Day 7 and 8:

NOTE: Task 2 is pending I need some clearance for that.

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.
Ans)
Code:-

```
package WiprpTask;
class TreeNode {
  int value;
  TreeNode left;
  TreeNode right;
  TreeNode(int value) {
     this.value = value;
     left = right = null;
  }
}
public class BalancedBinaryTree {
    static class HeightBalancedStatus {
      int height;
      boolean isBalanced;
      HeightBalancedStatus(int height, boolean isBalanced)
{
         this.height = height;
         this.isBalanced = isBalanced;
      }
  }
  private static HeightBalancedStatus
checkBalanced(TreeNode node) {
      if (node == null) {
         return new HeightBalancedStatus(0, true);
      }
```

```java
      HeightBalancedStatus leftStatus =
checkBalanced(node.left);
      HeightBalancedStatus rightStatus =
checkBalanced(node.right);
      boolean isBalanced = leftStatus.isBalanced &&
rightStatus.isBalanced
                        && Math.abs(leftStatus.height -
rightStatus.height) <= 1;
      int height = 1 + Math.max(leftStatus.height,
rightStatus.height);
      return new HeightBalancedStatus(height, isBalanced);
   }
    public static boolean isBalanced(TreeNode root) {
      return checkBalanced(root).isBalanced;
   }
   public static void main(String[] args) {
        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.left = new TreeNode(6);
      root.right.right = new TreeNode(7);
      System.out.println("Is the tree balanced? " +
isBalanced(root));
        TreeNode unbalancedRoot = new TreeNode(1);
      unbalancedRoot.left = new TreeNode(2);
      unbalancedRoot.left.left = new TreeNode(3);
      unbalancedRoot.left.left.left = new TreeNode(4);
      System.out.println("Is the tree balanced? " +
isBalanced(unbalancedRoot));
   }
}
```

<u>OUTPUT:-</u>

```
Is the tree balanced? true
Is the tree balanced? 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
Ans)
<u>Code:-</u>

```java
package WiprpTask;
import java.util.ArrayList;
public class MinHeap {
   private ArrayList<Integer> heap;
   public MinHeap() {
       this.heap = new ArrayList<>();
   }
   private void swap(int i, int j) {
       int temp = heap.get(i);
       heap.set(i, heap.get(j));
       heap.set(j, temp);
   }
   private int parent(int index) {
       return (index - 1) / 2;
   }
   private int leftChild(int index) {
       return 2 * index + 1;
   }
   private int rightChild(int index) {
       return 2 * index + 2;
   }
   public void insert(int value) {
```

```
        heap.add(value);
        int index = heap.size() - 1;
        while (index > 0 && heap.get(index) <
heap.get(parent(index))) {
            swap(index, parent(index));
            index = parent(index);
        }
    }
    public int getMin() {
        if (heap.isEmpty()) {
            throw new IllegalStateException("Heap is empty");
        }
        return heap.get(0);
    }
    public int removeMin() {
        if (heap.isEmpty()) {
            throw new IllegalStateException("Heap is empty");
        }
        int min = heap.get(0);
        int lastElement = heap.remove(heap.size() - 1);
        if (!heap.isEmpty()) {
            heap.set(0, lastElement);
            heapifyDown(0);
        }
        return min;
    }
    private void heapifyDown(int index) {
        int smallest = index;
        int left = leftChild(index);
        int right = rightChild(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);
        }
    }
    public void printHeap() {
        for (int i : heap) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        MinHeap minHeap = new MinHeap();

        minHeap.insert(10);
        minHeap.insert(5);
        minHeap.insert(3);
        minHeap.insert(2);
        minHeap.insert(8);

        System.out.println("Heap elements: ");
        minHeap.printHeap();

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

        System.out.println("Removed minimum element: " +
minHeap.removeMin());
```

```java
        System.out.println("Heap elements after removing minimum: ");
        minHeap.printHeap();
    }
}
```

OUTPUT:-

```
Heap elements:
2 3 5 10 8
Minimum element: 2
Removed minimum element: 2
Heap elements after removing minimum:
3 8 5 10
```

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.

Ans)

Code:-

```java
package WiprpTask;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
import java.util.Stack;
public class GraphWork {
    private HashMap<String, ArrayList<String>> adjList = new HashMap<>();
    public static void main(String[] args) {
        GraphWork myGraph = new GraphWork();
        myGraph.addVertex("A");
        myGraph.addVertex("B");
```

```java
        myGraph.addVertex("C");
        myGraph.printGraph();
        myGraph.addEdge("A", "B");
        myGraph.printGraph();
        myGraph.addEdge("A", "C");
        myGraph.printGraph();
        System.out.println(myGraph.addEdge("C", "A"));
        myGraph.printGraph();
        myGraph.removeVertex("C");
        myGraph.printGraph();
    }
    public boolean addEdge(String vertex1, String vertex2) {
        if (adjList.get(vertex1) != null && adjList.get(vertex2) !=
null) {
            adjList.get(vertex1).add(vertex2);
            if (hasCycle()) {
                adjList.get(vertex1).remove(vertex2);
                return false;
            }
            return true;
        }
        return false;
    }
    private boolean hasCycle() {
        Set<String> visited = new HashSet<>();
        Set<String> recursionStack = new HashSet<>();
        for (String vertex : adjList.keySet()) {
            if (dfs(vertex, visited, recursionStack)) {
                return true;
            }
        }
        return false;
    }
```

```java
  private boolean dfs(String vertex, Set<String> visited,
Set<String> recursionStack) {
      if (recursionStack.contains(vertex)) {
          return true;
      }
      if (visited.contains(vertex)) {
          return false;
      }
      visited.add(vertex);
      recursionStack.add(vertex);
      for (String neighbor : adjList.get(vertex)) {
          if (dfs(neighbor, visited, recursionStack)) {
              return true;
          }
      }
      recursionStack.remove(vertex);
      return false;
  }
  public boolean removeEdge(String vertex1, String
vertex2) {
      if (adjList.get(vertex1) != null && adjList.get(vertex2) !=
null) {
          adjList.get(vertex1).remove(vertex2);
          return true;
      }
      return false;
  }
  public boolean addVertex(String vertex) {
      if (adjList.get(vertex) == null) {
          adjList.put(vertex, new ArrayList<String>());
          return true;
      }
      return false;
```

```
  }
  private boolean removeVertex(String vertex) {
      if (adjList.get(vertex) == null) {
          return false;
      }
      for (String adjacentVertex : adjList.get(vertex)) {
          adjList.get(adjacentVertex).remove(vertex);
      }
      adjList.remove(vertex);
      return true;
  }
  public void printGraph() {
      System.out.println(adjList);
  }
}
```

OUTPUT:-

```
{A=[], B=[], C=[]}
{A=[B], B=[], C=[]}
{A=[B, C], B=[], C=[]}
false
{A=[B, C], B=[], C=[]}
{A=[B, C], B=[]}
```

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.
Ans)
Code:-

```
package WiprpTask;
import java.util.*;
public class UndirectedGraph {
  private Map<String, List<String>> adjList;
```

```java
public UndirectedGraph() {
    adjList = new HashMap<>();
}
public void addVertex(String vertex) {
    adjList.putIfAbsent(vertex, new ArrayList<>());
}
public void addEdge(String vertex1, String vertex2) {
    adjList.get(vertex1).add(vertex2);
    adjList.get(vertex2).add(vertex1);
}
public void bfs(String startVertex) {
    Set<String> visited = new HashSet<>();
    Queue<String> queue = new LinkedList<>();
    queue.add(startVertex);
    visited.add(startVertex);
    while (!queue.isEmpty()) {
        String vertex = queue.poll();
        System.out.print(vertex + " ");
        for (String neighbor : adjList.get(vertex)) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                queue.add(neighbor);
            }
        }
    }
}
public static void main(String[] args) {
    UndirectedGraph graph = new UndirectedGraph();
    graph.addVertex("Assam");
    graph.addVertex("Bihar");
    graph.addVertex("Calcutta");
    graph.addVertex("Delhi");
    graph.addVertex("Uttarpradesh");
```

```
      graph.addEdge("Assam", "Bihar");
      graph.addEdge("Assam", "Calcutta");
      graph.addEdge("Bihar", "Delhi");
      graph.addEdge("Calcutta", "Uttarpradesh");
      System.out.println("BFS starting from vertex
Assam:");
      graph.bfs("Assam");
   }
}
```

OUTPUT:-

```
BFS starting from vertex Assam:
Assam Bihar Calcutta Delhi Uttarpradesh
```

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.
Ans)
Code:-

```
package WiprpTask;
import java.util.*;
public class DepthFirstSearch {
   private static class Graph {
      private Map<Integer, List<Integer>> adjList;
      public Graph() {
         adjList = new HashMap<>();
      }
      public void addEdge(int src, int dest) {
         adjList.computeIfAbsent(src, k -> new
ArrayList<>()).add(dest);
         adjList.computeIfAbsent(dest, k -> new
ArrayList<>()).add(src);
      }
      public void dfs(int start) {
```

```
        Set<Integer> visited = new HashSet<>();
        dfsRecursive(start, visited);
    }
    private void dfsRecursive(int vertex, Set<Integer>
visited) {
        visited.add(vertex);
        System.out.print(vertex + " ");
        List<Integer> neighbors = adjList.get(vertex);
        if (neighbors != null) {
            for (int neighbor : neighbors) {
                if (!visited.contains(neighbor)) {
                    dfsRecursive(neighbor, visited);
                }
            }
        }
    }
  }
  public static void main(String[] args) {
    Graph graph = new Graph();
    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 Search (DFS)
Recursive:");
    System.out.print("Starting from vertex 0: ");
    graph.dfs(0);
    System.out.println();
    System.out.print("Starting from vertex 2: ");
    graph.dfs(2);
    System.out.println();
```

```
    }
}
```

OUTPUT:-

```
Depth-First Search (DFS) Recursive:
Starting from vertex 0: 0 1 2 3
Starting from vertex 2: 2 0 1 3
```