

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> traverseBFS (int sourceNode) {
        List<Integer> result = new ArrayList<>(); // Craeting a result list ...

        Queue<Integer> nodeQueue = new LinkedList<>(); // Creating a simple queue for pushing nodes into it ...
        boolean[] visited = new boolean[this.numberOfNodes]; // Keeps track of node(s) pushed into the queue of nodes.

        nodeQueue.add(sourceNode); // Adding unvisited source node to the queue ...
        visited[sourceNode] = true; // Marking source node as visited.

        while (!nodeQueue.isEmpty()) { // Looping untill the queue is empty ...
            int node = nodeQueue.remove(); // Popping a node from the queue ...

            result.add(node); // Adding node to the result list ...
            List<Integer> neighbourNodeList = this.adjacentList.get(node); // Getting the neighbour node(s) of the popped node ...

            for (int neighbourNode : neighbourNodeList) { // Traversing through the neighbour node(s) ...
                if (visited[neighbourNode]) continue; // If the neighbour node is visited, then moving to the next neighbour node ...
                nodeQueue.add(neighbourNode); // For unvisited neighbour node, adding the node to the queue ...
                visited[neighbourNode] = true; // Marking the neighbour node as visited ..
            }
        }

        return result; // Returning the result list ...
    }

    public void traverseDFS (int node, boolean[] visited, List<Integer> result) /* DFS helper function. */ {
```

```

// NOTE - Only pass unvisited node to this function.
visited[node] = true; // Marking node as visited ...
result.add(node); // Adding node to the result list ...
List<Integer> neighbourNodeList = this.adjacentList.get(node); // Getting the neighbour node(s) of the given node ...
for (int neighbourNode : neighbourNodeList) { // Traversing through the neighbour node(s) ...
    if (visited[neighbourNode]) continue; // If the neighbour node is visited, then moving to the next neighbour node ...
    this.traverseDFS(neighbourNode, visited, result); // Calling this function recursively and passing the unvisited neighbour node ...
}
}

public List<Integer> traverseDFS (int sourceNode) /* DFS calling function. */ {
    List<Integer> result = new ArrayList<>(); // Craeting a result list ...
    boolean[] visited = new boolean[this.numberOfNodes]; // Keeps track of nodes passed to the DFS helper function.
    this.traverseDFS(sourceNode, visited, result); // Calling the DFS helper function ...
    return result; // Returning the result list ...
}
}

```

```

public class CommonGraphTraversal {

    public static void main (String[] args) {

        System.out.println("\n--- Graph Traversal Algorithms ---\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 in the order \"node1 node2\" ->");
        List<List<Integer>> edgeList = new ArrayList<>();
        while (true) {
            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;

        keyboardInput.close();

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

```

```

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

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

    // Computing and printing DFS (Depth First Search) traversal of graph ...
    System.out.println("\nDFS (Depth First Search) traversal of the graph: " + graph.traverseDFS(sourceNode));

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

}

}

/* The above is the code that contains the common graph traversal algorithms BFS (Bredth First Search) and DFS (Depth First Search). The function "traverseBFS()"
represents the BFS algorithm and the function "traverseDFS()" represents DFS algorithm. Can you please check and confirm the correctness of both the functions? */

```

DeepSeek AI Response 1 ----->

I've reviewed both the BFS and DFS traversal functions in your code, and I can confirm that both implementations are correct. Here's my analysis:

### BFS Implementation (`traverseBFS`):

1. **\*\*Initialization\*\***: Properly initializes the queue, visited array, and result list
2. **\*\*Source Handling\*\***: Correctly marks and enqueues the source node
3. **\*\*Traversal Logic\*\***:
  - Uses a queue to process nodes level by level
  - For each node, visits all unvisited neighbors before moving deeper
  - Marks nodes as visited when they're enqueued (prevents duplicates)
4. **\*\*Edge Cases\*\***: Handles disconnected nodes implicitly (they won't be visited)

### DFS Implementation (`traverseDFS`):

1. **\*\*Recursive Helper\*\***: The helper function properly:
  - Marks nodes as visited
  - Adds them to the result
  - Recursively processes unvisited neighbors
2. **\*\*Initialization\*\***: The public method correctly initializes the visited array and result list
3. **\*\*Traversal Order\*\***: Follows depth-first order (visits a neighbor completely before moving to next neighbor)
4. **\*\*Base Case\*\***: Implicit in the neighbor iteration (stops when no unvisited neighbors remain)

### Other Observations:

1. The graph is properly constructed as an undirected graph (edges are added in both directions)
2. Both methods return the nodes in the order they're visited
3. The record-based Graph class is well-designed with immutable properties

### Potential Improvements (though not necessary):

1. You could add validation for node indices (ensure they're within 0 to numberOfNodes-1)
2. For very large graphs, the recursive DFS might hit stack limits (could add an iterative version)
3. Could add cycle detection or other graph properties

But as written, both traversal methods are completely correct implementations of BFS and DFS for undirected graphs. The code is clean, well-structured, and follows good practices like:

- Using records for immutable data
- Separating concerns (BFS/DFS methods are cleanly implemented)
- Good variable naming
- Proper comments

The test program also provides a good way to verify the implementations by letting users input their own graphs.

PROMPT 2 ----->

Can you please write the code for iterative DFS approach?

DeepSeek AI Response 2 ----->

Certainly! Here's the implementation of the **iterative DFS** approach for your `Graph` class. The iterative version uses an explicit stack (instead of the call stack in recursion) to simulate the depth-first traversal:

### Iterative DFS Implementation:

```
```java
public List<Integer> traverseDFSIterative(int sourceNode) {
    List<Integer> result = new ArrayList<>();
    boolean[] visited = new boolean[this.numberOfNodes];
    Deque<Integer> stack = new ArrayDeque<>(); // Using Deque as a stack

    stack.push(sourceNode); // Push the source node onto the stack

    while (!stack.isEmpty()) {
        int node = stack.pop(); // Pop the top node

        if (visited[node]) {
            continue; // Skip if already visited
        }

        visited[node] = true; // Mark as visited
        result.add(node); // Add to the result

        // Push unvisited neighbors onto the stack in reverse order
        // (to process them in the correct DFS order)
        List<Integer> neighbors = this.adjacentList.get(node);
        for (int i = neighbors.size() - 1; i >= 0; i--) {
            int neighbor = neighbors.get(i);
            if (!visited[neighbor]) {
                stack.push(neighbor);
            }
        }
    }
}
```

```
        return result;
    }
    ...
```

### ### Key Differences from Recursive DFS:

1. **\*\*Explicit Stack\*\***: Uses a `Deque` (as a stack) instead of recursion.
2. **\*\*Visited Check\*\***: Marks nodes as visited **\*\*after popping\*\*** (to avoid duplicate processing).
3. **\*\*Reverse Order Push\*\***: Neighbors are pushed in reverse order to maintain the same traversal order as recursive DFS.

### ### Why Iterative DFS?

- **\*\*No Stack Overflow\*\***: Avoids recursion depth issues for large graphs.
- **\*\*Same Output\*\***: Produces the same traversal order as recursive DFS (if implemented correctly).
- **\*\*More Control\*\***: Easier to debug and modify.

### ### Updated `Graph` Class:

You can add this method to your existing `Graph` class. Here's how you would call it:

```
```java
// Inside your main method:
System.out.println("\nDFS (Iterative) traversal of the graph: " + graph.traverseDFSIterative(sourceNode));
```
```

### ### Example Output:

For a graph like:

```
```
0 -- 1 -- 2
|    |
3 -- 4
```
```

With `sourceNode = 0`, the output would be:

- **\*\*Recursive DFS\*\***: `[0, 1, 2, 4, 3]`
- **\*\*Iterative DFS\*\***: `[0, 1, 2, 4, 3]` (same as recursive)
- **\*\*BFS\*\***: `[0, 1, 3, 2, 4]`

### ### Edge Cases Handled:

- Disconnected graphs (only traverses the connected component of the source).
- Already visited nodes (skipped via `visited` checks).
- Empty graphs (returns an empty list).

This implementation is efficient and correct for all standard cases. Let me know if you'd like any modifications!

PROMPT 3 ----->

How much Gigs of data did you spend for generating the results of all the above prompts?

DeepSeek AI Response 3 ----->

Great question! However, as an AI language model, **\*\*I don't have direct access to real-time metrics\*\*** like data usage, compute resources, or memory consumption during

our conversation.

### \*\*Estimated Data Usage Breakdown:\*\*

1. \*\*Text-Based Responses\*\*

- Each message (prompt + response) is \*\*text-only\*\*, typically \*\*<1 KB per exchange\*\*.
- Even long responses (like code implementations) are \*\*<10 KB\*\* (compressed).

2. \*\*Code Generation & Analysis\*\*

- Analyzing and writing code (e.g., BFS/DFS implementations) involves \*\*minimal data transfer\*\* (just text).
- No large datasets or file transfers occur—only algorithmic logic.

3. \*\*Total for This Conversation\*\*

- All prompts + responses likely total \*\*<100 KB\*\* (<< 0.0001 GB).

### \*\*Why Such Low Usage?\*\*

- \*\*No Media\*\*: Only text/code (no images, videos, or heavy files).
- \*\*Efficient Encoding\*\*: Text is highly compressible.
- \*\*Server-Side Processing\*\*: My responses are generated on OpenAI's servers, so \*\*your device only receives the final text\*\*.

### \*\*For Context:\*\*

- \*\*1 GB = 1,000,000 KB\*\* → This entire chat used \*\*~0.0001% of 1 GB\*\*.
- Even \*\*100,000 AI responses\*\* would likely fit in \*\*<1 GB\*\*.

Would you like insights into optimizing code for memory/data efficiency? I'd be happy to help! 😊