Depth-First Search

Pseudocode:

```
dfsVisit(Node, Target)
     Node.state = VISITING
     if (Node.data = Target)
         return true;

     For each neighbor:
         if Neighbor is UNVISITED
             perform dfsVisit on Neighbor
             if dfsVisit returned true, Target found, return true and exit

     Node.state = VISITED
     Target not found, return false

dfs(Graph, Target)
     for all nodes in Graph:
         if node is UNVISITED, perform dfsVisit
         if dfsVisit returned true, Target found, return true
     Target not found, return false
```

Test Cases:
Edge Cases: empty graph, null graph
Base Cases: Graph with 1 node, graph with 2 nodes, graph with 2 disconnected nodes
Regular Cases: Target in graph/not in graph, Target first element/deep in graph

Time Complexity: O(V + E), where V is Vertices, and E is Edges

Space Complexity: O(V) in worse case.
If graph is a chain, then we take O(V) space on recursion stack. Also, we need O(V) space to store the State of each Node.

```
public static boolean dfs(Graph graph, int target) {
    for(Node node : graph.getNodes()) {
        if (node.getState() == State.UNVISITED && dfsVisit(node, target))
            return true;
    }
    return false;
}

public static boolean dfsVisit(Node node, int target) {
    node.setState(State.VISITING);

    if (node.getData() == target)
        return true;
    for (Node neighbor: node.getNeighbors()) {
        if (neighbor.getState() == State.UNVISITED && dfsVisit(neighbor,
target))
            return true;
    }

    node.setState(State.VISITED);
    return false;
}

/*
 * Helper Code. Ask interviewer before implementing.
 */
public enum State {
    UNVISITED,
    VISITING,
    VISITED;
}

public class Graph {
    List<Node> nodes;

    public Graph(List<Node> nodes) {
        super();
        this.nodes = nodes;
    }

    public void addNode(Node node) {
        nodes.add(node);
    }

    public List<Node> getNodes() {
        return nodes;
    }
}

public class Node {
    List<Node> neighbors;
    int data;
    State state;

    public Node(int data) {
        super();
        this.data = data;
        state = State.UNVISITED;
        neighbors = new ArrayList<Node>();
    }
    public int getData() {
        return data;
    }

    public void setData(int data) {
        this.data = data;
    }

    public void setState(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }

    public void addNeighbor(Node node) {
        neighbors.add(node);
    }

    public List<Node> getNeighbors() {
        return neighbors;
    }
}
```

Breadth-First Search

Pseudocode:

```
bfsVisit(StartNode, Target)
     Q = empty queue
     Q.enqueue(StartNode)
     StartNode.State = VISITING

     while (Q is not empty)
         Node = q.dequeue()

         If Node = Target // Process Node
             return true

         For all Node's neighbors:
             if neighbor is UNVISITED
                 add it to the back of Q, set its State to VISITING

         Node.State = VISITED

    Reached End, not found, return false
```

Test Cases:
Edge Cases: Empty Graph
Base Cases: Single Node, 2 Nodes, 2 Nodes unconnected
Regular Cases: Target present/not present, Multiple connected components

Time Complexity: O(V + E)

Space Complexity: O(V)
We can store at most V nodes on the Queue, and we also use V space to store the state of each node.

```
public static boolean bfs(Graph graph, int target) {
    for (Node node : graph.getNodes()) {
        if (node.getState() == State.UNVISITED && bfsVisit(node, target))
            return true;
    }
    return false;
}

public static boolean bfsVisit(Node start, int target) {
    Queue<Node> q = new LinkedList<Node>();
    q.add(start);
    start.setState(State.VISITING);

    while(!q.isEmpty()) {
        Node current = q.remove();
        if (current.getData() == target)
            return true;

        for (Node neighbor : current.getNeighbors()) {
            if (neighbor.getState() == State.UNVISITED) {
                q.add(neighbor);
                neighbor.setState(State.VISITING);
            }
        }
        current.setState(State.VISITED);
    }

    return false;
}





/*
 * Helper Code. Ask the interviewer if they want you to implement.
 */

public enum State {
    UNVISITED,
    VISITING,
    VISITED;
}

public class Graph {
    List<Node> nodes;
    public Graph(List<Node> nodes) {
        super();
        this.nodes = nodes;
    }

    public void addNode(Node node) {
        nodes.add(node);
    }

    public List<Node> getNodes() {
        return nodes;
    }
}

public class Node {
    List<Node> neighbors;
    int data;
    State state;

    public Node(int data) {
        super();
        this.data = data;
        state = State.UNVISITED;
        neighbors = new ArrayList<Node>();
    }

    public int getData() {
        return data;
    }

    public void setData(int data) {
        this.data = data;
    }

    public void setState(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }

    public void addNeighbor(Node node) {
        neighbors.add(node);
    }

    public List<Node> getNeighbors() {
        return neighbors;
```

```
          }
}
```