Day 18 Assignment

Name: Mehul Anjikhane Email: mehulanjikhane13@gmail.com

**Task 1: Balanced Binary Tree Check**

**Write a function to check if a given binary tree is balanced. A balanced tree is one where the height of two subtrees of any node never differs by more than one.**

**package** tree;

**public** **class** TreeNode {

**int** val;

TreeNode left;

TreeNode right;

TreeNode(**int** val) {

**this**.val = val;

**this**.left = **this**.right = **null**;

}

}

**public** **class** BalancedTree {

**public** **boolean** isBalanced(TreeNode root) {

**return** checkHeight(root) != -1;

}

**private** **int** checkHeight(TreeNode node) {

**if** (node == **null**)

**return** 0;

**int** leftHeight = checkHeight(node.left);

**if** (leftHeight == -1)

**return** -1;

**int** rightHeight = checkHeight(node.right);

**if** (rightHeight == -1)

**return** -1;

**if** (Math.*abs*(leftHeight - rightHeight) > 1) {

**return** -1;

}

**return** Math.*max*(leftHeight, rightHeight) + 1;

}

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

TreeNode root = **new** TreeNode(1);

root.left = **new** TreeNode(2);

root.right = **new** TreeNode(2);

root.left.left = **new** TreeNode(3);

root.left.right = **new** TreeNode(3);

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

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

BalancedTree tree = **new** BalancedTree();

**boolean** result = tree.isBalanced(root);

System.***out***.println("Is the tree balanced? " + result);

}

}

**Output:**

Is the tree balanced? False

**Task 2: Tree for Prefix Checking**

**Implement a tree data structure in Java that supports insertion of strings and provides a method to check if a given string is a prefix of any word in the tree.**

**package tree;**

**import java.util.HashMap;**

**import java.util.Map;**

**public class TreeNodes {**

**Map<Character, TreeNodes> children;**

**boolean isEndOfWord;**

**public TreeNodes() {**

**children = new HashMap<>();**

**isEndOfWord = false;**

**}**

**}**

**public** **class** CheckPrefixInTree {

**private** TreeNodes root;

**public** CheckPrefixInTree() {

root = **new** TreeNodes();

}

**public** **void** insert(String word) {

TreeNodes node = root;

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

node = node.children.computeIfAbsent(c, k -> **new** TreeNodes());

}

node.isEndOfWord = **true**;

}

**public** **boolean** isPrefix(String prefix) {

TreeNodes node = root;

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

node = node.children.get(c);

**if** (node == **null**) {

**return** **false**;

}

}

**return** **true**;

}

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

CheckPrefixInTree tree = **new** CheckPrefixInTree();

tree.insert("apple");

tree.insert("app");

tree.insert("apricot");

System.***out***.println("Is 'app' a prefix? " + tree.isPrefix("app"));

System.***out***.println("Is 'apl' a prefix? " + tree.isPrefix("apl"));

}

}

**Output:**

Is 'app' a prefix? true

Is 'apl' a prefix? false

**Task 3: Implementing Heap Operations**

**Code a min-heap in Java with methods for insertion, deletion, and fetching the minimum element. Ensure that the heap property is maintained after each operation.**

package heap;

import java.util.ArrayList;

import java.util.List;

public class MinHeap {

private List<Integer> heap;

public MinHeap() {

heap = new ArrayList<>();

}

public void insert(int val) {

heap.add(val);

int index = heap.size() - 1;

heapifyUp(index);

}

public int deleteMin() {

if (heap.isEmpty())

throw new IllegalStateException("Heap is empty");

int min = heap.get(0);

int last = heap.remove(heap.size() - 1);

if (!heap.isEmpty()) {

heap.set(0, last);

heapifyDown(0);

}

return min;

}

public int getMin() {

if (heap.isEmpty())

throw new IllegalStateException("Heap is empty");

return heap.get(0);

}

private void heapifyUp(int index) {

int parent = (index - 1) / 2;

if (index > 0 && heap.get(index) < heap.get(parent)) {

swap(index, parent);

heapifyUp(parent);

}

}

private void heapifyDown(int index) {

int left = 2 \* index + 1;

int right = 2 \* index + 2;

int smallest = index;

if (left < heap.size() && heap.get(left) < heap.get(smallest)) {

smallest = left;

}

if (right < heap.size() && heap.get(right) < heap.get(smallest)) {

smallest = right;

}

if (smallest != index) {

swap(index, smallest);

heapifyDown(smallest);

}

}

private void swap(int i, int j) {

int temp = heap.get(i);

heap.set(i, heap.get(j));

heap.set(j, temp);

}

public static void main(String[] args) {

MinHeap heap = new MinHeap();

heap.insert(3);

heap.insert(1);

heap.insert(4);

heap.insert(1);

heap.insert(5);

heap.insert(9);

System.out.println("Minimum element: " + heap.getMin());

System.out.println("Deleted minimum element: " + heap.deleteMin());

System.out.println("Minimum element after deletion: " +

heap.getMin());

}

}

**Output:**

Minimum element: 1

Deleted minimum element: 1

Minimum element after deletion: 1

**Task 4: Graph Edge Addition Validation**

**Given a directed graph, write a function that adds an edge between two nodes and then checks if the graph still has no cycles. If a cycle is created, the edge should not be added.**

**package** graph;

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

**public** **class** GraphEdge {

**private** **final** Map<Integer, List<Integer>> adjList;

**public** GraphEdge() {

adjList = **new** HashMap<>();

}

**public** **void** addNode(**int** node) {

adjList.putIfAbsent(node, **new** ArrayList<>());

}

**public** **boolean** addEdge(**int** from, **int** to) {

addNode(from);

addNode(to);

// Check for cycles before adding the edge

**if** (createsCycle(from, to)) {

**return** **false**;

}

adjList.get(from).add(to);

**return** **true**;

}

**private** **boolean** createsCycle(**int** from, **int** to) {

Set<Integer> visited = **new** HashSet<>();

**return** hasCycle(from, to, visited);

}

**private** **boolean** hasCycle(**int** current, **int** target, Set<Integer> visited) {

**if** (current == target) {

**return** **true**;

}

visited.add(current);

**for** (**int** neighbor : adjList.getOrDefault(current,

Collections.*emptyList*())) {

**if** (!visited.contains(neighbor) && hasCycle(neighbor, target,

visited)) {

**return** **true**;

}

}

visited.remove(current);

**return** **false**;

}

**public** **void** printGraph() {

**for** (**int** node : adjList.keySet()) {

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

**for** (**int** neighbor : adjList.get(node)) {

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

}

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

}

}

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

GraphEdge graph = **new** GraphEdge();

graph.addNode(1);

graph.addNode(2);

graph.addNode(3);

graph.addNode(4);

System.***out***.println("Adding edge 1 -> 2: " + graph.addEdge(1, 2));

System.***out***.println("Adding edge 2 -> 3: " + graph.addEdge(2, 3));

System.***out***.println("Adding edge 3 -> 4: " + graph.addEdge(3, 4));

System.***out***.println("Adding edge 4 -> 1 (creates cycle): " +

graph.addEdge(4, 1));

System.***out***.println("Graph:");

graph.printGraph();

}

}

**Output:**

Adding edge 1 -> 2: true

Adding edge 2 -> 3: true

Adding edge 3 -> 4: true

Adding edge 4 -> 1 (creates cycle): true

Graph:

1 -> 2

2 -> 3

3 -> 4

4 -> 1

**Task 5: Breadth-First Search (BFS) Implementation**

**For a given undirected graph, implement BFS to traverse the graph starting from a given node and print each node in the order it is visited.**

**package** graph;

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

**public** **class** BFSUndirectedGraph {

**private** **final** Map<Integer, List<Integer>> adjList;

**public** BFSUndirectedGraph() {

adjList = **new** HashMap<>();

}

**public** **void** addEdge(**int** from, **int** to) {

adjList.putIfAbsent(from, **new** ArrayList<>());

adjList.putIfAbsent(to, **new** ArrayList<>());

adjList.get(from).add(to);

adjList.get(to).add(from);

}

**public** **void** bfs(**int** start) {

Set<Integer> visited = **new** HashSet<>();

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

queue.add(start);

visited.add(start);

**while** (!queue.isEmpty()) {

**int** node = queue.poll();

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

**for** (**int** neighbor : adjList.getOrDefault(node,

Collections.*emptyList*())) {

**if** (!visited.contains(neighbor)) {

queue.add(neighbor);

visited.add(neighbor);

}

}

}

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

}

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

BFSUndirectedGraph graph = **new** BFSUndirectedGraph();

graph.addEdge(1, 2);

graph.addEdge(1, 3);

graph.addEdge(2, 4);

graph.addEdge(3, 4);

graph.addEdge(4, 5);

System.***out***.println("BFS traversal starting from node 1:");

graph.bfs(1);

}

}

**Output:**

BFS traversal starting from node 1:

1 2 3 4 5

**Task 6: Depth-First Search (DFS) Recursive**

**Write a recursive DFS function for a given undirected graph. The function should visit every node and print it out**.

**package** graph;

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

**public** **class** DFSUndirectedGraph {

**private** **final** Map<Integer, List<Integer>> adjList;

**public** DFSUndirectedGraph() {

adjList = **new** HashMap<>();

}

**public** **void** addEdge(**int** from, **int** to) {

adjList.putIfAbsent(from, **new** ArrayList<>());

adjList.putIfAbsent(to, **new** ArrayList<>());

adjList.get(from).add(to);

adjList.get(to).add(from);

}

**public** **void** dfs(**int** start) {

Set<Integer> visited = **new** HashSet<>();

dfsHelper(start, visited);

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

}

**private** **void** dfsHelper(**int** node, Set<Integer> visited) {

**if** (visited.contains(node)) {

**return**;

}

visited.add(node);

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

**for** (**int** neighbor : adjList.getOrDefault(node,

Collections.*emptyList*())) {

dfsHelper(neighbor, visited);

}

}

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

DFSUndirectedGraph graph = **new** DFSUndirectedGraph();

graph.addEdge(1, 2);

graph.addEdge(1, 3);

graph.addEdge(2, 4);

graph.addEdge(3, 4);

graph.addEdge(4, 5);

System.***out***.println("DFS traversal starting from node 1:");

graph.dfs(1);

}

}

**Output:**

DFS traversal starting from node 1:

1 2 4 3 5