# Day 7&8

**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.

**Program:**

public class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(){}

TreeNode(int val){

this.val = val;

}

TreeNode(int val, TreeNode left, TreeNode right){

this.val = val;

this.left = left;

this.right = right;

}

}

package Assignments.Day7and8;

public class Task1 {

private static boolean isBalanced(TreeNode root){

if(root == null) return true;

return height(root) >= 0;

}

private static int height(TreeNode root) {

if(root == null)return 0;

int left = height(root.left);

int right = height(root.right);

if(left == -1 || right == -1) return -1;

if(Math.abs(left -right) > 1) return -1;

return 1+Math.max(left,right);

}

public static void main(String[] args) {

TreeNode node = new TreeNode(1);

node.left = new TreeNode(2);

node.right = new TreeNode(2);

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

node.left.right = new TreeNode(3);

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

node.left.left.right = new TreeNode(4);

TreeNode node1 =new TreeNode(3);

node1.left = new TreeNode(9);

node1.right = new TreeNode(20);

node1.right.left = new TreeNode(7);

node1.right.right = new TreeNode(15);

if(isBalanced(node1)){

System.out.println("The given Tree Node1 is Balanced");

}else

System.out.println("The given Tree Node1 is not Balanced");

if(isBalanced(node)){

System.out.println("The given Tree Node is Balanced");

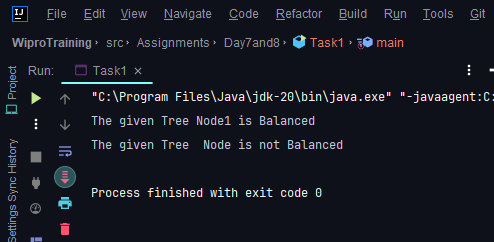
}else

System.out.println("The given Tree Node is not Balanced");

}

}

**Output:**



**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.

**Program:**

package Assignments.Day7and8;

import java.util.HashMap;

import java.util.Map;

public class Task2 {

private TrieNode root;

public Task2() {

root = new TrieNode();

}

private static class TrieNode {

private Map<Character, TrieNode> children;

private boolean isWord;

public TrieNode() {

children = new HashMap<>();

isWord = false;

}

}

public void insert(String word) {

TrieNode cur = root;

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

char ch = word.charAt(i);

TrieNode node = cur.children.get(ch);

if (node == null) {

node = new TrieNode();

cur.children.put(ch, node);

}

cur = node;

}

cur.isWord = true;

}

public boolean isPrefix(String prefix) {

TrieNode current = root;

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

char ch = prefix.charAt(i);

TrieNode node = current.children.get(ch);

if (node == null) {

return false;

}

current = node;

}

return true;

}

public static void main(String[] args) {

Task2 trie = new Task2();

trie.insert("apple");

trie.insert("banana");

trie.insert("cherry");

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

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

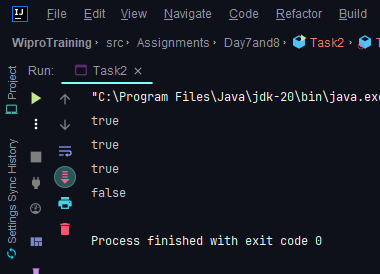
System.out.println(trie.isPrefix("che")); // true

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

}

}

**Output:**



**Task 3: Implementing Heap Operations**

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

**Program:**

package Assignments.Day7and8;

public class Task3 {

private final int[] heap;

private int size;

private final int maxSize;

private static final int FRONT = 1;

public Task3(int maxSize) {

this.maxSize = maxSize;

this.size = 0;

heap = new int[this.maxSize + 1];

heap[0] = Integer.MIN\_VALUE;

}

public static void main(String[] args) {

System.out.println("The min heap is: ");

Task3 minHeap = new Task3(15);

minHeap.insert(5);

minHeap.insert(3);

minHeap.insert(17);

minHeap.insert(10);

minHeap.insert(84);

minHeap.insert(19);

minHeap.insert(6);

minHeap.insert(22);

minHeap.insert(9);

minHeap.display();

System.out.println("The Min val is " + minHeap.remove());

}

private int remove() {

int pop = heap[FRONT];

heap[FRONT] = heap[size--];

minElement(FRONT);

return pop;

}

private void minElement(int position) {

if (!isLeaf()){

int swapPosition;

if (rightChild(position)<= size){

swapPosition = heap[leftChild(position)]<heap[rightChild(position)]?leftChild(position): rightChild(position);

}else {

swapPosition = leftChild(position);

}

if (heap[position] > heap[leftChild(position)] || heap[position]> heap[rightChild(position)]){

swap(position,swapPosition);

minElement(swapPosition);

}

}

}

private int leftChild(int position) {

return 2\*position;

}

private int rightChild(int position) {

return 2\*position + 1;

}

private boolean isLeaf() {

return FRONT > size / 2;

}

private void display() {

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

System.out.print("PARENT: " + heap[i] + " LEFT CHILD: " + heap[2\*i] + " RIGHT CHILD: "+ heap[2\*i+1]);

System.out.println();

}

}

private void insert(int n) {

if(size >= maxSize){

return;

}

heap[++size] = n;

int cur = size;

while(heap[cur] < heap[parent(cur)]){

swap(cur, parent(cur));

cur = parent(cur);

}

}

private void swap(int a, int b){

int temp = heap[a];

heap[a] = heap[b];

heap[b] = temp;

}

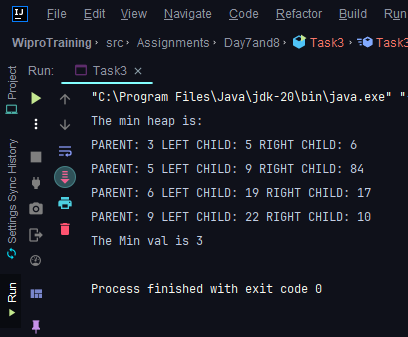
private int parent(int position) {

return position/2;

}

}

**Output:**



**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.

**Program:**

package Assignments.Day7and8;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.List;

public class Task4 {

private final int V;

private final List<List<Integer>> adj;

public Task4(int V)

{

this.V = V;

adj = new ArrayList<>(V);

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

adj.add(new LinkedList<>());

}

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

boolean[] recStack)

{

if (recStack[i])

return true;

if (visited[i])

return false;

visited[i] = true;

recStack[i] = true;

List<Integer> children = adj.get(i);

for (Integer c : children)

if (isCyclicUtil(c, visited, recStack))

return true;

recStack[i] = false;

return false;

}

private void addEdge(int source, int dest)

{

adj.get(source).add(dest);

}

private 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 static void main(String[] args) {

Task4 graph = new Task4(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);

// Function call

if (graph.isCyclic())

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

else

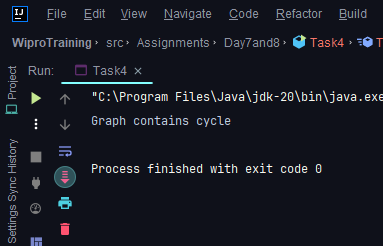
System.out.println("Graph doesn't "

+ "contain cycle");

}

}

**Output:**



**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.

**Program:**

package Assignments.Day7and8;

import java.util.\*;

class Task5 {

private int V;

private LinkedList<Integer>[] adj;

public Task5(int v) {

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>();

}

}

public void addEdge(int v, int w) {

adj[v].add(w);

adj[w].add(v);

}

public void BFS(int start) {

boolean[] visited = new boolean[V];

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

visited[start] = true;

queue.add(start);

while (!queue.isEmpty()) {

int current = queue.poll();

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

for (int next : adj[current]) {

if (!visited[next]) {

visited[next] = true;

queue.add(next);

}

}

}

}

public static void main(String[] args) {

Task5 graph = new Task5(6);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 3);

graph.addEdge(2, 4);

graph.addEdge(3, 5);

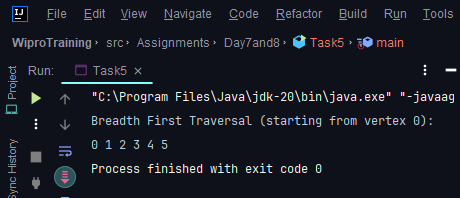
System.out.println("Breadth First Traversal (starting from vertex 0):");

graph.BFS(0);

}

}

**Output:**



**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.

**Program:**

package Assignments.Day7and8;

import java.util.\*;

class Task6 {

private int V;

private final LinkedList<Integer>[] adj; // Adjacency list

public Task6(int v) {

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<>();

}

}

public void addEdge(int v, int w) {

adj[v].add(w);

adj[w].add(v);

}

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

visited[v] = true;

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

for (int next : adj[v]) {

if (!visited[next]) {

DFSUtil(next, visited);

}

}

}

public void DFS(int v) {

boolean[] visited = new boolean[V];

DFSUtil(v, visited);

}

public static void main(String[] args) {

Task6 graph = new Task6(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);

System.out.println("Depth First Traversal (starting from vertex 2):");

graph.DFS(2);

}

}

**Output:**

