Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

* depth first search algorithm

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

// A class to store a graph edge

class Edge

{

    int source, dest;

    public Edge(int source, int dest)

    {

        this.source = source;

        this.dest = dest;

    }

}

// A class to represent a graph object

class Graph

{

    // A list of lists to represent an adjacency list

    List<List<Integer>> adjList = null;

    // Constructor

    Graph(List<Edge> edges, int n)

    {

        adjList = new ArrayList<>();

        for (int i = 0; i < n; i++) {

            adjList.add(new ArrayList<>());

        }

        // add edges to the undirected graph

        for (Edge edge: edges)

        {

            int src = edge.source;

            int dest = edge.dest;

            adjList.get(src).add(dest);

            adjList.get(dest).add(src);

        }

    }

}

class Main

{

    // Function to perform DFS traversal on the graph on a graph

    public static void DFS(Graph graph, int v, boolean[] discovered)

    {

        // mark the current node as discovered

        discovered[v] = true;

        // print the current node

        System.out.print(v + " ");

        // do for every edge (v, u)

        for (int u: graph.adjList.get(v))

        {

            // if `u` is not yet discovered

            if (!discovered[u]) {

                DFS(graph, u, discovered);

            }

        }

    }

    public static void main(String[] args)

    {

        // List of graph edges as per the above diagram

        List<Edge> edges = Arrays.asList(

                // Notice that node 0 is unconnected

                new Edge(1, 2), new Edge(1, 7), new Edge(1, 8), new Edge(2, 3),

                new Edge(2, 6), new Edge(3, 4), new Edge(3, 5), new Edge(8, 9),

                new Edge(8, 12), new Edge(9, 10), new Edge(9, 11)

            );

        // total number of nodes in the graph (labelled from 0 to 12)

        int n = 13;

// build a graph from the given edges

        Graph graph = new Graph(edges, n);

        // to keep track of whether a vertex is discovered or not

        boolean[] discovered = new boolean[n];

        // Perform DFS traversal from all undiscovered nodes to

        // cover all connected components of a graph

        for (int i = 0; i < n; i++)

        {  if (!discovered[i]) {

                DFS(graph, i, discovered);

            }   }  } }

**Output :**

0 1 2 3 4 5 6 7 8 9 10 11 12

* Breadth First Search

import java.util.\*;

// A class to store a graph edge

class Edge

{

    int source, dest;

    public Edge(int source, int dest)

    {

        this.source = source;

        this.dest = dest;

    }

}

// A class to represent a graph object

class Graph

{

    // A list of lists to represent an adjacency list

    List<List<Integer>> adjList = null;

    // Constructor

    Graph(List<Edge> edges, int n)

    {

        adjList = new ArrayList<>();

        for (int i = 0; i < n; i++) {

            adjList.add(new ArrayList<>());

        }

        // add edges to the undirected graph

        for (Edge edge: edges)

        {

            int src = edge.source;

            int dest = edge.dest;

            adjList.get(src).add(dest);

            adjList.get(dest).add(src);

        }

    }

}

class Main

{

    // Perform BFS recursively on the graph

    public static void recursiveBFS(Graph graph, Queue<Integer> q,

                                    boolean[] discovered)

    {

        if (q.isEmpty()) {

            return;

        }

        // dequeue front node and print it

        int v = q.poll();

        System.out.print(v + " ");

        // do for every edge (v, u)

        for (int u: graph.adjList.get(v))

        {

            if (!discovered[u])

            {

                // mark it as discovered and enqueue it

                discovered[u] = true;

                q.add(u);

            }

        }

        recursiveBFS(graph, q, discovered);

    }

    public static void main(String[] args)

    {

        // List of graph edges as per the above diagram

        List<Edge> edges = Arrays.asList(

                new Edge(1, 2), new Edge(1, 3), new Edge(1, 4), new Edge(2, 5),

                new Edge(2, 6), new Edge(5, 9), new Edge(5, 10), new Edge(4, 7),

                new Edge(4, 8), new Edge(7, 11), new Edge(7, 12)

                // vertex 0, 13, and 14 are single nodes

        );

        // total number of nodes in the graph (labelled from 0 to 14)

        int n = 15;

        // build a graph from the given edges

        Graph graph = new Graph(edges, n);

        // to keep track of whether a vertex is discovered or not

        boolean[] discovered = new boolean[n];

        // create a queue for doing BFS

        Queue<Integer> q = new ArrayDeque<>();

        // Perform BFS traversal from all undiscovered nodes to

        // cover all connected components of a graph

        for (int i = 0; i < n; i++)

        {

            if (discovered[i] == false)

            {

                // mark the source vertex as discovered

                discovered[i] = true;

                // enqueue source vertex

                q.add(i);

                // start BFS traversal from vertex `i`

                recursiveBFS(graph, q, discovered);

            }

        }

    }

}

Output:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14