This prevents the same node from being added multiple times

DFS Implementation Analysis:

The DFS implementation looks correct. It:

1. Marks nodes as visited before processing their neighbors
2. Uses recursion to explore depth-first
3. Properly handles the visited array

Testing Recommendations:

You should test both implementations with various graphs, including:

1. A simple linear graph
2. A graph with cycles
3. A disconnected graph
4. A complete graph (where every node is connected to every other node)

For example:

```
```java
public static void main(String[] args) {
 Graph g = new Graph(4);
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 3);

 System.out.println("BFS: " + g.traverseByBFS(0));
 System.out.println("DFS: " + g.traverseByDFS(0));
}
```
```

The fixed BFS implementation will work correctly for all these cases, while the original might produce duplicates or incorrect orders in some cases.

PROMPT 2 ----->

```
import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;
```

```

record Graph (Map<Integer, List<Integer>> adjacentList, int numberOfNodes) {

    public Graph (int numberOfNodes) {
        this(new HashMap<>(), numberOfNodes);
        for (int node = 0; node < numberOfNodes; node++)
            this.adjacentList.put(node, new ArrayList<>());
    }

    public void addEdge (int node1, int node2) {
        this.adjacentList.get(node1).add(node2);
        this.adjacentList.get(node2).add(node1);
    }

    public List<Integer> traverseByBFS (int sourceNode) {
        List<Integer> result = new ArrayList<>();

        boolean[] visited = new boolean[this.numberOfNodes]; // Keeps track of visited node(s).
        Queue<Integer> nodeQueue = new LinkedList<>();
        nodeQueue.add(sourceNode); // Adding the source node to the freshly created queue.
        visited[sourceNode] = true; // Marking the source node as visited. (New Line 1.)

        while (!nodeQueue.isEmpty()) {
            int node = nodeQueue.remove();
            result.add(node);
            //visited[node] = true; // Never mark here after removing the node from the queue.
            List<Integer> neighbourNodeList = this.adjacentList.get(node);
            for (int neighbourNode : neighbourNodeList) {
                if (visited[neighbourNode]) continue;
                nodeQueue.add(neighbourNode);
                visited[neighbourNode] = true; // Marking the neighbour node as visited. (New Line 2.)
            }
        }

        return result;
    }

    public void traverseByDFS (int node, boolean[] visited, List<Integer> result) /* DFS helper function. */ {
        // NOTE - Only pass unvisited node.
    }
}

```

```

        result.add(node);
        visited[node] = true;
        List<Integer> neighbourNodeList = this.adjacentList.get(node);
        for (int neighbourNode : neighbourNodeList) {
            if (visited[neighbourNode]) continue;
            this.traverseByDFS(neighbourNode, visited, result);
        }
    }

    public List<Integer> traverseByDFS (int sourceNode) /* DFS calling function. */ {
        List<Integer> result = new ArrayList<>();
        boolean[] visited = new boolean[this.numberOfNodes]; // Keeps track of visited node(s).
        this.traverseByDFS(sourceNode, visited, result);
        return result;
    }
}

public class CommonGraphTraversal {

    public static void main (String[] args) {

        System.out.println("\n--- Traversing Through The Graph ---\n");
        System.out.println("[ NOTE - Entering any negative integer at any point of time while taking input will close the input. ]");

        Scanner keyboardInput = new Scanner(System.in);

        System.out.print("\nEnter the number of nodes in your graph: ");
        int numberOfNodes = keyboardInput.nextInt();
        if (numberOfNodes < 0) return;

        System.out.println("\nAdd edge(s) to the graph ->");
        List<List<Integer>> edgeList = new ArrayList<>();
        while (true) {
            System.out.print("Enter a new edge [ node_1 node_2 ]: ");
            int node1 = keyboardInput.nextInt();
            if (node1 < 0) break;

```

```

        int node2 = keyboardInput.nextInt();
        if (node2 < 0) break;
        edgeList.add(List.of(node1, node2));
    }

    System.out.print("\nEnter the source node of your graph: ");
    int sourceNode = keyboardInput.nextInt();
    if (sourceNode < 0) return;

    Graph graph = new Graph(numberOfNodes); // Creating a new graph ...

    // Adding edge(s) to the graph ...
    for (List<Integer> edge : edgeList)
        graph.addEdge(edge.get(0), edge.get(1));

    // BFS (Bredth First Search) traversal of the graph ...
    System.out.println("\nBFS (Bredth First Search) traversal of the graph: " +
graph.traverseByBFS(sourceNode));

    // DFS (Depeth First Search) traversal of the graph ...
    System.out.println("\nDFS (Depeth First Search) traversal of the graph: " +
graph.traverseByDFS(sourceNode));

    System.out.println("\n");
}

}

/ * This is the revised code. Can you please check and confirm the correctness of both the functions? * /

```

DeepSeek AI Response 2 ----->

Your revised code looks correct! Both the BFS and DFS implementations now properly handle node visitation and traversal. Here's the confirmation:

BFS Implementation (Corrected):

☒ ****Fixes Applied:****

1. ****Mark `sourceNode` as visited when adding to the queue**** (`visited[sourceNode] = true`).

2. **Mark neighbors as visited when adding them to the queue** (`visited[neighbourNode] = true`).
3. **Removed redundant `visited[node] = true` after removal from the queue** (this was the root cause of duplicates).

☑ **Why it works:**

- Ensures no node is added to the queue more than once.
- Maintains the correct BFS order (level-by-level traversal).

DFS Implementation (Already Correct):

☑ **Correct Behavior:**

1. Marks nodes as visited **before** recursively exploring neighbors (`visited[node] = true`).
2. Only processes unvisited neighbors (`if (visited[neighbourNode]) continue`).
3. Naturally handles cycles and disconnected components.

Test Case Verification:

Your `main` method provides a good way to test the implementations. Here's what happens with a sample input:

Input:

...

Number of nodes: 4

Edges: [0,1], [0,2], [1,2], [2,3]

Source node: 0

...

Expected Output:

- **BFS:** `[0, 1, 2, 3]` (or `[0, 2, 1, 3]` depending on edge order, since 1 and 2 are at the same level).
- **DFS:** `[0, 1, 2, 3]` (or `[0, 2, 3, 1]` depending on adjacency list ordering).

Edge Cases to Test:

1. **Disconnected Graph:**

- Nodes: 4, Edges: [0,1], [2,3], Source: 0
- BFS: `[0, 1]` (only reaches nodes connected to 0).
- DFS: `[0, 1]` (same as BFS).

2. **Cycle:**

- Nodes: 3, Edges: [0,1], [1,2], [2,0], Source: 0
- BFS: `[0, 1, 2]` (no duplicates).
- DFS: `[0, 1, 2]` (no infinite recursion).

3. ****Single Node:****

- Nodes: 1, Edges: [], Source: 0
- BFS/DFS: `[0]`.

Final Notes:

- Your BFS fix ensures ****no duplicates**** in the queue and proper traversal.
- DFS was already correct and remains unchanged.
- The `main` method provides a good interactive way to test the code.

****Conclusion:**** Both implementations are now correct and handle all standard graph traversal cases properly.
Well done! 🚀