LAB 10

PRACTICE QUESTIONS

Program 1:Write a simple Binary Search TreeProgram

**package** PRACTICE;

**public** **class** BST\_class {

//node class that defines BST node

**class** Node {

**int** key;

Node left, right;

**public** Node(**int** data){

key = data;

left = right = **null**;

}

}

// BST root node

Node root;

// Constructor for BST =>initial empty tree

BST\_class(){

root = **null**;

}

// insert a node in BST

**void** insert(**int** key) {

root = insert\_Recursive(root, key);

}

//recursive insert function

Node insert\_Recursive(Node root, **int** key) {

//tree is empty

**if** (root == **null**) {

root = **new** Node(key);

**return** root;

}

//traverse the tree

**if** (key < root.key) //insert in the left subtree

root.left = insert\_Recursive(root.left, key);

**else** **if** (key > root.key) //insert in the right subtree

root.right = insert\_Recursive(root.right, key);

// return pointer

**return** root;

}

// method for inorder traversal of BST

**void** inorder() {

inorder\_Recursive(root);

}

// recursively traverse the BST

**void** inorder\_Recursive(Node root) {

**if** (root != **null**) {

inorder\_Recursive(root.left);

System.***out***.print(root.key + " ");

inorder\_Recursive(root.right);

}

}

**boolean** search(**int** key) {

root = search\_Recursive(root, key);

**if** (root!= **null**)

**return** **true**;

**else**

**return** **false**;

}

//recursive insert function

Node search\_Recursive(Node root, **int** key) {

// Base Cases: root is null or key is present at root

**if** (root==**null** || root.key==key)

**return** root;

// val is greater than root's key

**if** (root.key > key)

**return** search\_Recursive(root.left, key);

// val is less than root's key

**return** search\_Recursive(root.right, key);

}

}

**class** Main{

**public** **static** **void** main(String[] args) {

BST\_class bst = **new** BST\_class();

bst.insert(45);

bst.insert(10);

bst.insert(7);

bst.insert(12);

bst.insert(90);

bst.insert(50);

System.***out***.println("The BST Created with input data(Left-root-right):");

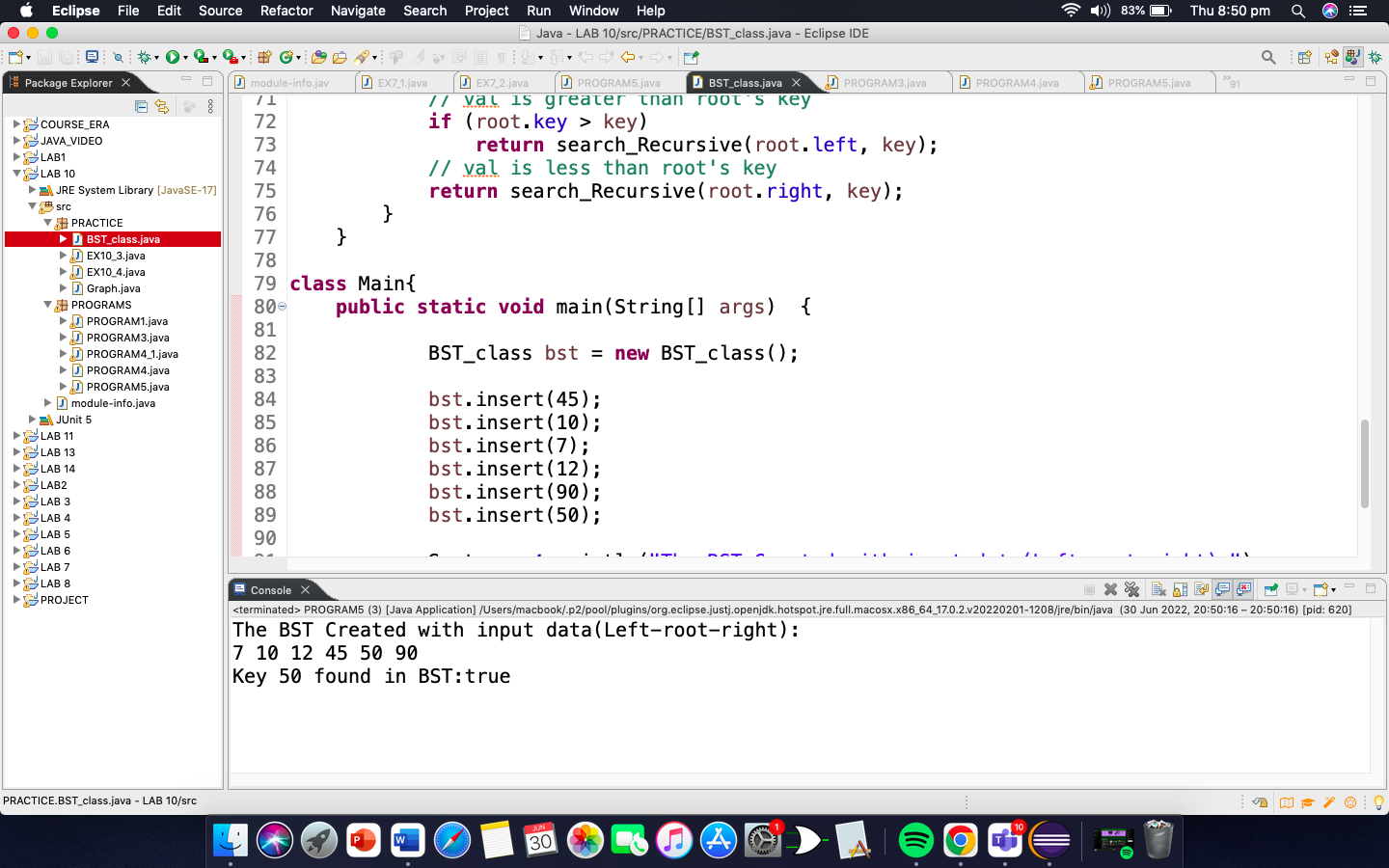
bst.inorder();

**boolean** ret\_val = bst.search (50);

System.***out***.println("\nKey 50 found in BST:" + ret\_val );

}

}



Program2:Implement Graph Data Structure

**package** PRACTICE;

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

// inner class

// to keep track of edges

**class** Edge{

**int** src, dest;

}

// number of vertices and edges

**int** vertices, edges;

// array to store all edges

Edge[] edge;

Graph(**int** vertices, **int** edges){

**this**.vertices = vertices;

**this**.edges = edges;

// initialize the edge array

edge = **new** Edge[edges];

**for**(**int** i=0; i<edges; i++) {

// each element of the edge array

// is an object of Edge type

edge[i] = **new** Edge();

}

}

**public** **static** **void** main(String[] args) {

// create an object of Graph class

**int** noVertices = 5;

**int** noEdges = 8;

Graph g = **new** Graph(noVertices, noEdges);

// create graph

g.edge[0].src = 1; // edge 1---2

g.edge[0].dest = 2;

g.edge[1].src = 1; // edge 1---3

g.edge[1].dest = 3;

g.edge[2].src = 1; // edge 1---4

g.edge[2].dest = 4;

g.edge[3].src = 2; // edge 2---4

g.edge[3].dest = 4;

g.edge[4].src = 2; // edge 2---5

g.edge[4].dest = 5;

g.edge[5].src = 3; // edge 3---4

g.edge[5].dest = 4;

g.edge[6].src = 3; // edge 3---5

g.edge[6].dest = 5;

g.edge[7].src = 4; // edge 4---5

g.edge[7].dest = 5;

// print graph

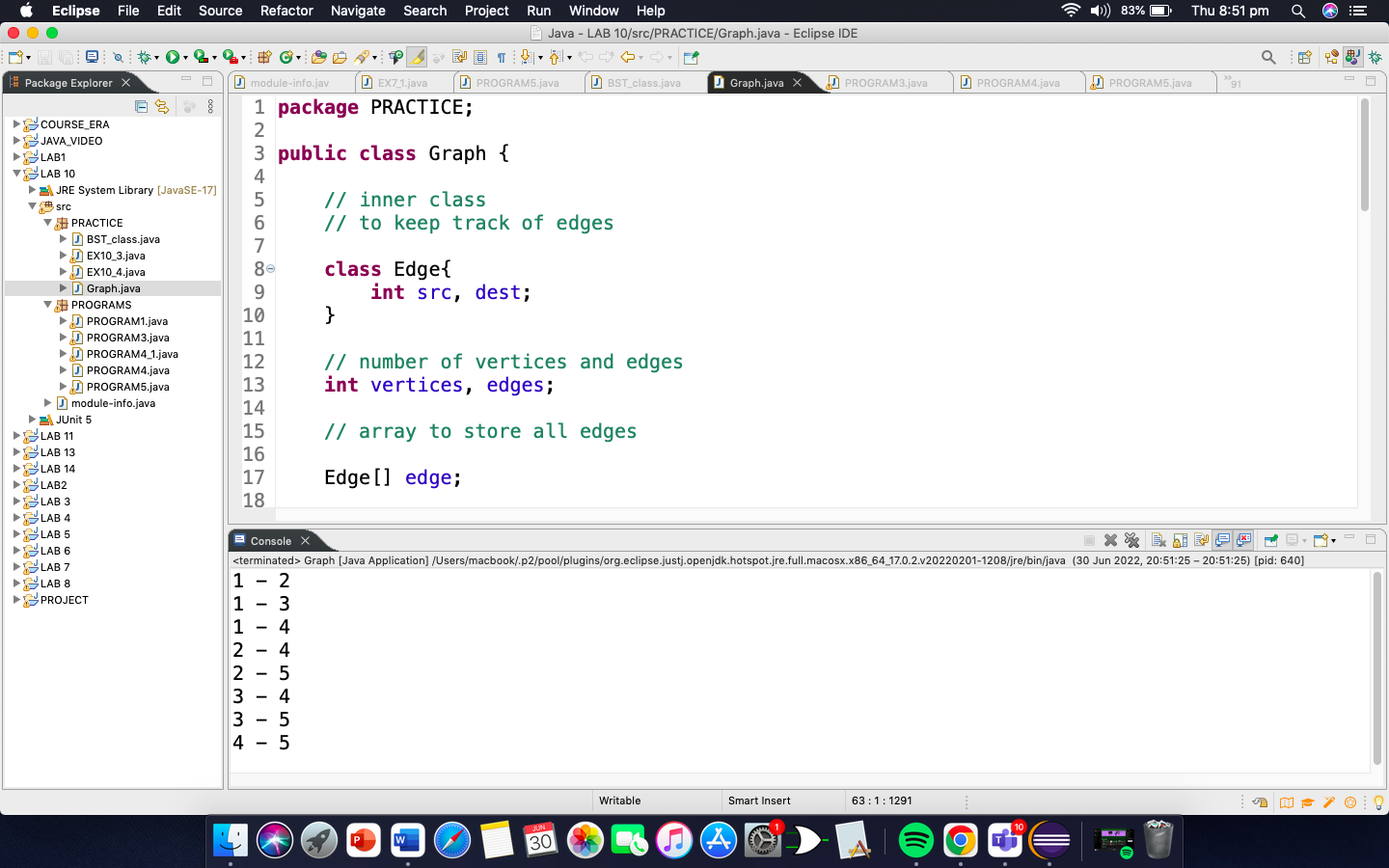
**for**(**int** i=0; i<noEdges; i++) {

System.***out***.println(g.edge[i].src + " - " + g.edge[i].dest);

}

}

}



Program 3:Program to detect cycle in an undirected graph

package PRACTICE;

import java.util.\*;

import java.io.\*;

//This class represents a directed graph using adjacency list representation

public class EX10\_3 {

// No. of vertices

private int V;

// Adjacency List Representation

private LinkedList<Integer> adj[];

//constructor

EX10\_3(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(v);

adj[w].add(w);

}

// A recursive function that uses visited[] and parent to detect cycle in subgraph reachable

// from vertex v.

boolean isCyclicUtil(int v, Boolean visited[], int parent) {

// Mark the current node as visited

visited[v] = true;

Integer i;

// Recur for all the vertices adjacent to this vertex

Iterator<Integer> it = adj[v].iterator();

while(it.hasNext()) {

i = it.next();

// If an adjacent is not

// visited, then recur for that

// adjacent

if(!visited[i]) {

if(isCyclicUtil(i, visited, v)) {

return true;

}

}

// If an adjacent is visited and not parent of current vertex,then there is a cycle.

else if (i != parent) {

return true;

}

return false;

}

return false;

}

// Returns true if the graph contains a cycle, else false.

boolean isCyclic() {

// Mark all the vertices as not visited and not part of recursion stack

Boolean visited[] = new Boolean[V];

for (int i = 0; i < V; i++) {

visited[i] = false;

}

// Call the recursive helper function to detect cycle in different DFS tree.

for(int u=0; u<V; u++) {

// Don't recur for u if already visited

if(!visited[u]) {

if(isCyclicUtil(u, visited, -1)) {

return true;

}

}

}

return false;

}

// Driver method to test above methods

public static void main(String args[]) {

// Create a graph given

// in the above diagram

EX10\_3 g1 = new EX10\_3(5);

g1.addEdge(1, 0);

g1.addEdge(0, 2);

g1.addEdge(2, 1);

g1.addEdge(0, 3);

g1.addEdge(3, 4);

if (g1.isCyclic()) {

System.out.println("Graph contains cycle");

}else {

System.out.println("graph does't contains cycle");

}

EX10\_3 g2 = new EX10\_3(3);

g2.addEdge(0, 1);

g2.addEdge(1, 2);

if (g2.isCyclic())

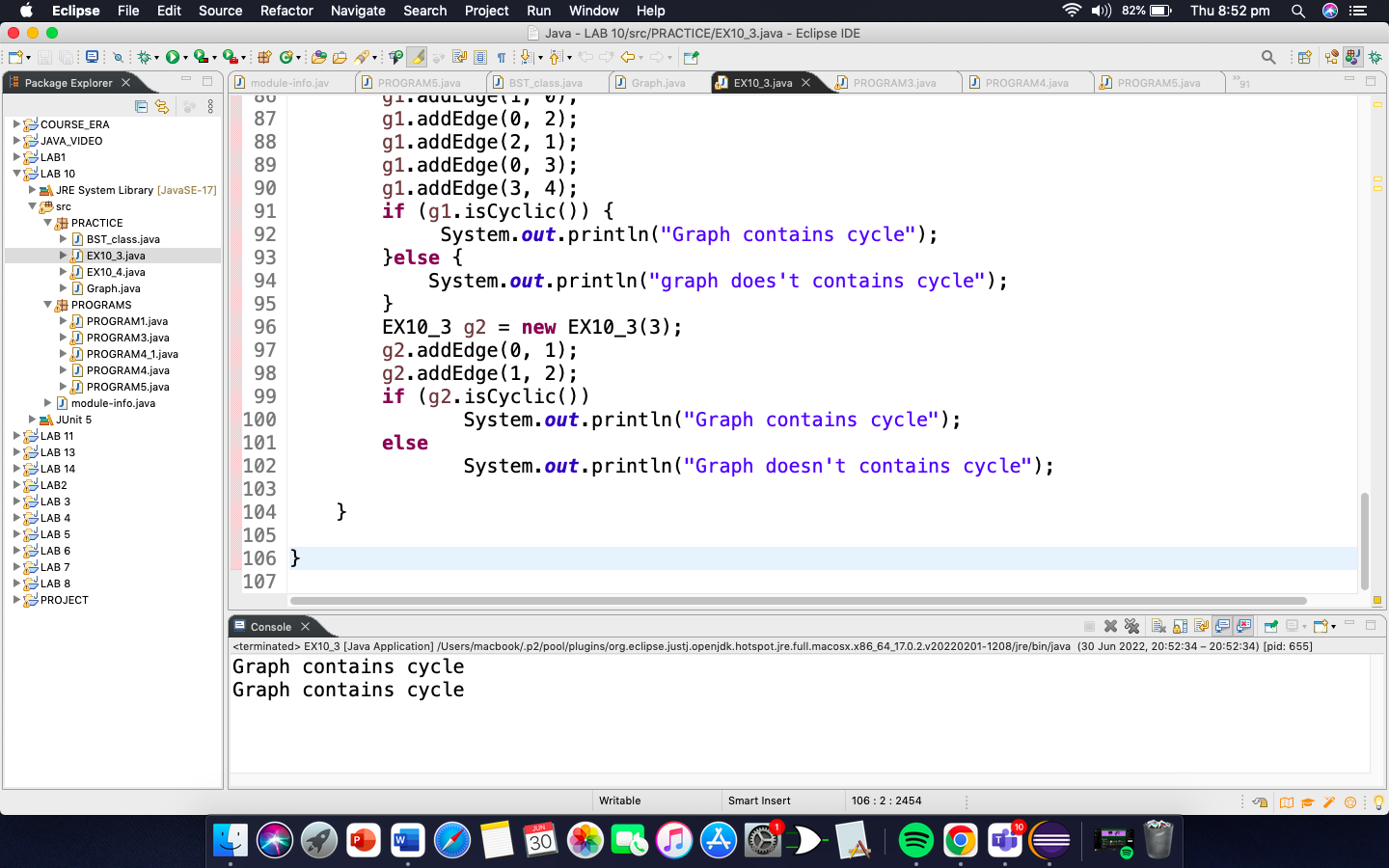
System.out.println("Graph contains cycle");

else

System.out.println("Graph doesn't contains cycle");

}

}



Program4:Program to find numberof triangles in an UndirectedGraph. The program is foradjacency matrix representationof the graph

**package** PRACTICE;

**import** java.io.\*;

**public** **class** EX10\_4 {

**int** V = 4;

**void** multiply(**int** A[][], **int** B[][], **int** C[][]) {

**for**(**int** i=0; i<V; i++) {

**for**(**int** j=0; j<V; j++) {

C[i][j] = 0;

**for**(**int** k=0; k<V; k++) {

C[i][j] += A[i][k]\* B[k][j];

}

}

}

}

**int** getTrace(**int** graph[][]) {

**int** trace = 0;

**for**(**int** i=0; i<V; i++) {

trace += graph[i][i];

}

**return** trace;

}

**int** triangleInGraph(**int** graph[][]) {

// To Store graph^2

**int**[][] aux2 = **new** **int**[V][V];

// To Store graph^3

**int**[][] aux3 = **new** **int**[V][V];

// Initialising aux matrices

// with 0

**for** (**int** i = 0; i < V; ++i) {

**for** (**int** j = 0; j < V; ++j) {

aux2[i][j] = aux3[i][j] = 0;

}

}

multiply(graph, graph, aux2);

// after this multiplication aux3

// is graph^3 printMatrix(aux3)

multiply(graph, aux2, aux3);

**int** trace = getTrace(aux3);

**return** trace / 6;

}

// Driver code

**public** **static** **void** main(String args[]) {

EX10\_4 obj = **new** EX10\_4();

**int** graph[][] = { {0, 1, 1, 0},

{1, 0, 1, 1},

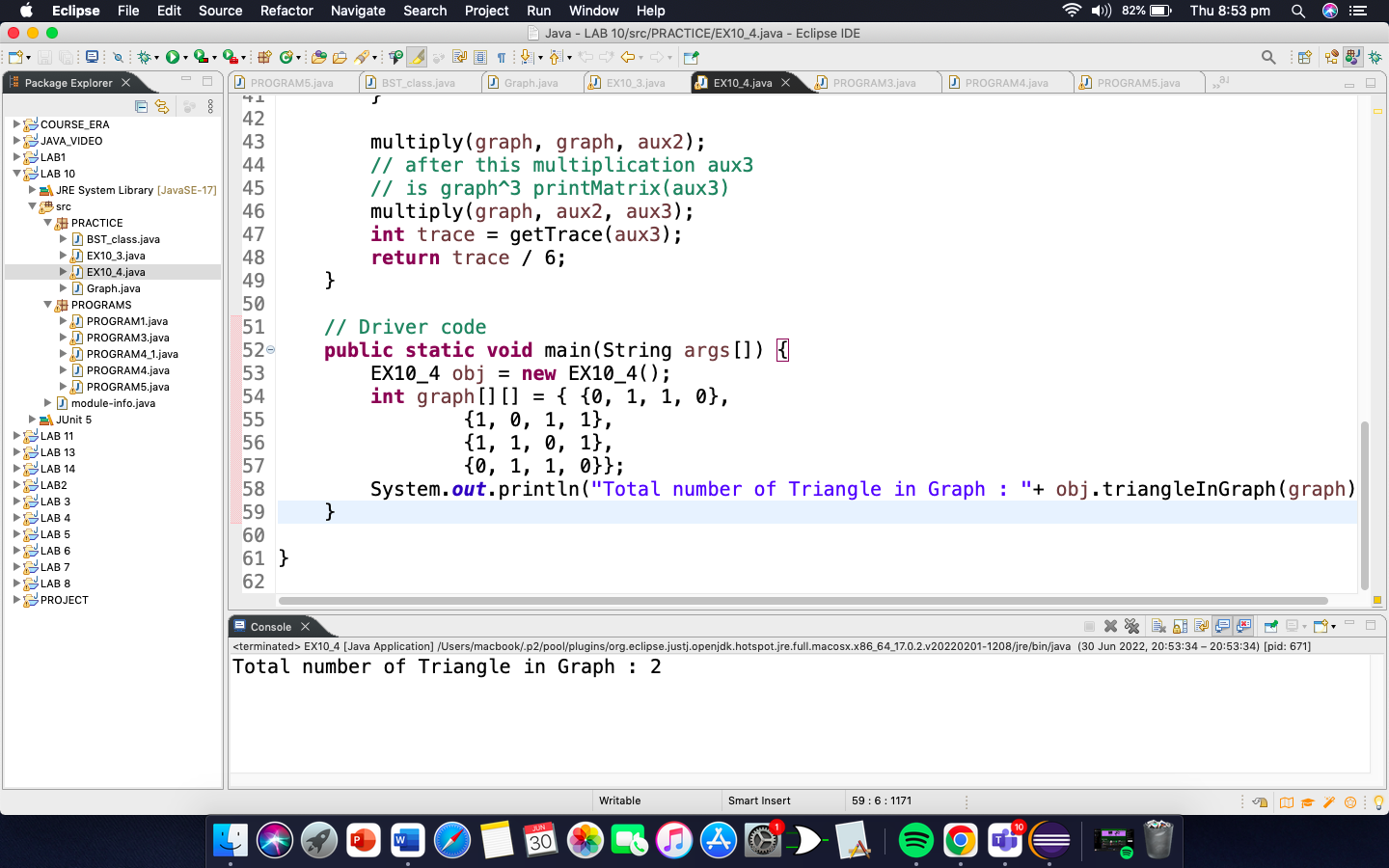
{1, 1, 0, 1},

{0, 1, 1, 0}};

System.***out***.println("Total number of Triangle in Graph : "+ obj.triangleInGraph(graph));

}

}



PROGRAMMING EXERCISE

1. Write a code to check weather a graph is connected or not?

**package** PROGRAMS;

**import** java.util.\*;

**public** **class** PROGRAM1 {

**static** **class** Graph{

**int** vertices;

LinkedList<Integer> adjList [];

**public** Graph(**int** vertices){

**this**.vertices = vertices;

adjList = **new** LinkedList[vertices];

**for** (**int** i = 0; i <vertices ; i++) {

adjList[i] = **new** LinkedList<>();

}

}

**public** **void** addEdge(**int** source, **int** destination){

adjList[source].addFirst(destination);

adjList[destination].addFirst(source);

}

}

**public** **void** isConnected(Graph graph){

**int** vertices = graph.vertices;

LinkedList<Integer> adjList [] = graph.adjList;

**boolean**[] visited = **new** **boolean**[vertices];

DFS(0, adjList, visited);

**int** count = 0;

**for** (**int** i = 0; i <visited.length ; i++) {

**if**(visited[i])

count++;

}

**if**(vertices==count){

System.***out***.println("Given graph is connected");

}**else**{

System.***out***.println("Given graph is not connected");

}

}

**public** **void** DFS(**int** source, LinkedList<Integer> adjList [], **boolean**[] visited){

visited[source] = **true**;

**for** (**int** i = 0; i <adjList[source].size() ; i++) {

**int** neighbor = adjList[source].get(i);

**if**(visited[neighbor]==**false**){

DFS(neighbor, adjList, visited);

}

}

}

**public** **static** **void** main(String[] args) {

Graph graph = **new** Graph(5);

graph.addEdge(0,1);

graph.addEdge(1,3);

graph.addEdge(3,2 );

PROGRAM1 c = **new** PROGRAM1();

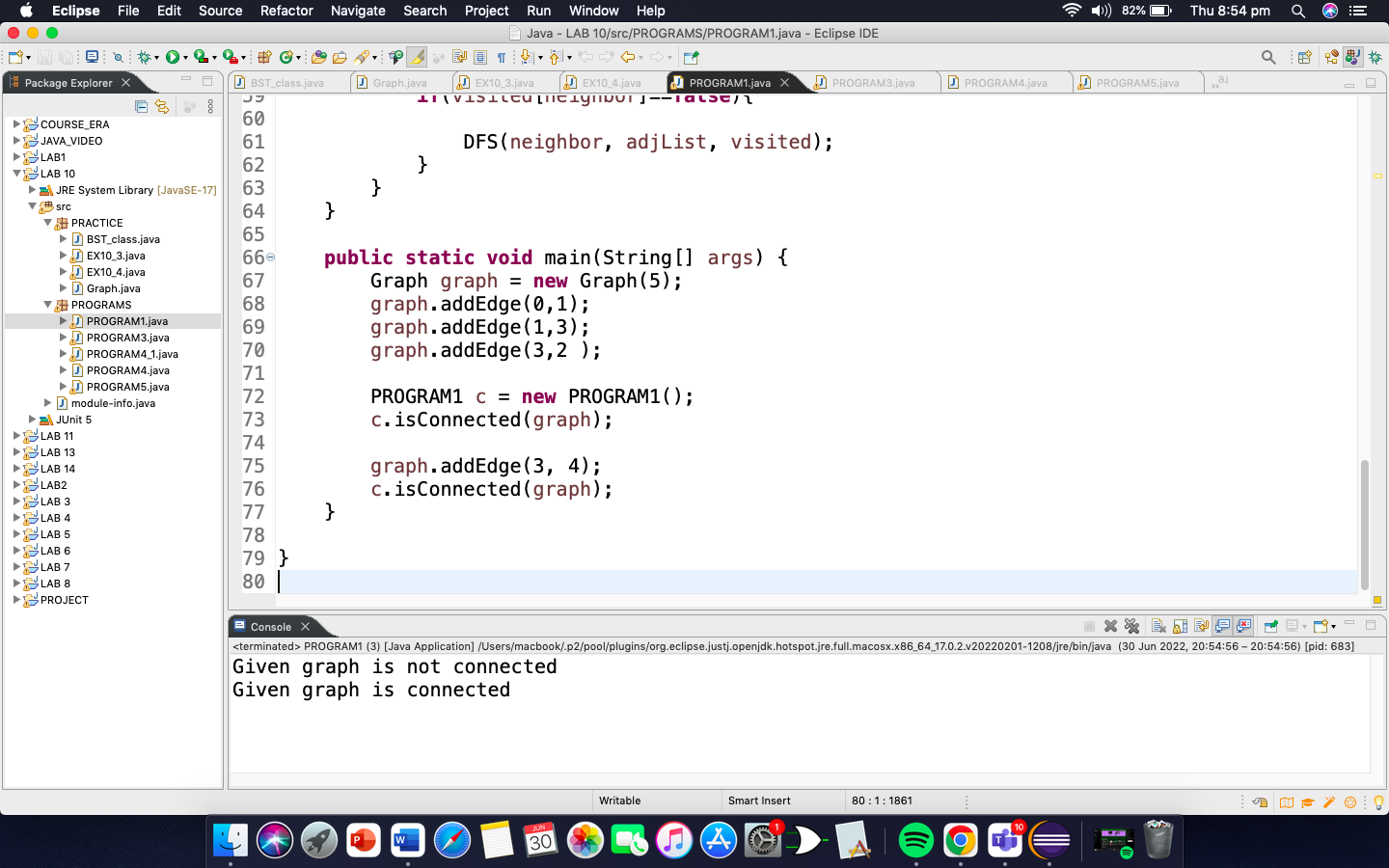
c.isConnected(graph);

graph.addEdge(3, 4);

c.isConnected(graph);

}

}



1. Implement the following graphs and show how you can find that it has cycles in it or not.Calculate its vertices and edges, in degrees and out degrees. Then convert your program into generic code.

package PROGRAMS;

import java.util.Iterator;

import java.util.Scanner;

import java.util.LinkedList;

public class PROGRAM3 {

static class Graph

{

// vertices and edges

int v, e;

int[][] dir;

//Graph Constructor

Graph(int v, int e) {

this.v = v;

this.e = e;

dir = new int[v][];

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

dir[i] = new int[v];

}

}

// No. of vertices

private int V;

// Adjacency List Representation

private LinkedList<Integer> adj[];

//constructor

PROGRAM3(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(v);

adj[w].add(w);

}

static Graph createGraph(int v, int e)

{

Graph G = new Graph(v, e);

G.dir[1][2] = 1;

G.dir[1][3] = 1;

G.dir[2][7] = 1;

G.dir[2][6] = 1;

G.dir[2][8] = 1;

G.dir[3][4] = 1;

G.dir[3][5] = 1;

G.dir[6][8] = 1;

G.dir[6][10] = 1;

G.dir[6][9] = 1;

G.dir[7][9] = 1;

G.dir[10][11] = 1;

return G;

}

// A recursive function that uses visited[] and parent to detect cycle in subgraph reachable

// from vertex v.

boolean isCyclicUtil(int v, Boolean visited[], int parent) {

// Mark the current node as visited

visited[v] = true;

Integer i;

// Recur for all the vertices adjacent to this vertex

Iterator<Integer> it = adj[v].iterator();

while(it.hasNext()) {

i = it.next();

// If an adjacent is not

// visited, then recur for that

// adjacent

if(!visited[i]) {

if(isCyclicUtil(i, visited, v)) {

return true;

}

}

// If an adjacent is visited and not parent of current vertex,then there is a cycle.

else if (i != parent) {

return true;

}

return false;

}

return false;

}

// Returns true if the graph contains a cycle, else false.

boolean isCyclic() {

// Mark all the vertices as not visited and not part of recursion stack

Boolean visited[] = new Boolean[V];

for (int i = 0; i < V; i++) {

visited[i] = false;

}

// Call the recursive helper function to detect cycle in different DFS tree.

for(int u=0; u<V; u++) {

// Don't recur for u if already visited

if(!visited[u]) {

if(isCyclicUtil(u, visited, -1)) {

return true;

}

}

}

return false;

}

public int countEdges()

{

int sum = 0;

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

sum += adj[i].size();

return sum/2;

}

public int countVertices() {

return V;

}

static int findDegree(Graph G, int ver)

{

int degree = 0;

int d = 0;

for (int i = 0; i < G.v; i++) {

if (G.dir[ver][i] == 1)

degree++;

}

for(int j=0; j<G.v; j++) {

if (G.dir[j][ver] == 1)

d++;

}

if(G.dir[ver][ver] == 1) degree++;

System.out.println("indegree of verteice is " + d);

System.out.println("outdegree of verteice is " + degree);

return degree;

}

public static void main(String[] args)

{

PROGRAM3 g = new PROGRAM3(12);

g.addEdge(1, 2);

g.addEdge(1, 3);

g.addEdge(3, 4);

g.addEdge(3, 5);

g.addEdge(2, 6);

g.addEdge(2, 7);

g.addEdge(2, 8);

g.addEdge(6, 9);

g.addEdge(6, 8);

g.addEdge(6, 10);

g.addEdge(7, 9);

g.addEdge(10, 11);

if (g.isCyclic())

System.out.println("Graph contains cycle");

else

System.out.println("Graph doesn't contains cycle");

System.out.println("number of edges in graph is "+ g.countEdges());

System.out.println("number of vertices in graph is "+ g.countVertices());

int vertices = g.countVertices() + 1;;

int edges = g.countEdges();;

Graph G = createGraph(vertices, edges);

Scanner input = new Scanner(System.in);

System.out.println("Degree of which verteice?");

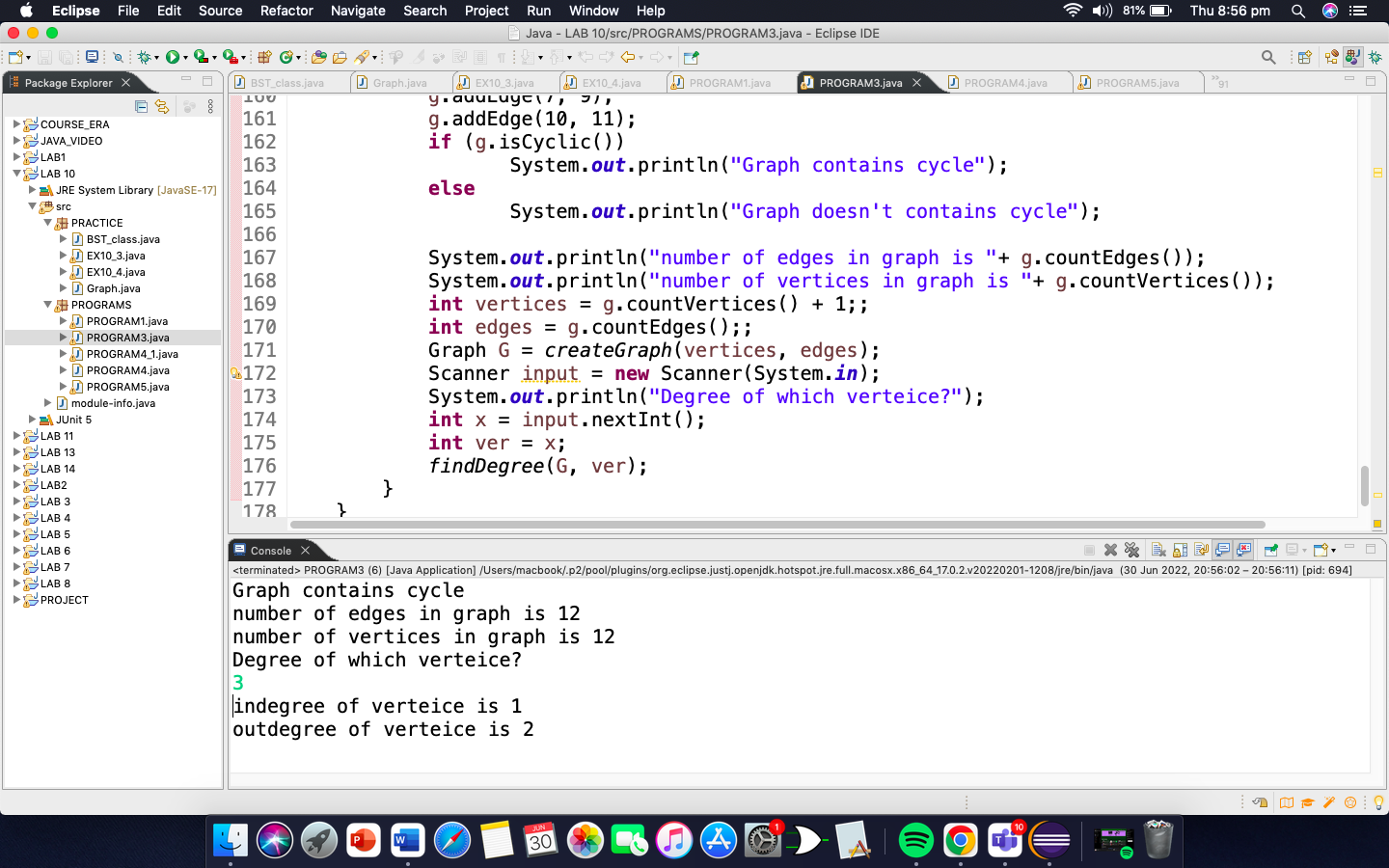
int x = input.nextInt();

int ver = x;

findDegree(G, ver);

}

}



1. Write a code for the Adjacency Matrix. An Adjacency Matrix is simple technique to use 2-dimensionalarray into matrix format.

**package** PROGRAMS;

**public** **class** PROGRAM4 {

**private** **boolean** adjMatrix[][];

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

// Initialize the matrix

**public** PROGRAM4(**int** numVertices) {

**this**.numVertices = numVertices;

adjMatrix = **new** **boolean**[numVertices][numVertices];

}

// Add edges

**public** **void** addEdge(**int** i, **int** j) {

adjMatrix[i][j] = **true**;

adjMatrix[j][i] = **true**;

}

// Print the matrix

**public** String toString() {

StringBuilder s = **new** StringBuilder();

**for** (**int** i = 0; i < numVertices; i++) {

s.append(i + ": ");

**for** (**boolean** j : adjMatrix[i]) {

s.append((j ? 1 : 0) + " ");

}

s.append("\n");

}

**return** s.toString();

}

**public** **static** **void** main(String args[]) {

PROGRAM4 g = **new** PROGRAM4(5);

g.addEdge(0, 1);

g.addEdge(0, 4);

g.addEdge(1, 2);

g.addEdge(1, 3);

g.addEdge(3, 0);

g.addEdge(3, 2);

g.addEdge(3, 4);

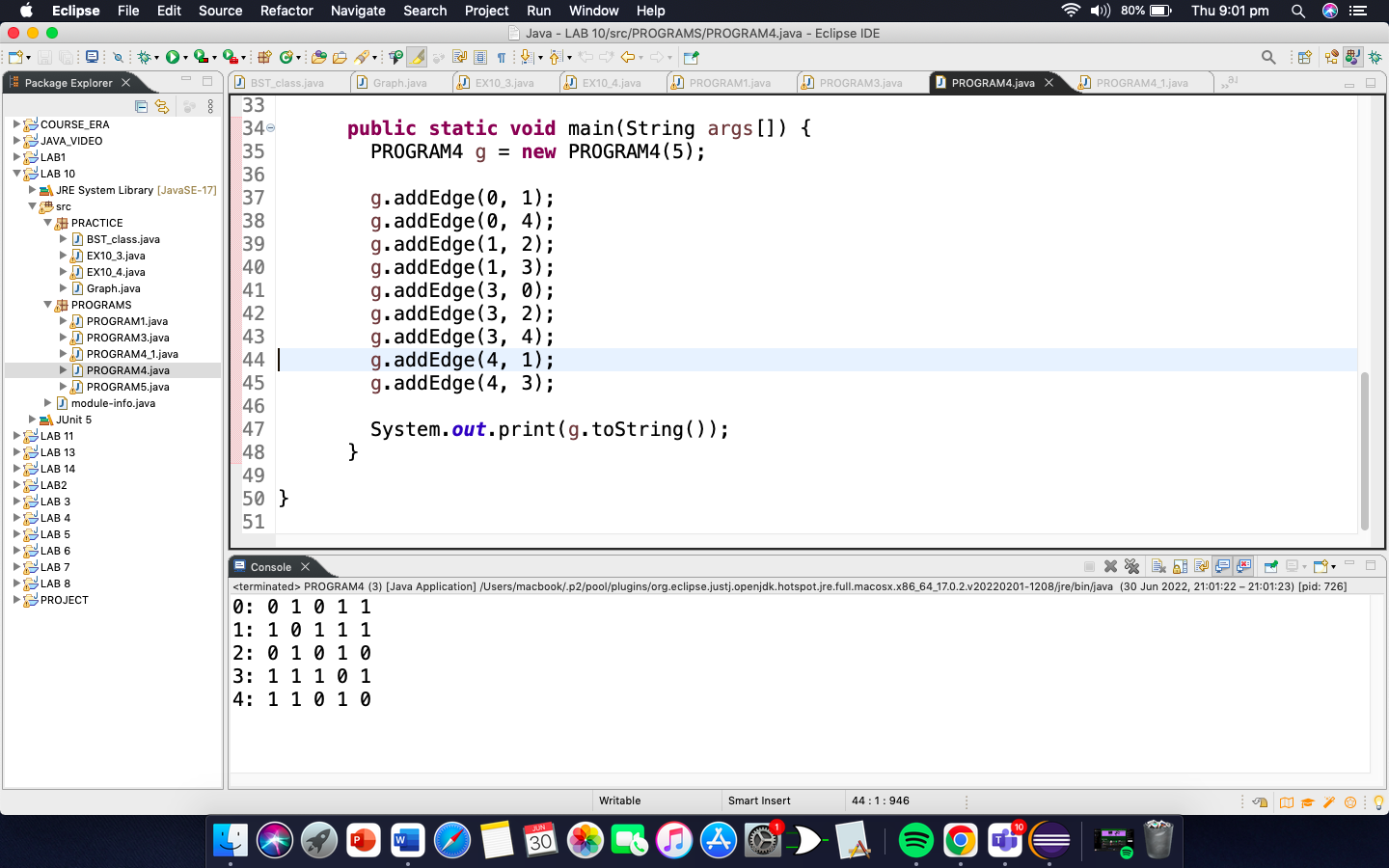
g.addEdge(4, 1);

g.addEdge(4, 3);

System.***out***.print(g.toString());

}

}



**package** PROGRAMS;

**import** java.util.Scanner;

**public** **class** PROGRAM4\_1{

**private** **final** **int** vertices;

**private** **int**[][] adjacency\_matrix;

**public** PROGRAM4\_1(**int** v)

{

vertices = v;

adjacency\_matrix = **new** **int**[vertices + 1][vertices + 1];

}

**public** **void** makeEdge(**int** to, **int** from, **int** edge) {

adjacency\_matrix[to][from] = edge;

}

**public** **int** getEdge(**int** to, **int** from) {

**return** adjacency\_matrix[to][from];

}

**public** **static** **void** main(String args[])

{

**int** v, e, count = 1, to = 0, from = 0;

Scanner sc = **new** Scanner(System.***in***);

PROGRAM4\_1 graph;

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

v = sc.nextInt();

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

e = sc.nextInt();

graph = **new** PROGRAM4\_1(v);

System.***out***.println("Enter the edges: <to> <from>");

**while** (count <= e)

{

to = sc.nextInt();

from = sc.nextInt();

graph.makeEdge(to, from, 1);

count++;

}

System.***out***.println("The adjacency matrix for the given graph is: ");

**for** (**int** i = 1; i <= v; i++)

{

System.***out***.print(i + " : ");

**for** (**int** j = 1; j <= v; j++)

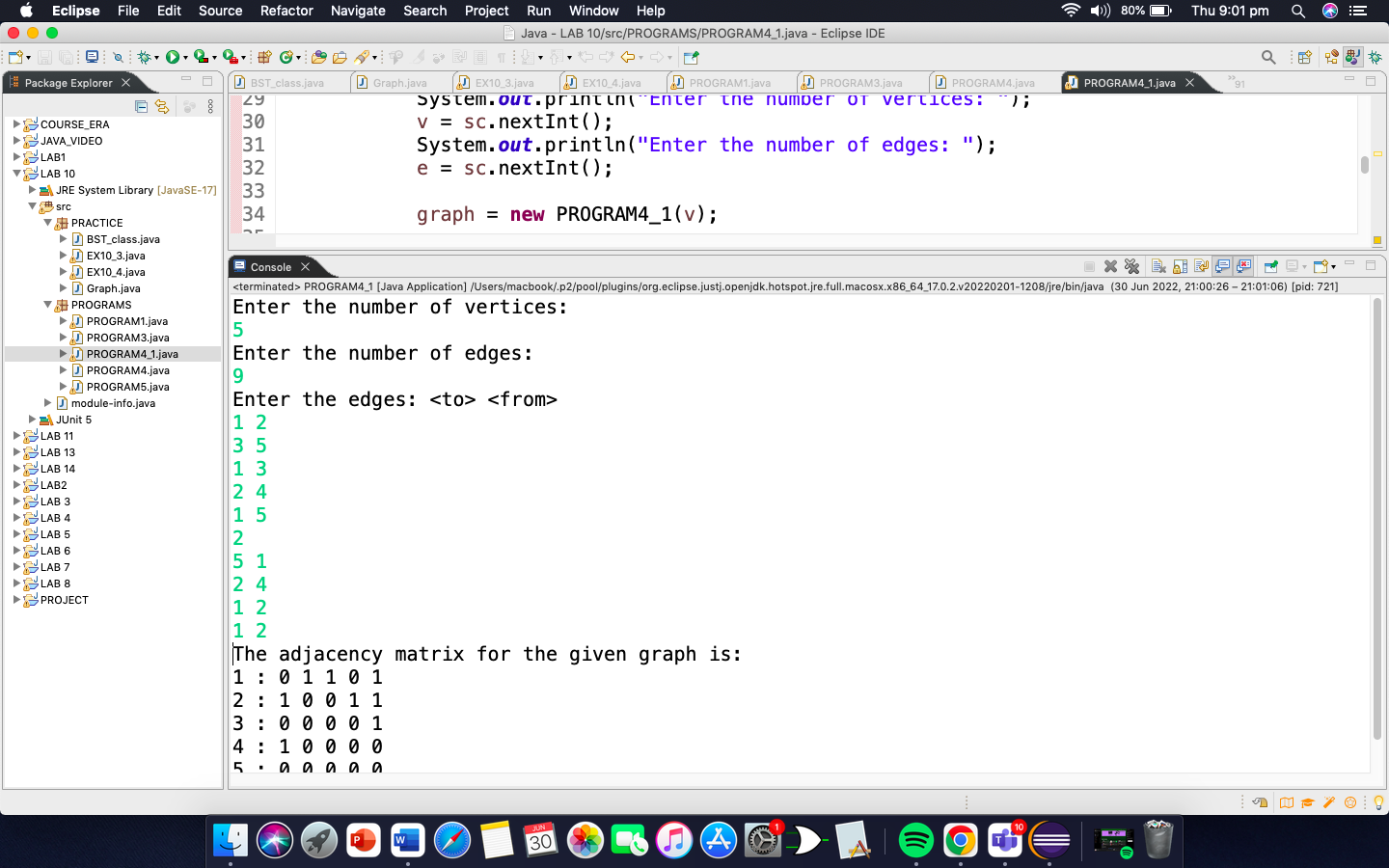
System.***out***.print(graph.getEdge(i, j) + " ");

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

}

}

}



1. Now instead of using array what if, we use linked list. Now convert the code to show how it works using linked list.

**package** PROGRAMS;

**import** java.util.\*;

**public** **class** PROGRAM5 {

**static** **void** insert(Vector<Integer> adj[],**int** u, **int** v){

// Insert a vertex v to vertex u

adj[u].add(v);

**return**;

}

//Function to display adjacency list

**static** **void** printList(Vector<Integer> adj[],**int** V){

**for**(**int** i = 1; i < V; i++){

System.***out***.print(i);

**for**(**int** j : adj[i])

System.***out***.print(" --> " + j);

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

}

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

}

//Driver code

**public** **static** **void** main(String[] args){

**int** V = 6;

Vector<Integer> []adjList = **new** Vector[V];

**for**(**int** i = 1; i < adjList.length; i++)

adjList[i] = **new** Vector<Integer>();

// Inserting edges

*insert*(adjList, 1, 2);

*insert*(adjList, 1, 5);

*insert*(adjList, 2, 3);

*insert*(adjList, 2, 4);

*insert*(adjList, 4, 1);

*insert*(adjList, 4, 3);

*insert*(adjList, 4, 5);

*insert*(adjList, 5, 2);

*insert*(adjList, 5, 4);

// Display adjacency list

System.***out***.print("Adjacency List: \n");

*printList*(adjList, V);

}

}

