1.A

class TreeNode {

int value;

TreeNode left;

TreeNode right;

TreeNode(int value) {

this.value = value;

}

}

class Result {

boolean isBalanced;

int height;

Result(boolean isBalanced, int height) {

this.isBalanced = isBalanced;

this.height = height;

}

}

public class BalancedBinaryTree {

public static boolean isBalanced(TreeNode root) {

Result result = isBalancedRecursive(root, 0);

return result.isBalanced;

}

private static Result isBalancedRecursive(TreeNode node, int depth) {

if (node == null) {

return new Result(true, -1);

}

Result leftSubtreeResult = isBalancedRecursive(node.left, depth + 1);

Result rightSubtreeResult = isBalancedRecursive(node.right, depth + 1);

boolean isBalanced = Math.abs(leftSubtreeResult.height - rightSubtreeResult.height) <= 1;

boolean subtreesAreBalanced = leftSubtreeResult.isBalanced && rightSubtreeResult.isBalanced;

int height = Math.max(leftSubtreeResult.height, rightSubtreeResult.height) + 1;

return new Result(isBalanced && subtreesAreBalanced, height);

}

public static void main(String[] args) {

// Example usage:

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

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

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

System.out.println(isBalanced(root)); // Output: true

}

}

2.A

class TrieNode {

TrieNode[] children;

boolean isEndOfWord;

TrieNode() {

children = new TrieNode[26];

isEndOfWord = false;

}

}

public class Trie {

private TrieNode root;

public Trie() {

root = new TrieNode();

}

public void insert(String word) {

TrieNode curr = root;

for (char c : word.toCharArray()) {

int index = c - 'a';

if (curr.children[index] == null) {

curr.children[index] = new TrieNode();

}

curr = curr.children[index];

}

curr.isEndOfWord = true;

}

public boolean isPrefix(String prefix) {

TrieNode curr = root;

for (char c : prefix.toCharArray()) {

int index = c - 'a';

if (curr.children[index] == null) {

return false;

}

curr = curr.children[index];

}

return true;

}

}

public class Main {

public static void main(String[] args) {

Trie trie = new Trie();

trie.insert("hello");

trie.insert("world");

trie.insert("hey");

trie.insert("hi");

System.out.println(trie.isPrefix("he")); // true

System.out.println(trie.isPrefix("hel")); // true

System.out.println(trie.isPrefix("hey")); // true

System.out.println(trie.isPrefix("hell")); // false

System.out.println(trie.isPrefix("abc")); // false

}

}

3.A

import java.util.ArrayList;

class MinHeap {

private ArrayList<Integer> heap;

private int size;

private int maxsize;

private static final int FRONT = 1;

public MinHeap(int maxsize) {

this.maxsize = maxsize;

this.heap = new ArrayList<>(this.maxsize + 1);

this.heap.add(Integer.MIN\_VALUE); // Sentinel value at index 0

this.size = 0;

}

private int parent(int pos) {

return pos / 2;

}

private int leftChild(int pos) {

return 2 \* pos;

}

private int rightChild(int pos) {

return 2 \* pos + 1;

}

private boolean isLeaf(int pos) {

return pos > size / 2;

}

private void swap(int fpos, int spos) {

int tmp = heap.get(fpos);

heap.set(fpos, heap.get(spos));

heap.set(spos, tmp);

}

private void minHeapify(int pos) {

if (!isLeaf(pos)) {

int swapPos = pos;

if (rightChild(pos) <= size) {

swapPos = (heap.get(leftChild(pos)) < heap.get(rightChild(pos))) ? leftChild(pos) : rightChild(pos);

} else {

swapPos = leftChild(pos);

}

if (heap.get(pos) > heap.get(leftChild(pos)) || heap.get(pos) > heap.get(rightChild(pos))) {

swap(pos, swapPos);

minHeapify(swapPos);

}

}

}

public void insert(int element) {

if (size >= maxsize) {

return; // Heap is full

}

heap.add(++size, element);

int current = size;

while (heap.get(current) < heap.get(parent(current))) {

swap(current, parent(current));

current = parent(current);

}

}

public int extractMin() {

if (isEmpty()) {

throw new RuntimeException("Heap is empty");

}

int min = heap.get(FRONT);

heap.set(FRONT, heap.get(size--));

minHeapify(FRONT);

return min;

}

public boolean isEmpty() {

return size == 0;

}

public void print() {

for (int i = 1; i <= size / 2; i++) {

System.out.print("PARENT: " + heap.get(i) +

" LEFT CHILD: " + heap.get(2 \* i) +

" RIGHT CHILD: " + heap.get(2 \* i + 1));

System.out.println();

}

}

public static void main(String[] args) {

MinHeap minHeap = new MinHeap(15);

minHeap.insert(5);

minHeap.insert(3);

minHeap.insert(10);

minHeap.insert(8);

System.out.println("Extracted Min: " + minHeap.extractMin());

System.out.println("Extracted Min: " + minHeap.extractMin());

}

}

4.A

import java.util.\*;

class Graph {

private int V;

private LinkedList<Integer> adj[];

Graph(int v) {

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList();

}

void addEdge(int v, int w) {

adj[v].add(w);

}

private boolean isCyclicUtil(int v, boolean visited[],

boolean recStack[]) {

if (recStack[v])

return true;

if (visited[v])

return false;

visited[v] = true;

recStack[v] = true;

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

while (i.hasNext()) {

int next = i.next();

if (isCyclicUtil(next, visited, recStack))

return true;

}

recStack[v] = false;

return false;

}

boolean isCyclic() {

boolean visited[] = new boolean[V];

boolean recStack[] = new boolean[V];

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

if (isCyclicUtil(i, visited, recStack))

return true;

return false;

}

}

public class GraphEdgeValidation {

static boolean addEdgeAndValidate(Graph graph, int src, int dest) {

graph.addEdge(src, dest);

if (graph.isCyclic()) {

// If cycle detected, remove the added edge

graph.adj[src].removeLast();

return false;

}

return true;

}

public static void main(String args[]) {

Graph graph = new Graph(4);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 2);

graph.addEdge(2, 0);

graph.addEdge(2, 3);

graph.addEdge(3, 3);

int src = 2, dest = 1;

if (addEdgeAndValidate(graph, src, dest))

System.out.println("Edge (" + src + " -> " + dest + ") added successfully!");

else

System.out.println("Edge (" + src + " -> " + dest + ") cannot be added as it creates a cycle!");

src = 1;

dest = 3;

if (addEdgeAndValidate(graph, src, dest))

System.out.println("Edge (" + src + " -> " + dest + ") added successfully!");

else

System.out.println("Edge (" + src + " -> " + dest + ") cannot be added as it creates a cycle!");

}

}

5.A

import java.util.\*;

class Graph {

private int V; // Number of vertices

private LinkedList<Integer> adj[];

Graph(int v) {

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList();

}

void addEdge(int v, int w) {

adj[v].add(w);

adj[w].add(v);

}

void BFS(int start) {

boolean visited[] = new boolean[V];

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

visited[start] = true;

queue.add(start);

while (!queue.isEmpty()) {

start = queue.poll();

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

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

while (i.hasNext()) {

int n = i.next();

if (!visited[n]) {

visited[n] = true;

queue.add(n);

}

}

}

}

}

public class BFSTraversal {

public static void main(String args[]) {

Graph graph = new Graph(4);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 2);

graph.addEdge(2, 3);

System.out.println("BFS traversal starting from node 2:");

graph.BFS(2);

}

}

6.A

import java.util.\*;

class Graph {

private int V;

private LinkedList<Integer> adj[];

Graph(int v) {

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList();

}

void addEdge(int v, int w) {

adj[v].add(w);

adj[w].add(v); // For undirected graph

}

void DFSUtil(int v, boolean visited[]) {

visited[v] = true;

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

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

while (i.hasNext()) {

int n = i.next();

if (!visited[n])

DFSUtil(n, visited);

}

}

void DFS() {

boolean visited[] = new boolean[V];

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

if (!visited[i])

DFSUtil(i, visited);

}

}

}

public class DFSTraversal {

public static void main(String args[]) {

Graph graph = new Graph(4);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 2);

graph.addEdge(2, 3);

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

graph.DFS();

}

}