**DFS JAVA PROGRAM**

// Java program to print DFS

// traversal from a given given

// graph

import java.io.\*;

import java.util.\*;

// This class represents a

// directed graph using adjacency

// list representation

class Graph {

    private int V; // No. of vertices

    // Array  of lists for

    // Adjacency List Representation

    private LinkedList<Integer> adj[];

    // Constructor

    @SuppressWarnings("unchecked") Graph(int v)

    {

        V = v;

        adj = new LinkedList[v];

        for (int i = 0; i < v; ++i)

            adj[i] = new LinkedList();

    }

    // Function to add an edge into the graph

    void addEdge(int v, int w)

    {

        adj[v].add(w); // Add w to v's list.

    }

    // A function used by DFS

    void DFSUtil(int v, boolean visited[])

    {

        // Mark the current node as visited and print it

        visited[v] = true;

        System.out.print(v + " ");

        // Recur for all the vertices adjacent to this

        // vertex

        Iterator<Integer> i = adj[v].listIterator();

        while (i.hasNext()) {

            int n = i.next();

            if (!visited[n])

                DFSUtil(n, visited);

        }

    }

    // The function to do DFS traversal. It uses recursive

    // DFSUtil()

    void DFS()

    {

        // Mark all the vertices as not visited(set as

        // false by default in java)

        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS

        // traversal starting from all vertices one by one

        for (int i = 0; i < V; ++i)

            if (visited[i] == false)

                DFSUtil(i, visited);

    }

    // Driver Code

    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, 0);

        g.addEdge(2, 3);

        g.addEdge(3, 3);

        System.out.println(

            "Following is Depth First Traversal");

        g.DFS();

    }

}

OUTPUT

Following is Depth First Traversal

0 1 2 3

**BFS**

// Java program to print BFS traversal from a given source vertex.

// BFS(int s) traverses vertices reachable from s.

import java.io.\*;

import java.util.\*;

// This class represents a directed graph using adjacency list

// representation

class Graph

{

    private int V;   // No. of vertices

    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor

    Graph(int v)

    {

        V = v;

        adj = new LinkedList[v];

        for (int i=0; i<v; ++i)

            adj[i] = new LinkedList();

    }

    // Function to add an edge into the graph

    void addEdge(int v,int w)

    {

        adj[v].add(w);

    }

    // prints BFS traversal from a given source s

    void BFS(int s)

    {

        // Mark all the vertices as not visited(By default

        // set as false)

        boolean visited[] = new boolean[V];

        // Create a queue for BFS

        LinkedList<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it

        visited[s]=true;

        queue.add(s);

        while (queue.size() != 0)

        {

            // Dequeue a vertex from queue and print it

            s = queue.poll();

            System.out.print(s+" ");

            // Get all adjacent vertices of the dequeued vertex s

            // If a adjacent has not been visited, then mark it

            // visited and enqueue it

            Iterator<Integer> i = adj[s].listIterator();

            while (i.hasNext())

            {

                int n = i.next();

                if (!visited[n])

                {

                    visited[n] = true;

                    queue.add(n);

                }

            }

        }

    }

    // Driver method to

    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, 0);

        g.addEdge(2, 3);

        g.addEdge(3, 3);

        System.out.println("Following is Breadth First Traversal "+

                           "(starting from vertex 2)");

        g.BFS(2);

    }

}

OUTPUT:

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

**Prim’s Minimum Spanning Tree (MST) | Greedy**

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class MST {

    // Number of vertices in the graph

    private static final int V = 5;

    // A utility function to find the vertex with minimum key

    // value, from the set of vertices not yet included in MST

    int minKey(int key[], Boolean mstSet[])

    {

        // Initialize min value

        int min = Integer.MAX\_VALUE, min\_index = -1;

        for (int v = 0; v < V; v++)

            if (mstSet[v] == false && key[v] < min) {

                min = key[v];

                min\_index = v;

            }

        return min\_index;

    }

    // A utility function to print the constructed MST stored in

    // parent[]

    void printMST(int parent[], int graph[][])

    {

        System.out.println("Edge \tWeight");

        for (int i = 1; i < V; i++)

            System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);

    }

    // Function to construct and print MST for a graph represented

    // using adjacency matrix representation

    void primMST(int graph[][])

    {

        // Array to store constructed MST

        int parent[] = new int[V];

        // Key values used to pick minimum weight edge in cut

        int key[] = new int[V];

        // To represent set of vertices included in MST

        Boolean mstSet[] = new Boolean[V];

        // Initialize all keys as INFINITE

        for (int i = 0; i < V; i++) {

            key[i] = Integer.MAX\_VALUE;

            mstSet[i] = false;

        }

        // Always include first 1st vertex in MST.

        key[0] = 0; // Make key 0 so that this vertex is

        // picked as first vertex

        parent[0] = -1; // First node is always root of MST

        // The MST will have V vertices

        for (int count = 0; count < V - 1; count++) {

            // Pick thd minimum key vertex from the set of vertices

            // not yet included in MST

            int u = minKey(key, mstSet);

            // Add the picked vertex to the MST Set

            mstSet[u] = true;

            // Update key value and parent index of the adjacent

            // vertices of the picked vertex. Consider only those

            // vertices which are not yet included in MST

            for (int v = 0; v < V; v++)

                // graph[u][v] is non zero only for adjacent vertices of m

                // mstSet[v] is false for vertices not yet included in MST

                // Update the key only if graph[u][v] is smaller than key[v]

                if (graph[u][v] != 0 && mstSet[v] == false && graph[u][v] < key[v]) {

                    parent[v] = u;

                    key[v] = graph[u][v];

                }

        }

        // print the constructed MST

        printMST(parent, graph);

    }

    public static void main(String[] args)

    {

        /\* Let us create the following graph

        2 3

        (0)--(1)--(2)

        | / \ |

        6| 8/ \5 |7

        | /     \ |

        (3)-------(4)

            9         \*/

        MST t = new MST();

        int graph[][] = new int[][] { { 0, 2, 0, 6, 0 },

                                      { 2, 0, 3, 8, 5 },

                                      { 0, 3, 0, 0, 7 },

                                      { 6, 8, 0, 0, 9 },

                                      { 0, 5, 7, 9, 0 } };

        // Print the solution

        t.primMST(graph);

    }

}

Output:

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5