**Digvijay Thakare(**[**digvijaythakare2017@gmail.com**](mailto:digvijaythakare2017@gmail.com)**)**

**Day 7\_8\_Core Java Assignment**

**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.**

**Code-**

package com.epwipro;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

this.left = null;

this.right = null;

}

}

public class BalancedBinaryTree {

// Helper function to check the height and balance of the tree

private static int checkHeight(TreeNode root) {

if (root == null) {

return 0;

}

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

if (leftHeight == -1) {

return -1; // Left subtree is not balanced

}

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

if (rightHeight == -1) {

return -1; // Right subtree is not balanced

}

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

return -1; // Current node is not balanced

}

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

}

// Function to check if the tree is balanced

public static boolean isBalanced(TreeNode root) {

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

}

public static void main(String[] args) {

// Example usage:

// Creating a balanced binary tree

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);

root.right.right = new TreeNode(6);

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

// Creating an unbalanced binary tree

TreeNode root2 = new TreeNode(1);

root2.left = new TreeNode(2);

root2.left.left = new TreeNode(3);

System.***out***.println("Is the tree balanced? " + *isBalanced*(root2));

}

}

**Output-**

Is the tree balanced? true

Is the tree balanced? false

**Task 2: Trie for Prefix Checking Implement a trie 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 trie**

**Code-**package com.epwipro;

import java.util.HashMap;

import java.util.Map;

class TrieNode {

Map<Character, TrieNode> children;

boolean isEndOfWord;

public TrieNode() {

children = new HashMap<>();

isEndOfWord = false;

}

}

public class Trie {

private TrieNode root;

public Trie() {

root = new TrieNode();

}

// Method to insert a word into the Trie

public void insert(String word) {

TrieNode node = root;

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

node.children.putIfAbsent(ch, new TrieNode());

node = node.children.get(ch);

}

node.isEndOfWord = true;

}

// Method to check if there is any word in the Trie that starts with the given prefix

public boolean startsWith(String prefix) {

TrieNode node = root;

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

node = node.children.get(ch);

if (node == null) {

return false;

}

}

return true;

}

public static void main(String[] args) {

Trie trie = new Trie();

// Insert words into the Trie

trie.insert("hello");

trie.insert("helium");

trie.insert("help");

trie.insert("hero");

trie.insert("hermit");

// Check for prefixes

System.***out***.println(trie.startsWith("hel"));

System.***out***.println(trie.startsWith("her"));

System.***out***.println(trie.startsWith("he"));

System.***out***.println(trie.startsWith("hero"));

System.***out***.println(trie.startsWith("hex"));

}

}

**Output-**

true

true

true

true

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."**

**Code-**

package com.epwipro;

import java.util.ArrayList;

public class MinHeap {

private ArrayList<Integer> heap;

public MinHeap() {

heap = new ArrayList<>();

}

// Get the index of the parent of the node at index i

private int parent(int i) {

return (i - 1) / 2;

}

// Get the index of the left child of the node at index i

private int left(int i) {

return 2 \* i + 1;

}

// Get the index of the right child of the node at index i

private int right(int i) {

return 2 \* i + 2;

}

// Swap the elements at indices i and j

private void swap(int i, int j) {

int temp = heap.get(i);

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

heap.set(j, temp);

}

// Insert a new element into the heap

public void insert(int element) {

heap.add(element);

int i = heap.size() - 1;

// Bubble up to maintain heap property

while (i != 0 && heap.get(parent(i)) > heap.get(i)) {

swap(i, parent(i));

i = parent(i);

}

}

// Get the minimum element (root of the heap)

public int getMin() {

if (heap.size() == 0) {

throw new IllegalStateException("Heap is empty");

}

return heap.get(0);

}

// Remove and return the minimum element (root of the heap)

public int extractMin() {

if (heap.size() == 0) {

throw new IllegalStateException("Heap is empty");

}

if (heap.size() == 1) {

return heap.remove(0);

}

int root = heap.get(0);

heap.set(0, heap.remove(heap.size() - 1));

// Bubble down to maintain heap property

minHeapify(0);

return root;

}

// Maintain the min-heap property by bubbling down the element at index i

private void minHeapify(int i) {

int left = left(i);

int right = right(i);

int smallest = i;

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 != i) {

swap(i, smallest);

minHeapify(smallest);

}

}

public static void main(String[] args) {

MinHeap minHeap = new MinHeap();

minHeap.insert(3);

minHeap.insert(2);

minHeap.insert(15);

minHeap.insert(5);

minHeap.insert(4);

minHeap.insert(45);

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

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

System.***out***.println("New minimum element: " + minHeap.getMin());

minHeap.insert(1);

System.***out***.println("New minimum element after inserting 1: " + minHeap.getMin()); // 1

}

}

**Output-**

Minimum element: 2

Extracted minimum: 2

New minimum element: 3

New minimum element after inserting 1: 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.**

**Code-**

package com.epwipro;

import java.util.\*;

class Graph {

private Map<Integer, List<Integer>> adjacencyList;

public Graph() {

adjacencyList = new HashMap<>();

}

// Method to add a node to the graph

public void addNode(int node) {

adjacencyList.putIfAbsent(node, new ArrayList<>());

}

// Method to add an edge to the graph and check for cycles

public boolean addEdge(int from, int to) {

addNode(from);

addNode(to);

// Temporarily add the edge

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

// Check if this addition creates a cycle

if (hasCycle()) {

// Remove the edge if it creates a cycle

adjacencyList.get(from).remove((Integer) to);

System.***out***.println("Adding edge from " + from + " to " + to + " creates a cycle. Edge not added.");

return false;

}

System.***out***.println("Adding edge from " + from + " to " + to + " does not create a cycle. Edge added.");

return true;

}

// Method to check if the graph has a cycle using DFS

private boolean hasCycle() {

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

Set<Integer> recursionStack = new HashSet<>();

for (Integer node : adjacencyList.keySet()) {

if (dfs(node, visited, recursionStack)) {

return true;

}

}

return false;

}

// Helper method for DFS to detect cycles

private boolean dfs(int node, Set<Integer> visited, Set<Integer> recursionStack) {

if (recursionStack.contains(node)) {

return true;

}

if (visited.contains(node)) {

return false;

}

visited.add(node);

recursionStack.add(node);

List<Integer> neighbors = adjacencyList.get(node);

if (neighbors != null) {

for (Integer neighbor : neighbors) {

if (dfs(neighbor, visited, recursionStack)) {

return true;

}

}

}

recursionStack.remove(node);

return false;

}

// Method to print the graph (for debugging purposes)

public void printGraph() {

for (Map.Entry<Integer, List<Integer>> entry : adjacencyList.entrySet()) {

System.***out***.println("Node " + entry.getKey() + " has edges to: " + entry.getValue());

}

}

public static void main(String[] args) {

Graph graph = new Graph();

// Example usage

graph.addEdge(1, 2); // true

graph.addEdge(2, 3); // true

graph.addEdge(3, 4); // true

graph.addEdge(4, 2); // false, creates a cycle

graph.printGraph();

}

}

**Output-**

Adding edge from 1 to 2 does not create a cycle. Edge added.

Adding edge from 2 to 3 does not create a cycle. Edge added.

Adding edge from 3 to 4 does not create a cycle. Edge added.

Adding edge from 4 to 2 creates a cycle. Edge not added.

Node 1 has edges to: [2]

Node 2 has edges to: [3]

Node 3 has edges to: [4]

Node 4 has edges to: []

**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.**

**Code-**

package com.epwipro;

import java.util.\*;

class Graph1 {

private Map<Integer, List<Integer>> adjacencyList;

public Graph1() {

adjacencyList = new HashMap<>();

}

// Method to add an edge to the graph (since the graph is undirected, add both ways)

public void addEdge(int from, int to) {

adjacencyList.putIfAbsent(from, new ArrayList<>());

adjacencyList.putIfAbsent(to, new ArrayList<>());

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

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

}

// Method to perform BFS starting from a given node

public void bfs(int start) {

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

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

// Start the BFS with the start node

visited.add(start);

queue.add(start);

System.***out***.println("BFS Traversal starting from node " + start + ":");

while (!queue.isEmpty()) {

int node = queue.poll();

System.***out***.println("Visited node: " + node);

// Visit all the neighbors of the current node

List<Integer> neighbors = adjacencyList.get(node);

if (neighbors != null) {

for (int neighbor : neighbors) {

if (!visited.contains(neighbor)) {

visited.add(neighbor);

queue.add(neighbor);

}

}

}

}

}

public static void main(String[] args) {

Graph1 graph = new Graph1();

// Adding edges to the graph

graph.addEdge(1, 2);

graph.addEdge(1, 3);

graph.addEdge(2, 4);

graph.addEdge(2, 5);

graph.addEdge(3, 6);

graph.addEdge(3, 7);

// Perform BFS starting from node 1

graph.bfs(1);

}

}

**Output-**

BFS Traversal starting from node 1:

Visited node: 1

Visited node: 2

Visited node: 3

Visited node: 4

Visited node: 5

Visited node: 6

Visited node: 7

**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.**

**Code-**

package com.epwipro;

import java.util.\*;

class Graph2 {

private Map<Integer, List<Integer>> adjacencyList;

public Graph2() {

adjacencyList = new HashMap<>();

}

// Method to add an edge to the graph (since the graph is undirected, add both ways)

public void addEdge(int from, int to) {

adjacencyList.putIfAbsent(from, new ArrayList<>());

adjacencyList.putIfAbsent(to, new ArrayList<>());

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

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

}

// Method to perform DFS recursively starting from a given node

public void dfsRecursive(int node, Set<Integer> visited) {

visited.add(node);

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

List<Integer> neighbors = adjacencyList.get(node);

if (neighbors != null) {

for (int neighbor : neighbors) {

if (!visited.contains(neighbor)) {

dfsRecursive(neighbor, visited);

}

}

}

}

// Method to start DFS traversal from a given node

public void startDFS(int start) {

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

System.***out***.println("DFS Traversal starting from node " + start + ":");

dfsRecursive(start, visited);

System.***out***.println(); // for a new line after traversal

}

public static void main(String[] args) {

Graph2 graph = new Graph2();

// Adding edges to the graph

graph.addEdge(1, 2);

graph.addEdge(1, 3);

graph.addEdge(2, 4);

graph.addEdge(2, 5);

graph.addEdge(3, 6);

graph.addEdge(3, 7);

// Perform DFS starting from node 2

graph.startDFS(2);

}

}

**Output-**

DFS Traversal starting from node 2:

2 1 3 6 7 4 5