The algorithm starts from a root node and examines each **branch as FAR as feasible** before retracing. So the basic idea is to start at the root or any arbitrary node and mark it, then advance to the next unmarked node and repeat until there are no more unmarked nodes.

```
class Graph {
    constructor() {
        this.adjacencyList = {};
    }

    addVertex(vertex) {
        if (!this.adjacencyList[vertex]) {
            this.adjacencyList[vertex] = [];
        }
    }

    addEdge(vertex1, vertex2) {
        this.adjacencyList[vertex1].push(vertex2);
        this.adjacencyList[vertex2].push(vertex1);
    }

    dfs(startingVertex) {
        const visited = {};
        const stack = [startingVertex]; // uses stack which is different from queue used in BFS
        const result = [];

        visited[startingVertex] = true;

        while (stack.length) {
            const currentVertex = stack.pop();
            result.push(currentVertex);

            this.adjacencyList[currentVertex].forEach(neighbor => {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    stack.push(neighbor);
                }
            });
        }

        return result;
    }
}

// Example usage:
const graph = new Graph();

graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addVertex("E");
graph.addVertex("F");
```

```
graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");
graph.addEdge("C", "E");
graph.addEdge("D", "E");
graph.addEdge("D", "F");
graph.addEdge("E", "F");

console.log("DFS result:", graph.dfs("A"));
```