# Write a Java program to construct a graph using adjacency matrix and implement DFS and BFS traversing of it. Code:-

**package** Graph;

//java program to represent graph using adjacency Matrix,

BFS and DFS

**import** java.util.Scanner;

**public** **class** Graph {

**private** **int**[][] adjacencyMatrix; // maintains edges

**private** **int** numVertices;

**public** Graph(**int** numVertices) { **this**.numVertices = numVertices;

adjacencyMatrix = **new**

**int**[numVertices][numVertices];

}

// method to add an edge to the graph **public** **void** addEdge(**int** start, **int** end) { adjacencyMatrix[start][end] = 1; adjacencyMatrix[end][start] = 1; // For an undirected graph

}

// Breadth First Search (BFS) **public** **void** bfs(**int** startVertex) { **boolean**[] visited = **new** **boolean**[numVertices]; **int**[] queue = **new** **int**[numVertices]; **int** front = 0, rear = 0;

// Mark the start vertex as visited and enqueue it

visited[startVertex] = **true**; queue[rear++] = startVertex;

System.***out***.println("BFS traversal starting from vertex : "+startVertex+ " : ");

**while**(front<rear) {

**int** currentVertex = queue[front++]; //

Dequeue

System.***out***.print(currentVertex+" ");

// endqueue all adjacent unvisited vertices **for**(**int** i=0; i<numVertices; i++) { **if**(adjacencyMatrix[currentVertex][i]==1

&& !visited[i]) {

visited[i] = **true**; queue[rear++] = i;

}

}

}

System.***out***.println();

}

// Depth First Search (DFS) **public** **void** dfs(**int** startVertex) { **boolean**[] visited = **new** **boolean**[numVertices];

**int**[] stack = **new** **int**[numVertices];

**int** top = -1;

// Push the start vertex onto the stack and mark it as visited

stack[++top]= startVertex; visited[startVertex] = **true**;

System.***out***.println("DFS travesal starting from

vertex "+startVertex+" : "); **while**(top>=0) {

|  |  |
| --- | --- |
| from stack | **int** currentVertex = stack[top--]; // Pop |
|  | System.***out***.print(currentVertex+ " "); |
|  |  |
| the stack | // Push all adjacent unvisited vertices onto |
|  | **for**(**int** i=numVertices-1;i>=0;i--) { |
|  | // Reverse order to mimic recursion |
|  | **if**(adjacencyMatrix[currentVertex][i]==1 |

&& !visited[i]) {

visited[i] = **true**;

stack[++top] = i;

}

}

}

System.***out***.println();

}

// Method to display the adjacency matrix **public** **void** displayAdjacencyMatrix() { System.***out***.println("Adjacency Matrix:"); **for**(**int** i=0;i<numVertices;i++) { // Vertices **for**(**int** j=0;j<numVertices;j++){ // edges System.***out***.print(adjacencyMatrix[i][j]

+ " ");

}

System.***out***.println();

}

}

**public** **static** **void** main(String[] args) { Scanner sc = **new** Scanner(System.***in***); System.***out***.println("Enter the number of verices:");

**int** numVertices = sc.nextInt();

Graph graph = **new** Graph(numVertices);

System.***out***.println("Enter the number of edges:

");

**int** numEdges = sc.nextInt();

System.***out***.println("Enter the edges (source and destination): ");

**for**(**int** i=0; i<numEdges;i++) { **int** src = sc.nextInt(); **int** dest = sc.nextInt();

graph.addEdge(src, dest);

}

graph.displayAdjacencyMatrix();

System.***out***.println("Enter the start vertex for

BFS: ");

**int** bfsStartVertex = sc.nextInt(); graph.bfs(bfsStartVertex); System.***out***.println("Enter the start vertex for

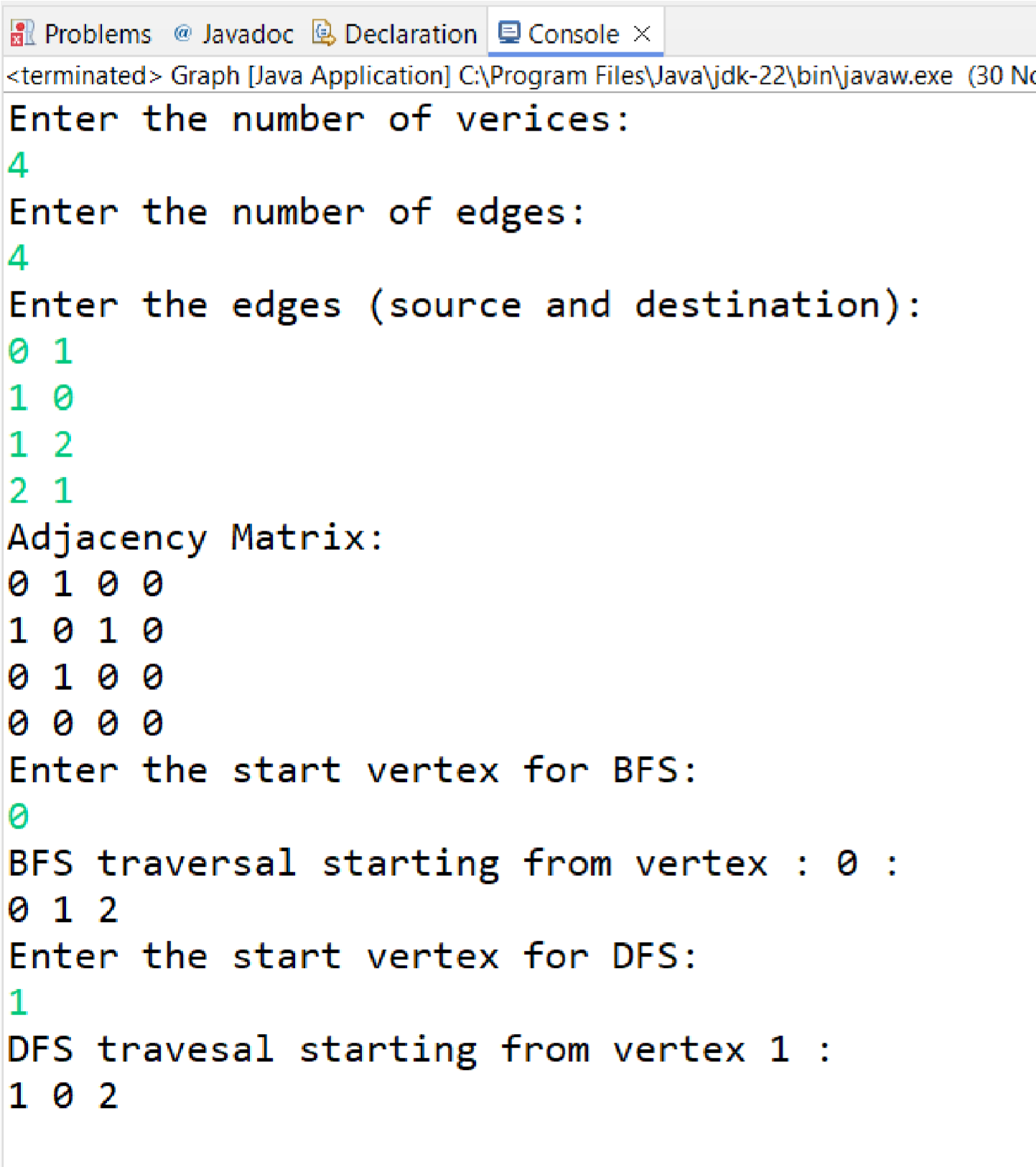
DFS: ");

**int** dfsStartVertex = sc.nextInt(); graph.dfs(dfsStartVertex); sc.close();

}

}

# Output:-



# Write a Java program to find the minimum spanning tree of a graph. Code:-

**package** Graph;

**import** java.util.Scanner;

**class** Edge {

**int** src, dest, weight;

// Constructor

Edge(**int** src, **int** dest, **int** weight) { **this**.src = src; **this**.dest = dest; **this**.weight = weight;

}

}

**public** **class** KruskalAlgorithm {

**private** **int** V; // Number of vertices **private** **int** E; // Number of edges

**private** Edge[] edges; // Array to store all edges **private** **int** edgeCount = 0; // Counter for added edges

// Constructor

**public** KruskalAlgorithm(**int** vertices, **int** edgesCount)

{

**this**.V = vertices; **this**.E = edgesCount; edges = **new** Edge[edgesCount];

}

// Add an edge to the graph

**public** **void** addEdge(**int** src, **int** dest, **int** weight) { edges[edgeCount++] = **new** Edge(src, dest, weight);

}

// Find the parent of a vertex (with path compression) **private** **int** findParent(**int**[] parent, **int** vertex) { **if** (parent[vertex] != vertex) {

parent[vertex] = findParent(parent, parent[vertex]); }

**return** parent[vertex];

}

// Perform union of two sets

**private** **void** union(**int**[] parent, **int**[] rank, **int** x, **int** y) { **int** rootX = findParent(parent, x); **int** rootY = findParent(parent, y);

**if** (rootX != rootY) {

**if** (rank[rootX] < rank[rootY]) { parent[rootX] = rootY;

} **else** **if** (rank[rootX] > rank[rootY]) { parent[rootY] = rootX;

} **else** {

parent[rootY] = rootX; rank[rootX]++;

}

}

}

// Function to sort edges by weight using a simple bubble sort

**private** **void** sortEdges() { **for** (**int** i = 0; i < E - 1; i++) {

**for** (**int** j = 0; j < E - i - 1; j++) { // Corrected loop condition

**if** (edges[j].weight > edges[j + 1].weight)

{

Edge temp = edges[j]; edges[j] = edges[j + 1]; edges[j + 1] = temp;

}

}

}

}

// Kruskal's algorithm to find MST **public** **void** kruskalMST() {

// Sort edges by weight sortEdges();

// Arrays for Union-Find **int**[] parent = **new** **int**[V]; **int**[] rank = **new** **int**[V];

// Initialize Union-Find structure **for** (**int** i = 0; i < V; i++) { parent[i] = i; rank[i] = 0;

}

Edge[] mst = **new** Edge[V - 1]; // Array to store the MST edges

**int** mstIndex = 0; // Index for MST edges **int** mstWeight = 0; // Total weight of MST

// Iterate through the sorted edges **for** (**int** i = 0; i < E; i++) {

**if** (mstIndex == V - 1) **break**; // Stop if MST is complete

Edge edge = edges[i];

**int** srcParent = findParent(parent, edge.src); **int** destParent = findParent(parent, edge.dest);

// If adding this edge doesn't form a cycle **if** (srcParent != destParent) { mst[mstIndex++] = edge; // Include the edge in MST

mstWeight += edge.weight; union(parent, rank, srcParent, destParent); // Merge sets

}

}

// Print the MST

System.***out***.println("Edges in the MST:"); System.***out***.println("Src -- Dest == Weight"); **for** (**int** i = 0; i < mstIndex; i++) { System.***out***.println(mst[i].src + " -- " + mst[i].dest + " == " + mst[i].weight);

}

System.***out***.println("Total Weight of MST: " + mstWeight);

}

// Main method to test the program **public** **static** **void** main(String[] args) { Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter the number of vertices:

");

**int** V = sc.nextInt();

System.***out***.println("Enter the number of edges: ");

**int** E = sc.nextInt();

KruskalAlgorithm graph = **new** KruskalAlgorithm(V,

E);

System.***out***.println("Enter the edges in the format: src dest weight");

**for** (**int** i = 0; i < E; i++) { // Removed incorrect semicolon **int** src = sc.nextInt(); **int** dest = sc.nextInt(); **int** weight = sc.nextInt(); graph.addEdge(src, dest, weight);

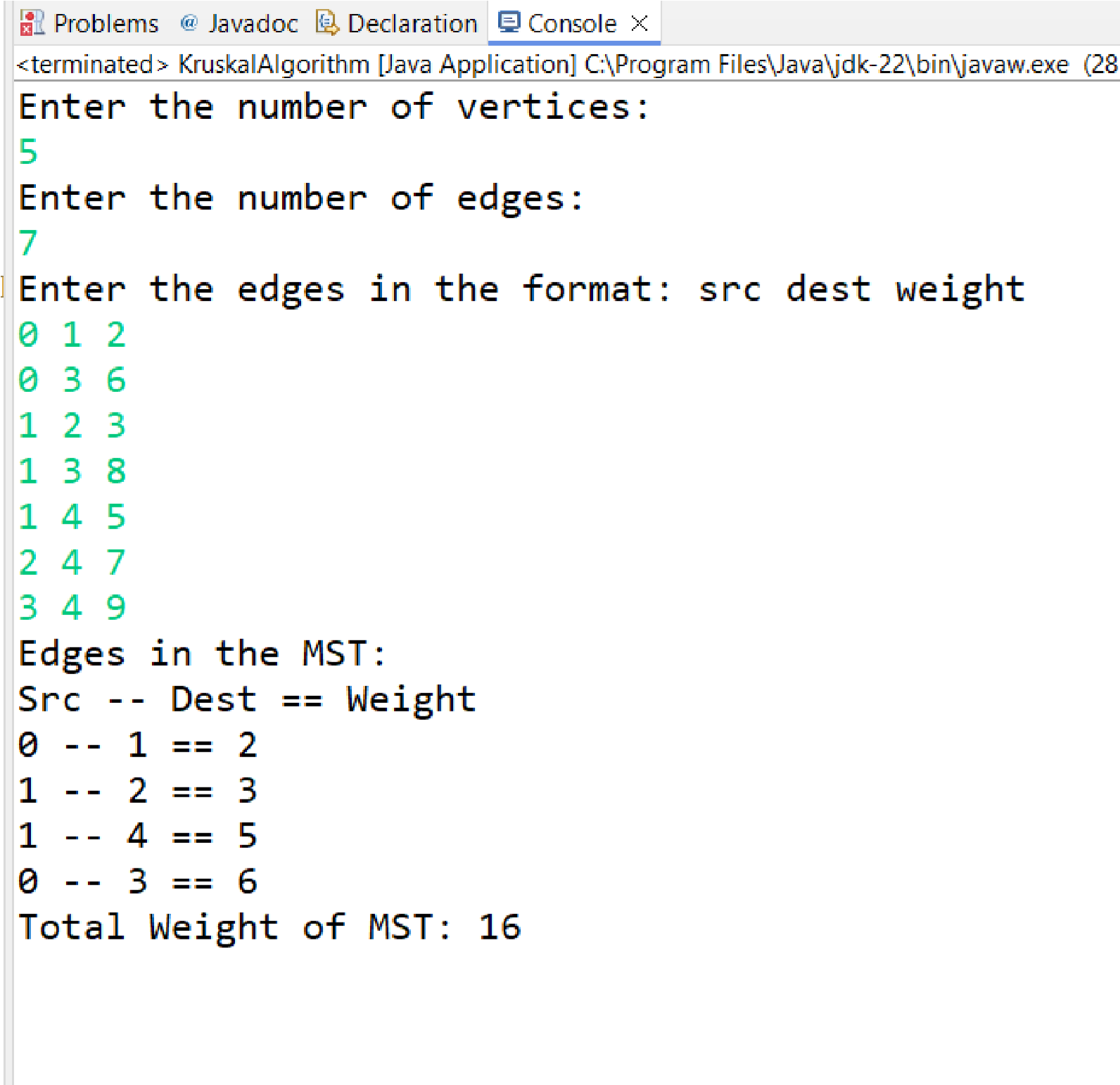
}

graph.kruskalMST(); sc.close();

}

}

# Output:-



**Conculsion :-**

**Implement non-linear data structures successfully.**