***Binary Tree***

import java.io.\*;

import java.util.\*;

class Node{

int data;

Node left;

Node right;

Node(int data){

this.data = data;

left=null;

right=null;

}

}

public class Main {

static int height=-1;

static int diameter=-1;

static boolean balanced=true;

static TreeMap<Integer,Integer> topview = new TreeMap<>();

static TreeMap<Integer,Integer> bottomview = new TreeMap<>();

public static void main (String[] args){

Node sumtree = new Node(35);

sumtree.left = new Node(22);

sumtree.right = new Node(13);

sumtree.left.left = new Node(17);

sumtree.left.right = new Node(5);

sumtree.right.left = new Node(6);

sumtree.right.right = new Node(7);

sumtree.left.left.left = new Node(8);

sumtree.left.left.right = new Node(9);

sumtree.left.right.right = new Node(5);

Node tree\_level = new Node(1);

tree\_level.left = new Node(2);

tree\_level.right = new Node(3);

tree\_level.left.left = new Node(4);

tree\_level.left.right = new Node(5);

tree\_level.right.left = new Node(6);

tree\_level.right.right = new Node(7);

tree\_level.left.left.left = new Node(8);

tree\_level.left.left.right = new Node(9);

// tree\_level.left.left.right.right = new Node(10);

// tree\_level.right.right.right = new Node(88);

/\*

Node tree\_level\_copy = new Node(1);

tree\_level\_copy.left = new Node(2);

tree\_level\_copy.right = new Node(3);

tree\_level\_copy.left.left = new Node(4);

tree\_level\_copy.left.right = new Node(5);

tree\_level\_copy.right.left = new Node(6);

tree\_level\_copy.right.right = new Node(7);

tree\_level\_copy.left.left.left = new Node(8);

tree\_level\_copy.left.left.right = new Node(9);

\*/

Node root = new Node(1);

root.left = new Node(2);

root.right = new Node(3);

root.left.left = new Node(4);

root.left.right = new Node(5);

root.left.left.left = new Node(8);

root.left.left.left.left = new Node(11);

root.left.left.left.left.right = new Node(13);

root.left.left.left.left.right.left = new Node(18);

root.left.left.left.left.right.left.left = new Node(19);//

root.left.right.right = new Node(9);

root.left.right.right.left = new Node(12);

root.left.right.right.left.left = new Node(14);

root.left.right.right.left.right = new Node(15);

root.left.right.right.left.right.right = new Node(16);

root.left.right.right.left.right.right.left = new Node(17);

root.left.right.right.right = new Node(20);//

root.left.right.right.right.right = new Node(21);//

root.left.right.right.right.right.right = new Node(22);//

root.left.right.right.right.right.right.right = new Node(23);//

root.right.left = new Node(6);

root.right.right = new Node(7);

root.right.right.right = new Node(10);

Node tree = new Node(20);

tree.left = new Node(8);

tree.right = new Node(22);

tree.left.left = new Node(5);

tree.left.right = new Node(3);

tree.right.left = new Node(4);

tree.right.right = new Node(25);

tree.left.right.left = new Node(10);

tree.left.right.right = new Node(14);

//bfs

// System.out.println("Level order traversal of binary tree is - ");

// ArrayList<Integer> al = levelOrder(tree\_level);

// Stack<Integer> ral = reverseLevelOrder(tree\_level);

// System.out.println("Reverse Level order traversal of binary tree is:");

// while(!ral.empty())

// System.out.print(ral.pop()+" ");

// leftView(tree\_level);

// rightView(tree\_level);

// leftView(root);

// rightView(root);

//dfs

// getHeight(tree\_level, -1);

// System.out.print("Height: " + height);

// System.out.println("\nPreorder: ");

// preOrderTraverse(root);

// System.out.println("\nPostorder: ");

// postOrderTraverse(root);

// System.out.println("\nInorder: ");

// inOrderTraverse(root);

// System.out.println("\nPreorder: ");

// preOrderTraverse(tree\_level);

// System.out.println("\nPostorder: ");

// postOrderTraverse(tree\_level);

// System.out.println("\nInorder: ");

// inOrderTraverse(tree\_level);

// diameter=-1;

// getDiameter(root);

// System.out.println("Diameter: " + diameter);

// diameter=-1;

// getDiameter(tree\_level);

// System.out.println("Diameter: " + diameter);

// System.out.println("\nMirror image of tree(Preorder): ");

// mirrorImage(tree\_level);

// preOrderTraverse(tree\_level);

// System.out.println("\nIs mirror image of tree(tree\_level,tree\_level): " + isMirrorImage(tree\_level,tree\_level));

// System.out.println("\nIs mirror image of tree(tree\_level,tree\_level\_copy): " + isMirrorImage(tree\_level,tree\_level\_copy));

// topView(tree\_level, 0);

// System.out.println("Top view = "+ topview.values().toString());

// topview.clear();

// topView(root, 0);

// System.out.println("Top view = "+ topview.values().toString());

// (new Main()).bottomView(tree,0);

// System.out.println("Bottom view = "+ bottomview.values().toString());

// isHeightBalanced(tree\_level,-1);

// System.out.println("tree is height Balanced? "+ balanced);

// printBoundaryNodes(tree\_level);

// System.out.println("\nSumTree(Preorder): ");

// convertToSumTree(tree\_level);

// preOrderTraverse(tree\_level);

// System.out.println("is sum tree: "+isSumTree(sumtree));

// String inorder = "8 4 9 2 5 1 6 3 7";

// String preorder = "1 2 4 8 9 5 3 6 7";

// String inorder = "11 19 18 13 8 4 2 5 14 12 15 17 16 9 20 21 22 23 1 6 3 7 10";

// String preorder = "1 2 4 8 11 13 18 19 5 9 12 14 15 16 17 20 21 22 23 3 6 7 10";

// Node treeFromTraversal = createTreeFromTraversal(inorder, preorder);

// int lvlarr[] = { 5, 6, 7, 8, 9, 10, 11 };

// System.out.println("\nRequired swaps: " + minSwapBSTFromCBT(lvlarr));

// Node lca = findLCAUtil(tree\_level, 8, 4);

// if(lca!=null){

// System.out.println("LCA(8,4): "+ lca.data);

// }else{

// System.out.println("LCA(8,4): null");

// }

//graph is tree or not? (non cyclic, connected): use a bool arr[no.of vertices] and traverse the graph, if false, set true, if already true, graph contain cycle. lastly, check if any vertex is false, if false, then not connected

// duplicate subtree in btree. dfs traverse tree that returns string of preorder traversal of node(for null node use a constant char). maintain a hashset of strings(string of preorder traversal at each node). everytime while inserting, check if string exists(duplicate- can maintain second arraylist for this), else insert into hashset.

//deletion of a node in bst: if leaf, remove directly. if it has one child, replace the key node with child, if it has two children, replace key node with inorder successor(uses recursion)

}

//Function to find lowest common ancestor if both node exists; null if both are absent; and returns the single node if other is not present

public static Node findLCAUtil(Node node, int n1, int n2){

if(node == null){

return null;

}

// System.out.println(node.data);

if(node.data==n1){

return node;

}

if(node.data==n2){

return node;

}

Node left\_lca = findLCAUtil(node.left, n1, n2);

Node right\_lca = findLCAUtil(node.right, n1, n2);

if(left\_lca!=null && right\_lca!=null){

return node;

}

return (left\_lca!=null)?(left\_lca):(right\_lca);

}

//Function to find min swaps for creating BST from complete BT

public static int minSwapBSTFromCBT(int[] lvlarr){

int size = lvlarr.length;

ArrayList<Integer> arrL = inorderFromLevelOrder(lvlarr, new ArrayList<Integer>(), 0, size);

int[] arr = new int[size];

int ind=0;

for(int elem : arrL){

arr[ind++]=elem;

}

int swaps = 0;

//Selection sort for minimum swaps

for(int i = 0; i<size-1; i++){

int min = arr[i];

int minindex = i;

int j = i+1;

for(; j<size; j++){

if(arr[j]<min){

min = arr[j];

minindex = j;

}

}

if(minindex!=i){

swaps++;

arr[minindex] = arr[i];

arr[i] = min;

}

}

return swaps;

}

//Func to create inorder traversal from lvl order traversal

public static ArrayList<Integer> inorderFromLevelOrder(int[] lvlarr, ArrayList<Integer> inarr, int index, int size){

if(index>=size){

return inarr;

}

inarr = inorderFromLevelOrder(lvlarr, inarr, 2\*index+1, size);

inarr.add(lvlarr[index]);

inarr = inorderFromLevelOrder(lvlarr, inarr, 2\*index+2, size);

return inarr;

}

//Function to create tree from inorder and postorder traversal

public static Node createTreeFromTraversal(String inorder, String preorder){

ArrayList<String> inStr = new ArrayList(Arrays.asList(inorder.split(" ")));

ArrayList<String> preStr = new ArrayList(Arrays.asList(preorder.split(" ")));

System.out.println("\ncreating tree..");

Node treeFromTraversal = createTreeFromTraversalUtil(inStr, preStr, null);

System.out.println("\npre");

preOrderTraverse(treeFromTraversal);

System.out.println("\npost");

postOrderTraverse(treeFromTraversal);

System.out.println("\nin");

inOrderTraverse(treeFromTraversal);

return treeFromTraversal;

}

public static Node createTreeFromTraversalUtil(ArrayList<String> inStr, ArrayList<String> preStr, Node node){

// int[] inOrderArr = Arrays.stream(inorder.split(" ")).mapToInt(Integer::parseInt).toArray();

// int[] preOrderArr = Arrays.stream(preorder.split(" ")).mapToInt(Integer::parseInt).toArray();

if(node == null && preStr.size()>0){

node = new Node(Integer.parseInt(preStr.get(0)));

int rootIndex = inStr.indexOf(preStr.get(0));

// System.out.println("For root index(inorder): "+rootIndex+" on data: " +node.data);

//left subtree

// System.out.println("creating left of : " + node.data + " using inStr index " + 0 + " - " + rootIndex + " and preStr index " + 1 + " - " + (rootIndex+1));

node.left = createTreeFromTraversalUtil(new ArrayList<String>(inStr.subList(0,rootIndex)), new ArrayList<String>(preStr.subList(1,rootIndex+1)), node.left);

//right subtree

// System.out.println("creating right of : " + node.data + " using inStr index " + (rootIndex+1) + " - " + inStr.size() + " and preStr index " + (rootIndex+1) + " - " + preStr.size());

node.right = createTreeFromTraversalUtil(new ArrayList<String>(inStr.subList(rootIndex+1, inStr.size())), new ArrayList<String>(preStr.subList(rootIndex+1, preStr.size())), node.right);

}

return node;

}

//Function to check if given tree is sum tree

public static boolean isSumTree(Node node){

if(node==null){

return true;

}else{

int result = sumTreeUtil(node);

if(result==node.data){

return true;

}

}

return false;

}

/\*

\* returns integer min value if false

\* returns sum of left & right subtree

\*/

public static int sumTreeUtil(Node node){

if(node.left==null&&node.right==null){

return node.data;

}

//store sum of left+right subtree data

int data = 0;

//store left/right subtree data

int ldata = 0;

int rdata = 0;

if(node.left!=null){

ldata=sumTreeUtil(node.left);

}

if(node.right!=null){

rdata=sumTreeUtil(node.right);

}

data=ldata+rdata;

if(data==node.data){

return data;

}else{

return Integer.MIN\_VALUE;

}

}

//Function to convert to sum tree

public static void convertToSumTree(Node node){

if(node==null){

return;

}

//store sum of new left+right subtree data

int data = 0;

//store old left/right node data

int ldata = 0;

int rdata = 0;

if(node.left!=null){

ldata=node.left.data;

convertToSumTree(node.left);

data = data + node.left.data;

}

if(node.right!=null){

rdata=node.right.data;

convertToSumTree(node.right);

data = data + node.right.data;

}

//in case of leaf nodes, all three are zeroes

//in case of non-leaf nodes, store old left node data+old right node data+new left node data+new right node data

node.data = ldata+rdata+data;

}

//Function to print boundary nodes of tree.

public static void printBoundaryNodes(Node node){

if(node!=null){

System.out.print(node.data);

}

printLeftBoundary(node.left);

printLeafNodes(node);

printRightBoundary(node.right);

System.out.println();

}

public static void printLeftBoundary(Node node){

if(node==null){

return;

}

if(node.left==null && node.right==null){

return;

}

System.out.print(" "+node.data);

if(node.left!=null){

printLeftBoundary(node.left);

} else if(node.right!=null){

printLeftBoundary(node.right);

}

}

public static void printRightBoundary(Node node){

if(node==null){

return;

}

if(node.left==null && node.right==null){

return;

}

if(node.right!=null){

printRightBoundary(node.right);

} else if(node.left!=null){

printRightBoundary(node.left);

}

System.out.print(" "+node.data);

}

public static void printLeafNodes(Node node){

if(node==null){

return;

}

if(node.left==null && node.right==null){

System.out.print(" "+node.data);

}

if(node.left!=null){

printLeafNodes(node.left);

}

if(node.right!=null){

printLeafNodes(node.right);

}

}

//Function to return if tree is height balanced.

public static int isHeightBalanced(Node node, int currHeight)

{

if(node==null){

return -1;

} else if(node.left==null && node.right==null){

return 0;

}

int lh=0;

int rh=0;

if(node.left!=null){

lh = getHeight(node.left,currHeight)+1;

}

if(node.right!=null){

rh = getHeight(node.right,currHeight)+1;

}

currHeight = lh>rh?lh:rh;

if(!(lh==rh || lh+1==rh || lh==rh+1)){

balanced = false;

}

return currHeight;

}

//Function for bottom view of a tree.

public void bottomView(Node node, int pos)

{

class NodePos{

Node node;

int pos;

public NodePos(Node node, int pos){

this.node=node;

this.pos=pos;

}

};

Queue<NodePos> q= new LinkedList<>();

q.add(new NodePos(node,pos));

while(!q.isEmpty()){

NodePos cur = q.poll();

if(cur.node!=null){

if(cur.node.left!=null){

q.add(new NodePos(cur.node.left,cur.pos-1));

}

if(cur.node.right!=null){

q.add(new NodePos(cur.node.right,cur.pos+1));

}

bottomview.put(cur.pos,cur.node.data);

}

}

}

//Function for top view of a tree.

public static void topView(Node node, int pos)

{

if(node==null)

return;

if(node.left!=null){

topView(node.left, pos-1);

}

if(node.right!=null){

topView(node.right, pos+1);

}

topview.put(pos,node.data);

}

//Function to print the left view of a tree.

public static void leftView(Node node)

{

System.out.println("Left View");

Queue<Node> qNode= new LinkedList<>();

Queue<Integer> qLevel= new LinkedList<>();

qNode.add(node);

qLevel.add(0);

int printLevel = 0;

while(!qNode.isEmpty()){

Node cur = qNode.poll();

int curLevel = qLevel.poll();

if(cur!=null){

if(curLevel==printLevel){

System.out.print(cur.data+" ");

printLevel++;

}

if(cur.left!=null){

qNode.add(cur.left);

qLevel.add(curLevel+1);

}

if(cur.right!=null){

qNode.add(cur.right);

qLevel.add(curLevel+1);

}

}

}

System.out.println();

return;

}

//Function to print the right view of a tree.

public static void rightView(Node node)

{

System.out.println("Right View");

Queue<Node> qNode= new LinkedList<>();

Queue<Integer> qLevel= new LinkedList<>();

qNode.add(node);

qLevel.add(0);

int printLevel = 0;

while(!qNode.isEmpty()){

Node cur = qNode.poll();

int curLevel = qLevel.poll();

if(cur!=null){

if(curLevel==printLevel){

System.out.print(cur.data+" ");

printLevel++;

}

if(cur.right!=null){

qNode.add(cur.right);

qLevel.add(curLevel+1);

}

if(cur.left!=null){

qNode.add(cur.left);

qLevel.add(curLevel+1);

}

}

}

System.out.println();

return;

}

//Function to check if trees are mirror images.

public static boolean isMirrorImage(Node node1, Node node2)

{

if(node1==null && node2==null){

return true;

} else if(node1==null||node2==null){

return false;

} else {

// System.out.println("comparing values "+node1.data+" and "+node2.data);

if(node1.data == node2.data

&& isMirrorImage(node1.left, node2.right)

&& isMirrorImage(node1.right, node2.left)){

return true;

}

}

return false;

}

//Function to return the mirror image of a tree.

public static Node mirrorImage(Node node)

{

if(node==null){

return null;

} else if(node.left==null && node.right==null){

return node;

} else {

Node temp = node.left;

node.left = node.right;

node.right = temp;

}

if(node.left!=null){

node.left=mirrorImage(node.left);

}

if(node.right!=null){

node.right=mirrorImage(node.right);

}

return node;

}

//Function to return the diameter/width of a tree.

public static int getDiameter(Node node)

{

if(node==null){

return 0;

} else if(node.left==null && node.right==null){

return 1;

}

int ld=0;

int rd=0;

if(node.left!=null){

ld=getDiameter(node.left);

}

if(node.right!=null){

rd=getDiameter(node.right);

}

diameter = Math.max(diameter,ld+rd+1);

return (ld>rd)?(ld+1):(rd+1);

}

//Function to return the pre-order traversal of a tree.

public static void preOrderTraverse(Node node)

{

if(node==null)

return;

System.out.print(node.data+" ");

if(node.left!=null)

preOrderTraverse(node.left);

if(node.right!=null)

preOrderTraverse(node.right);

}

//Function to return the post-order traversal of a tree.

public static void postOrderTraverse(Node node)

{

if(node==null)

return;

if(node.left!=null)

postOrderTraverse(node.left);

if(node.right!=null)

postOrderTraverse(node.right);

System.out.print(node.data+" ");

}

//Function to return the in-order traversal of a tree.

public static void inOrderTraverse(Node node)

{

if(node==null)

return;

if(node.left!=null)

inOrderTraverse(node.left);

System.out.print(node.data+" ");

if(node.right!=null)

inOrderTraverse(node.right);

}

//Function to return the height/depth of a tree.

public static int getHeight(Node node, int currHeight)

{

if(node==null){

return -1;

} else if(node.left==null && node.right==null){

return 0;

}

int lh=0;

int rh=0;

if(node.left!=null){

lh = getHeight(node.left,currHeight)+1;

}

if(node.right!=null){

rh = getHeight(node.right,currHeight)+1;

}

currHeight = lh>rh?lh:rh;

height = currHeight>height?currHeight:height;

return currHeight;

}

//Function to return the level order traversal of a tree.

public static ArrayList <Integer> levelOrder(Node node)

{

ArrayList<Integer> al = new ArrayList<>();

Queue<Node> q= new LinkedList<>();

q.add(node);

while(!q.isEmpty()){

Node cur = q.peek();

if(cur!=null){

al.add(cur.data);

System.out.print(cur.data+" ");

q.poll();

if(cur.left!=null){

q.add(cur.left);

}

if(cur.right!=null){

q.add(cur.right);

}

}

}

return al;

}

//Function to return the reverse level order traversal of a tree.

public static Stack<Integer> reverseLevelOrder(Node node)

{

Stack<Integer> s = new Stack<>();

Queue<Node> q= new LinkedList<>();

q.add(node);

while(!q.isEmpty()){

Node cur = q.poll();

if(cur!=null){

if(cur.right!=null){

q.add(cur.right);

}

if(cur.left!=null){

q.add(cur.left);

}

s.push(cur.data);

}

}

System.out.println();

return s;

}

}

***GRAPH***

import java.io.\*;

import java.util.\*;

class Graph{

int v;

boolean isDirected;

ArrayList<Integer> adj[];

Graph(int vertices, boolean isDirected){

this.v = vertices;

this.isDirected = isDirected;

adj = new ArrayList[v];

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

adj[i] = new ArrayList<Integer>();

}

}

public void addEdge(int v1, int v2){

adj[v1].add(v2);

if(!isDirected){

adj[v2].add(v1);

}

}

//bfs traversal+cycle detection

public void bfs(){

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

Boolean visited[] = new Boolean[v];

Arrays.fill(visited, Boolean.FALSE);

System.out.println("BFS:");

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

if(!visited[i]) {

q.add(i);

bfsUtil(q, visited);

}

}

System.out.println();

}

public void bfsUtil(Queue<Integer> q, Boolean visited[]){

while(!q.isEmpty()){

if(visited[q.peek()]){

if(!isDirected){

// System.out.println("cycle detected at : "+q.peek());

}

q.poll();

continue;

}

int curr = q.poll();

visited[curr] = true;

System.out.print(curr + " ");

for(int val: adj[curr]){

if(!visited[val]){

q.add(val);

}

else if(isDirected){

// System.out.println("cycle detected(directed graph) at : "+val);

}

}

}

}

//DFS traversal - iteration method + cycle detection(directed graph)

public void dfsIteration(){

Stack<Integer> s = new Stack<>();

Boolean visited[] = new Boolean[v];

Arrays.fill(visited, Boolean.FALSE);

System.out.println("DFS(non-recursive):");

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

s.push(i);

dfsIterationUtil(s, visited);

}

System.out.println();

}

public void dfsIterationUtil(Stack<Integer> s, Boolean[] visited){

while(!s.isEmpty()){

if(visited[s.peek()]){

return;

}

int curr = s.pop();

System.out.print(curr+" ");

visited[curr] = true;

for(int val: adj[curr]){

if(!visited[val]){

s.push(val);

}

else if(isDirected){

// System.out.println("cycle detected at : "+val);

}

}

}

}

//DFS traversal using recursion + cycle detection(directed graph)

public void dfsRec(){

Boolean visited[] = new Boolean[v];

Arrays.fill(visited, Boolean.FALSE);

System.out.println("DFS:");

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

dfsRecUtil(i, visited);

}

System.out.println();

}

public void dfsRecUtil(int curr, Boolean visited[]){

if(visited[curr]){

return;

}

visited[curr] = true;

System.out.print(curr + " ");

for(int val: adj[curr]){

if(!visited[val]){

dfsRecUtil(val,visited);

}

else if(isDirected){

// System.out.println("cycle detected at : "+val);

}

}

}

public Graph cloneGraph(){

Graph clone = new Graph(v, isDirected);

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

System.out.println("Clone using BFS");

bfs();

dfsIteration();

dfsRec();

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

q.add(i);

cloneUtil(q, clone);

}

clone.bfs();

clone.dfsIteration();

clone.dfsRec();

return clone;

}

public void cloneUtil(Queue<Integer> q, Graph clone){

while(!q.isEmpty()){

int curr = q.poll();

for(int val: adj[curr]){

clone.addEdge(curr,val);

}

}

}

}

/\*class MinStepsTargetKnight{

//not working

static int row[] = {1, 2, -1, 1, -2, 2, -1, -2};

static int col[] = {2, 1, 2, -2, 1, -1, -2, -1};

static int minSteps = Integer.MAX\_VALUE;

class Loc{

int r,c;

public Loc(int i, int j){

r=i;

c=j;

}

}

static Set<Loc> visited = new HashSet<>();

public static void minStepsTargetKnight(){

int N = 30;

int knightPos[] = { 1, 1 };

int targetPos[] = { 30, 30 };

// Queue<>

MinStepsTargetKnight obj = new MinStepsTargetKnight();

int steps = obj.minStepsTargetKnight(knightPos, targetPos, N, 0);

System.out.println("Min steps to reach target knight= " + steps);

}

public int minStepsTargetKnight(int[] knightPos, int[] targetPos, int N, int steps){

if(!isAllowed(knightPos[0], knightPos[1], N)

|| visited.contains(new Loc(knightPos[0], knightPos[1]))){

visited.add(new Loc(knightPos[0], knightPos[1]));

return -1;

}

if(knightPos[0] == targetPos[0] && knightPos[1] == targetPos[1]){

System.out.println("Reached in steps: " + steps);

if(steps>=0 && minSteps>steps){

minSteps = steps;

}

return steps;

}

for(int i=0; i<8; i++){

System.out.println("old - i: "+knightPos[0]+", j: "+knightPos[1]);

knightPos[0]+=row[i];

knightPos[1]+=col[i];

System.out.println("new - i: "+knightPos[0]+", j: "+knightPos[1]);

if(isAllowed(knightPos[0],knightPos[1],N)

&& !visited.contains(new Loc(knightPos[0], knightPos[1]))){

// int step =

minStepsTargetKnight(knightPos,targetPos,N,steps+1);

// if(step>=0 && minSteps>step){

// minSteps = step;

// }

}else{

System.out.println("Repeat/Invalid");

}

knightPos[0]-=row[i];

knightPos[1]-=col[i];

}

visited.add(new Loc(knightPos[0], knightPos[1]));

return minSteps;

}

public int minStepsTargetKnight1(int[] knightPos, int[] targetPos, int N, int steps){

if(!isAllowed(knightPos[0], knightPos[1], N)

|| visited.contains(new Loc(knightPos[0], knightPos[1]))){

visited.add(new Loc(knightPos[0], knightPos[1]));

return -1;

}

if(knightPos[0] == targetPos[0] && knightPos[1] == targetPos[1]){

System.out.println("Reached in steps: " + steps);

if(steps>=0 && minSteps>steps){

minSteps = steps;

}

return steps;

}

for(int i=0; i<8; i++){

System.out.println("old - i: "+knightPos[0]+", j: "+knightPos[1]);

knightPos[0]+=row[i];

knightPos[1]+=col[i];

System.out.println("new - i: "+knightPos[0]+", j: "+knightPos[1]);

if(isAllowed(knightPos[0],knightPos[1],N)

&& !visited.contains(new Loc(knightPos[0], knightPos[1]))){

// int step =

minStepsTargetKnight(knightPos,targetPos,N,steps+1);

// if(step>=0 && minSteps>step){

// minSteps = step;

// }

}else{

System.out.println("Repeat/Invalid");

}

knightPos[0]-=row[i];

knightPos[1]-=col[i];

}

visited.add(new Loc(knightPos[0], knightPos[1]));

return minSteps;

}

public static boolean isAllowed(int i, int j, int N){

if(i<=0 || j<=0 || i>N || j>N){

return false;

}

return true;

}

}\*/

class RatMaze{

//Rat in a maze

static List<String> s = new ArrayList<>();

static StringBuffer path = new StringBuffer();

public static void ratInMaze(){

int m[][] = { { 1, 0, 0, 0, 0 },

{ 1, 1, 1, 1, 1 },

{ 1, 1, 1, 0, 1 },

{ 0, 0, 0, 0, 1 },

{ 0, 0, 0, 0, 1 } };

int m2[][] = { { 1, 0, 0, 0 },

{ 1, 1, 0, 1 },

{ 0, 1, 0, 0 },

{ 1, 1, 1, 1 } };

int m1[][] = { { 1, 0, 0, 0 },

{ 1, 1, 1, 1 },

{ 0, 1, 0, 1 },

{ 1, 1, 1, 1 } };

int len = m.length;

Boolean visited[][] = new Boolean[len][len];

for(int x=0;x<len;x++){

Arrays.fill(visited[x],Boolean.FALSE);

}

ratInMaze(m, 0, 0, len, visited);

for(String p: s){

System.out.println(p);

}

}

public static void ratInMaze(int[][] m, int i, int j, int len, Boolean[][] visited){

if(!isAllowed(m, i, j, len, visited)){

return;

}

if(i==len-1 && j==len-1){

// System.out.println("reached destination using "+path);

s.add(path.toString());

return;

}

visited[i][j] = true;

if(isAllowed(m,i+1,j,len,visited)){

path.append("D");

ratInMaze(m,i+1,j,len,visited);

path.deleteCharAt(path.length()-1);

}

if(isAllowed(m,i,j+1,len,visited)){

path.append("R");

ratInMaze(m,i,j+1,len,visited);

path.deleteCharAt(path.length()-1);

}

if(isAllowed(m,i-1,j,len,visited)){

path.append("U");

ratInMaze(m,i-1,j,len,visited);

path.deleteCharAt(path.length()-1);

}

if(isAllowed(m,i,j-1,len,visited)){

path.append("L");

ratInMaze(m,i,j-1,len,visited);

path.deleteCharAt(path.length()-1);

}

visited[i][j]=false;

return;

}

public static boolean isAllowed(int[][] m, int i, int j, int len, Boolean[][] visited){

if(i<0 || i>=len || j<0 || j>=len || visited[i][j] || m[i][j]==0){

return false;

}

return true;

}

}

public class Main {

public static void main(String args[]){

//connected undirected graph

Graph g1 = new Graph(6,false);

g1.addEdge(0,1);

g1.addEdge(0,2);

g1.addEdge(1,3);

g1.addEdge(1,4);

g1.addEdge(2,4);

g1.addEdge(3,4);

g1.addEdge(3,5);

g1.addEdge(4,5);

//disconnected directed graph

Graph g2 = new Graph(9,true);

g2.addEdge(0, 1);

g2.addEdge(1, 2);

g2.addEdge(1, 3);

g2.addEdge(2, 4);

g2.addEdge(2, 5);

g2.addEdge(4, 2);

g2.addEdge(6, 8);

// g1.bfs();

// g1.dfsRec();

// g1.dfsIteration();

// g2.bfs();

// g2.dfsRec();

// g2.dfsIteration();

Graph clone = g1.cloneGraph();

// RatMaze.ratInMaze();

/\*//not working

MinStepsTargetKnight.minStepsTargetKnight();\*/

}

}

import java.io.\*;

import java.util.\*;

public class Main {

public static void main(String args[]){

int graph[][] = new int[][] { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },

{ 4, 0, 8, 0, 0, 0, 0, 11, 0 },

{ 0, 8, 0, 7, 0, 4, 0, 0, 2 },

{ 0, 0, 7, 0, 9, 14, 0, 0, 0 },

{ 0, 0, 0, 9, 0, 10, 0, 0, 0 },

{ 0, 0, 4, 14, 10, 0, 2, 0, 0 },

{ 0, 0, 0, 0, 0, 2, 0, 1, 6 },

{ 8, 11, 0, 0, 0, 0, 1, 0, 7 },

{ 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

dijkstra(graph, 0);

}

public static void dijkstra(int[][] graph, int src){

int v = graph.length;

int dist[] = new int[v];

boolean covered[] = new boolean[v];

int min = Integer.MAX\_VALUE;

int minIndex = -1;

// int mindist = Integer.MAX\_VALUE;

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

dist[i] = Integer.MAX\_VALUE;

}

dist[src]=0;

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

//mindistance

min = Integer.MAX\_VALUE;

minIndex = -1;

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

if(!covered[j] && min >= dist[j]){

min = dist[j];

minIndex = j;

}

}

covered[minIndex] = true;

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

if(!covered[j] && dist[minIndex] != Integer.MAX\_VALUE && graph[minIndex][j] != 0

&& ((graph[minIndex][j] + dist[minIndex]) < dist[j])){

dist[j] = dist[minIndex] + graph[minIndex][j];

}

}

}

//print soln.

System.out.println("Vertex \t\t Distance from Source");

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

System.out.println(i + " \t\t " + dist[i]);

}

}

}

***Heap***

import java.io.\*;

import java.util.\*;

class Heap{

int arr[];

int maxSize;

int curSize=0;

public Heap(int maxSize){

this.maxSize = maxSize;

arr = new int[maxSize];

Arrays.fill(arr, Integer.MIN\_VALUE);

}

public void push(int elem){

if(maxSize<=curSize){

System.out.println("Size Limit Reached.");

return;

}

System.out.println("Pushing "+elem+" at index "+curSize+".");

arr[curSize++] = elem;

// System.out.println("Size increased to "+curSize);

revHeapifyUtil();

}

public int popMax(){

int popped = arr[0];

arr[0] = arr[curSize-1];

arr[curSize-1] = Integer.MIN\_VALUE;

curSize--;

revHeapifyUtil();

return popped;

}

//heapify last node to root node iterating only for parent nodes

public void revHeapifyUtil(){

if(curSize<2){

return;

}

for(int i = (curSize/2-1); i>0;){

Main.heapify(arr, curSize, i, true);

i=(i-1)/2;

}

Main.heapify(arr, curSize, 0, true);

}

}

public class Main

{

public static void main(String[] args) {

System.out.println("HEAP");

// int arr[] = { 1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17 };

// int n = arr.length;

// buildHeap(arr, n, true);

// heapsort(arr, n, true);

// ArrayList<Integer> al = new ArrayList<>();

// int k = 3;

// slidingWindowMax(arr, n, al, k);

Heap mxHeap = new Heap(6);

mxHeap.push(7);

mxHeap.push(10);

mxHeap.push(4);

mxHeap.push(3);

mxHeap.push(20);

mxHeap.push(15);

printHeap(mxHeap.arr, mxHeap.curSize);

System.out.println("popped max elem: "+mxHeap.popMax());

printHeap(mxHeap.arr, mxHeap.curSize);

}

public static void slidingWindowMax(int[] arr, int n, ArrayList<Integer> al, int k){

if(k>n){

System.out.println("invalid k value...");

return;

}

int window[] = new int[k];

// initializing with k elements

for(int i=0; i<=n-k; i++){

for(int j=0; j<k; j++){

window[j] = arr[i+j];

}

buildHeap(window, k, true);

al.add(window[0]);

}

System.out.println(al);

}

public static void heapsort(int arr[], int n, boolean increasingOrder){

printHeap(arr, n);

for(int i=n-1; i>0; i--){

buildHeap(arr, i, increasingOrder);

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

}

printHeap(arr, n);

}

public static void buildHeap(int arr[], int n, boolean isMaxheap){

printHeap(arr, n);

int lastNonLeafIndex = n/2-1;

// System.out.println("last non leaf node: "+arr[lastNonLeafIndex]);

for(int i=lastNonLeafIndex; i>=0; i--){

heapify(arr, n, i, isMaxheap);

}

printHeap(arr, n);

}

public static void heapify(int arr[], int n, int index, boolean isMaxheap){

int lChild = 2\*index+1;

int rChild = 2\*index+2;

int lval,rval;

if(index>=n/2){

return;

}

if(lChild>=n){

// System.out.println("Left child does not exist.");

} else if(rChild>=n){

// System.out.println("Right child does not exist.");

}

// System.out.println("traverse for node: "+arr[index]);

// System.out.println("left: "+arr[lChild]+", right: "+arr[rChild]);

if(isMaxheap){

lval = (lChild<n)?arr[lChild]:Integer.MIN\_VALUE;

rval = (rChild<n)?arr[rChild]:Integer.MIN\_VALUE;

if(arr[index]<lval || arr[index]<rval){

if(lval>rval){

// System.out.println("replace with left child");

int tmp = arr[lChild];

arr[lChild] = arr[index];

arr[index] = tmp;

heapify(arr, n, lChild, isMaxheap);

} else {

// System.out.println("replace with right child");

int tmp = arr[rChild];

arr[rChild] = arr[index];

arr[index] = tmp;

heapify(arr, n, rChild, isMaxheap);

}

}

} else {

lval = (lChild<n)?arr[lChild]:Integer.MAX\_VALUE;

rval = (rChild<n)?arr[rChild]:Integer.MAX\_VALUE;

if(arr[index]>lval || arr[index]>rval){

if(lval<rval){

// System.out.println("replace with left child");

int tmp = arr[lChild];

arr[lChild] = arr[index];

arr[index] = tmp;

heapify(arr, n, lChild, isMaxheap);

} else {

// System.out.println("replace with right child");

int tmp = arr[rChild];

arr[rChild] = arr[index];

arr[index] = tmp;

heapify(arr, n, rChild, isMaxheap);

}

}

}

}

public static void printHeap(int[] arr, int n){

System.out.print("Array representation of heap: ");

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

System.out.print(arr[i]+" ");

}

System.out.println();

}

}