Assignment-1

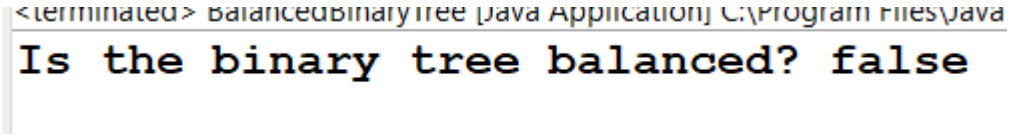
Balanced Binary Tree Check

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 com.wipro.day7and8;

public class BalancedBinaryTree {
    class Node {
        int val;
        Node left;
        Node right;

        Node(int val) {
            this.val = val;
        }
    }

    public boolean isBalanced(Node root) {
        if (root == null)
            return true;

        int leftHeight = getHeight(root.left);
        int rightHeight = getHeight(root.right);

        return Math.abs(leftHeight - rightHeight) <=
1 && isBalanced(root.left) && isBalanced(root.right);
    }

    private int getHeight(Node node) {
        if (node == null)
            return 0;

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

    public static void main(String[] args) {
        BalancedBinaryTree binaryTree = new
BalancedBinaryTree();
```

```
        Node root = binaryTree.new Node(1);
        root.left = binaryTree.new Node(2);
        root.right = binaryTree.new Node(3);
        root.left.left = binaryTree.new Node(4);
        root.left.right = binaryTree.new Node(5);
        root.left.left.left = binaryTree.new Node(6);

        System.out.println("Is the binary tree
balanced? " + binaryTree.isBalanced(root));
    }
}
```

OUTPUT:



Assignment-2

```
package com.wipro.day7and8;

import java.util.HashMap;
import java.util.Map;

public class Trie {
    class Node {
        Map<Character, Node> children;
        boolean isEndOfWord;

        Node() {
            children = new HashMap<>();
            isEndOfWord = false;
        }
    }

    private Node root;
```

```
    public Trie() {
        root = new Node();
    }

    public void insert(String word) {
        Node node = root;
        for (char c : word.toCharArray()) {
            node.children.putIfAbsent(c, new Node());
            node = node.children.get(c);
        }
        node.isEndOfWord = true;
    }

    public boolean startsWith(String prefix) {
        Node node = root;
        for (char c : prefix.toCharArray()) {
            if (!node.children.containsKey(c))
                return false;
            node = node.children.get(c);
        }
        return true;
    }

    public static void main(String[] args) {
        Trie trie = new Trie();
        trie.insert("apple");
        trie.insert("application");
        trie.insert("app");

        System.out.println("Is 'app' a prefix? " +
trie.startsWith("app"));
        System.out.println("Is 'apps' a prefix? " +
trie.startsWith("apps"));
    }
}
```

OUTPUT:

```
<terminated> Trie [Java Application] C:\Program Files\Java\j
Is 'app' a prefix? true
Is 'apps' a prefix? false
```

Assignment-3

Trie for Prefix Checking

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

Solution:

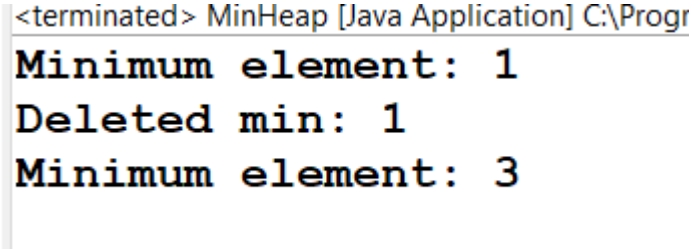
```
package com.wipro.day7and8;


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;

        while (index > 0 && heap.get((index - 1) / 2)
> heap.get(index)) {

            int parentIndex = (index - 1) / 2;

            int temp = heap.get(parentIndex);

            heap.set(parentIndex, heap.get(index));

            heap.set(index, temp);

            index = parentIndex;

        }

    }


    public int deleteMin() {

        if (heap.isEmpty())

            throw new IllegalStateException("Heap is
empty");


        int min = heap.get(0);

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

        heap.remove(heap.size() - 1);


        int index = 0;

        while (true) {

            int leftChildIndex = 2 * index + 1;

            int rightChildIndex = 2 * index + 2;
```

```
            int smallest = index;


            if (leftChildIndex < heap.size() &&
heap.get(leftChildIndex) < heap.get(smallest))

                smallest = leftChildIndex;

            if (rightChildIndex < heap.size() &&
heap.get(rightChildIndex) < heap.get(smallest))

                smallest = rightChildIndex;


            if (smallest == index)

                break;


            int temp = heap.get(index);

            heap.set(index, heap.get(smallest));

            heap.set(smallest, temp);

            index = smallest;

        }


        return min;

    }


    public int getMin() {

        if (heap.isEmpty())

            throw new IllegalStateException("Heap is
empty");
```

```
        return heap.get(0);
    }

    public static void main(String[] args) {
        MinHeap minHeap = new MinHeap();
        minHeap.insert(5);
        minHeap.insert(3);
        minHeap.insert(7);
        minHeap.insert(1);

        System.out.println("Minimum element: " +
minHeap.getMin());
        System.out.println("Deleted min: " +
minHeap.deleteMin());
        System.out.println("Minimum element: " +
minHeap.getMin());
    }
}
```

OUTPUT:



Assignment-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.

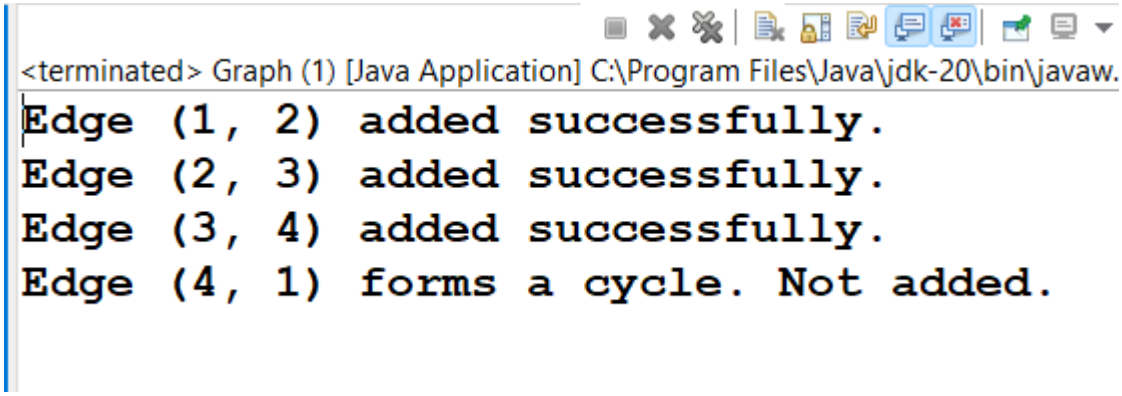
Solution:

```
package com.wipro.day7and8;


import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;


public class Graph {


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


    public Graph() {

        adjacencyList = new HashMap<>();

    }


    public void addEdge(int from, int to) {

        if (!adjacencyList.containsKey(from)) {
```

```
            adjacencyList.put(from, new
ArrayList<>());

        }

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



        // Check for cycles

        if (hasCycle(from)) {

            System.out.println("Edge (" + from + ", "
+ to + ") forms a cycle. Not added.");


adjacencyList.get(from).remove(adjacencyList.get(from
).size() - 1);

        } else {

            System.out.println("Edge (" + from + ", "
+ to + ") added successfully.");

        }

    }



    private boolean hasCycle(int node) {

        return hasCycleUtil(node, new ArrayList<>());

    }



    private boolean hasCycleUtil(int node,
List<Integer> visited) {

        if (visited.contains(node)) {

            return true;
```

```
        }


        visited.add(node);
        if (adjacencyList.containsKey(node)) {
            for (int neighbor :
adjacencyList.get(node)) {
                if (hasCycleUtil(neighbor, visited))
{
                    return true;
                }
            }
        }
        visited.remove(visited.size() - 1);
        return false;
    }


    public static void main(String[] args) {
        Graph graph = new Graph();
        graph.addEdge(1, 2);
        graph.addEdge(2, 3);
        graph.addEdge(3, 4);
        graph.addEdge(4, 1);
    }
}
```

OUTPUT:



Assignment-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.

Solution:

```
package com.wipro.day7and8;

import java.util.*;

public class Graph1 {
    private Map<Integer, List<Integer>>
adjacencyList;

    public Graph1() {
        adjacencyList = new HashMap<>();
    }

    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); // For
undirected graph
    }
```

```
    public void bfs(int startNode) {
        Queue<Integer> queue = new LinkedList<>();
        Set<Integer> visited = new HashSet<>();

        queue.offer(startNode);
        visited.add(startNode);

        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print(node + " ");

            for (int neighbor :
adjacencyList.getOrDefault(node, new ArrayList<>()))
{
                if (!visited.contains(neighbor)) {
                    queue.offer(neighbor);
                    visited.add(neighbor);
                }
            }
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Graph1 graph = new Graph1();
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);
        graph.addEdge(2, 5);
        graph.addEdge(3, 6);
        graph.addEdge(3, 7);

        System.out.println("BFS traversal starting
from node 1:");
        graph.bfs(1);
    }
}
```

OUTPUT:

```
<terminated> Graph1 [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.e
BFS traversal starting from node 1:
1 2 3 4 5 6 7
```

Assignment-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.

Solution:

```
package com.wipro.day7and8;


import java.util.*;

public class Graph2 {
    private Map<Integer, List<Integer>>
adjacencyList;

    public Graph2() {
        adjacencyList = new HashMap<>();
    }

    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); // For
undirected graph
    }

    public void dfs(int startNode) {
```

```
        Set<Integer> visited = new HashSet<>();
        dfsUtil(startNode, visited);
    }

    private void dfsUtil(int node, Set<Integer>
visited) {
        visited.add(node);
        System.out.print(node + " ");

        for (int neighbor :
adjacencyList.getOrDefault(node, new ArrayList<>()))
{
            if (!visited.contains(neighbor)) {
                dfsUtil(neighbor, visited);
            }
        }
    }

    public static void main(String[] args) {
        Graph2 graph = new Graph2();
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);
        graph.addEdge(2, 5);
        graph.addEdge(3, 6);
        graph.addEdge(3, 7);

        System.out.println("DFS traversal starting
from node 1:");
        graph.dfs(1);
    }
}
```

OUTPUT:

```
DFS traversal starting from node 1:
1 2 4 5 3 6 7
```

-----END-----