RPS DAY 7- 8 Assignments

Assignment 2

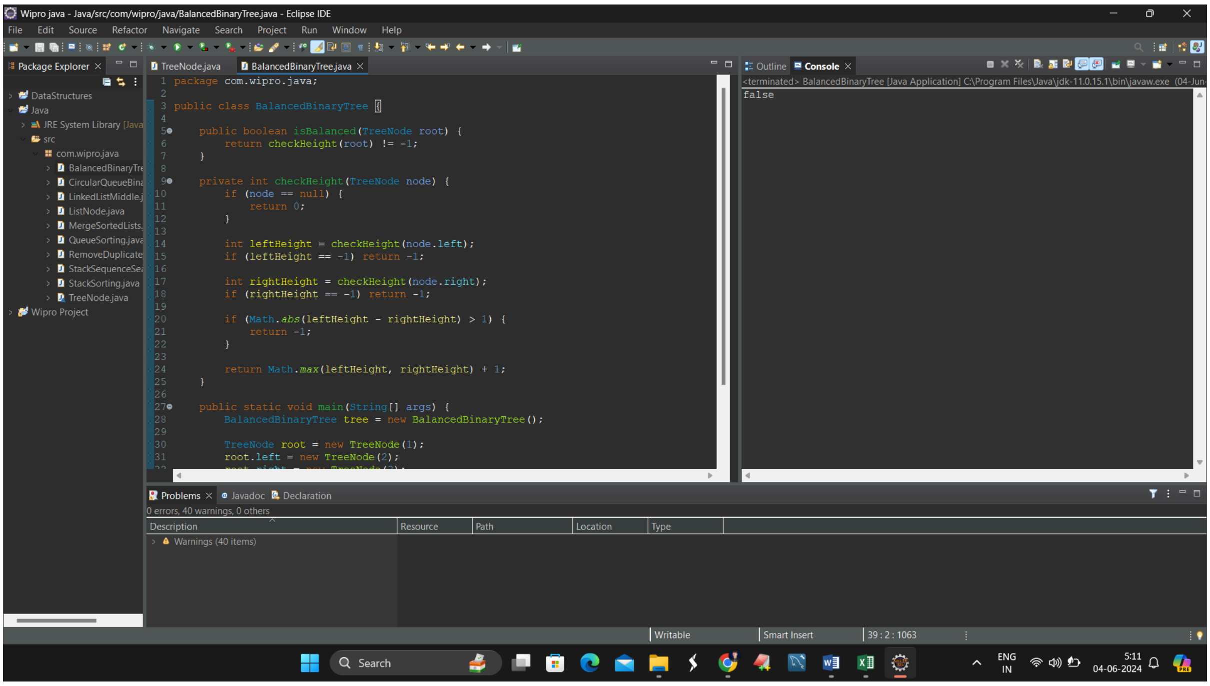
Name: Akshada Baad
Batch - CPPE

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.

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

public class BalancedBinaryTree {

    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) {
        BalancedBinaryTree tree = new BalancedBinaryTree();

        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.left.left.left = new TreeNode(8);

        System.out.println(tree.isBalanced(root));  // Output: false
    }
}
```

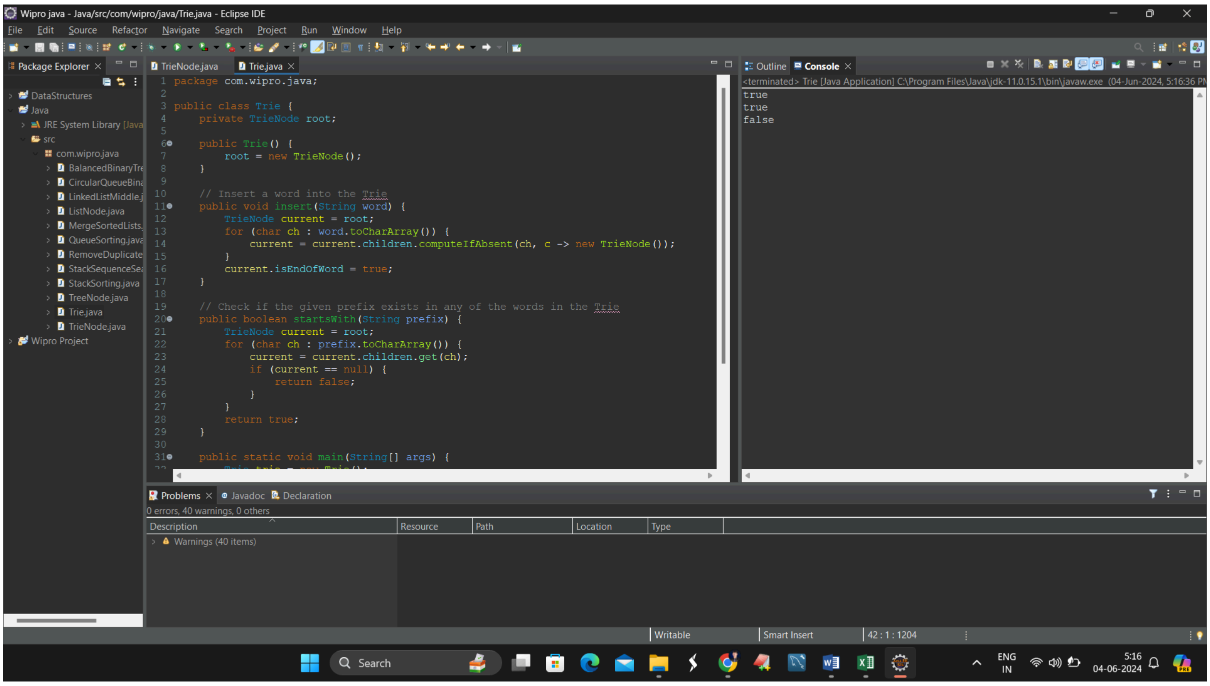


Task 2: Trie for Prefix Checking

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.

```
public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Insert a word into the Trie
    public void insert(String word) {
        TrieNode current = root;
        for (char ch : word.toCharArray()) {
            current = current.children.computeIfAbsent(ch, c -> new TrieNode());
        }
        current.isEndOfWord = true;
```

```
    }

    // Check if the given prefix exists in any of the words in the Trie
    public boolean startsWith(String prefix) {
        TrieNode current = root;
        for (char ch : prefix.toCharArray()) {
            current = current.children.get(ch);
            if (current == null) {
                return false;
            }
        }
        return true;
    }

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

        System.out.println(trie.startsWith("app"));  // Output: true
        System.out.println(trie.startsWith("ban"));  // Output: true
        System.out.println(trie.startsWith("bat"));  // Output: false
    }
}
```



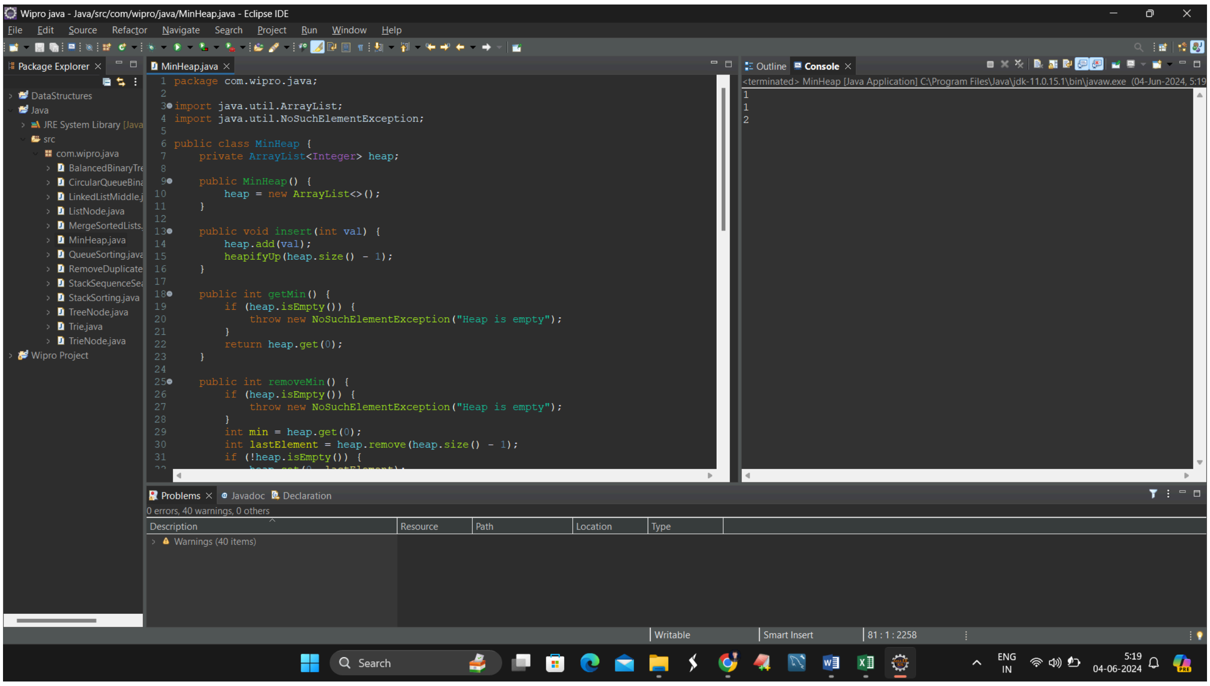
Task 3: Implementing Heap Operations

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.

```
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class MinHeap {
    private ArrayList<Integer> heap;

    public MinHeap() {
        heap = new ArrayList<>();
    }

    public void insert(int val) {
        heap.add(val);
        heapifyUp(heap.size() - 1);
    }
```

```
public int getMin() {
    if (heap.isEmpty()) {
        throw new NoSuchElementException("Heap is empty");
    }
    return heap.get(0);
}

public int removeMin() {
    if (heap.isEmpty()) {
        throw new NoSuchElementException("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 heapifyUp(int index) {
    int parentIndex = (index - 1) / 2;
    while (index > 0 && heap.get(index) < heap.get(parentIndex)) {
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = (index - 1) / 2;
    }
}

private void heapifyDown(int index) {
    int leftChild = 2 * index + 1;
```

```java
        int rightChild = 2 * index + 2;
        int smallest = index;

        if (leftChild < heap.size() && heap.get(leftChild) < heap.get(smallest)) {
            smallest = leftChild;
        }
        if (rightChild < heap.size() && heap.get(rightChild) < heap.get(smallest)) {
            smallest = rightChild;
        }
        if (smallest != index) {
            swap(index, smallest);
            heapifyDown(smallest);
        }
    }

    private void swap(int index1, int index2) {
        int temp = heap.get(index1);
        heap.set(index1, heap.get(index2));
        heap.set(index2, temp);
    }

    public static void main(String[] args) {
        MinHeap minHeap = new MinHeap();
        minHeap.insert(3);
        minHeap.insert(2);
        minHeap.insert(1);

        System.out.println(minHeap.getMin());  // Output: 1
        System.out.println(minHeap.removeMin());  // Output: 1
        System.out.println(minHeap.getMin());  // Output: 2
    }
}
```



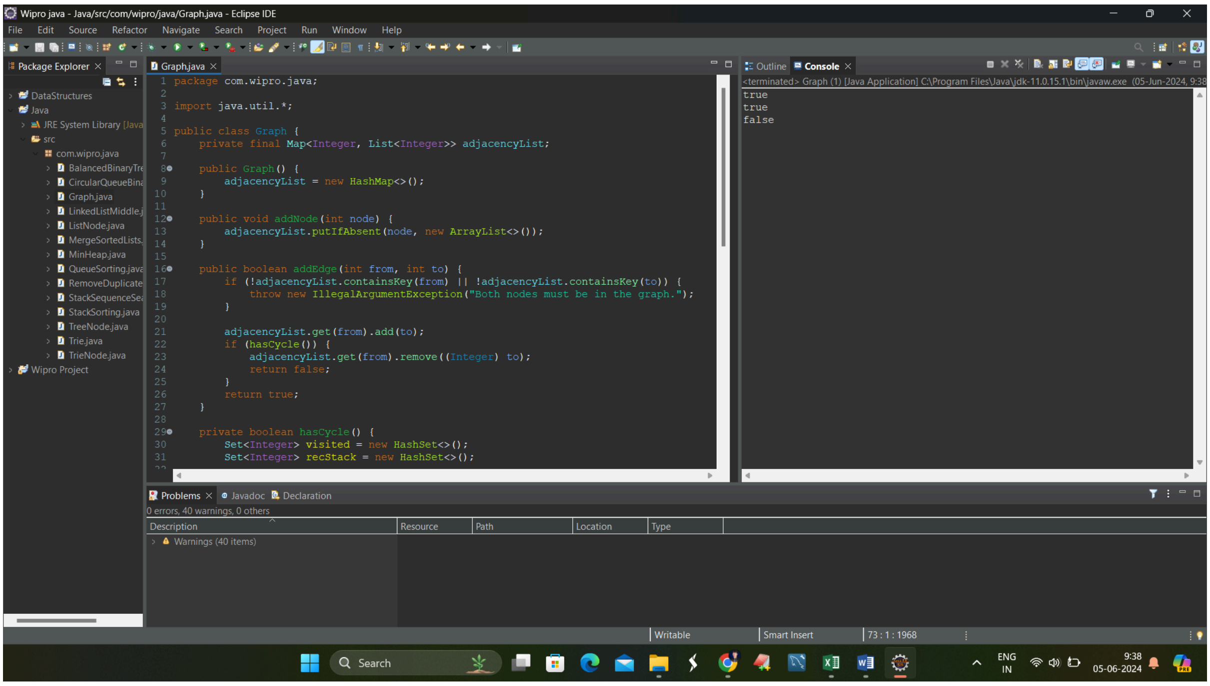
Task 4: Graph Edge Addition Validation

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.

```
import java.util.*;


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


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


    public void addNode(int node) {
        adjacencyList.putIfAbsent(node, new ArrayList<>());
    }
```

```java
public boolean addEdge(int from, int to) {
    if (!adjacencyList.containsKey(from) || !adjacencyList.containsKey(to)) {
        throw new IllegalArgumentException("Both nodes must be in the graph.");
    }

    adjacencyList.get(from).add(to);
    if (hasCycle()) {
        adjacencyList.get(from).remove((Integer) to);
        return false;
    }
    return true;
}

private boolean hasCycle() {
    Set<Integer> visited = new HashSet<>();
    Set<Integer> recStack = new HashSet<>();

    for (Integer node : adjacencyList.keySet()) {
        if (dfs(node, visited, recStack)) {
            return true;
        }
    }
    return false;
}

private boolean dfs(int node, Set<Integer> visited, Set<Integer> recStack) {
    if (recStack.contains(node)) {
        return true;
    }
    if (visited.contains(node)) {
        return false;
    }
```

```
        visited.add(node);
        recStack.add(node);

        for (Integer neighbor : adjacencyList.get(node)) {
            if (dfs(neighbor, visited, recStack)) {
                return true;
            }
        }

        recStack.remove(node);
        return false;
    }

    public static void main(String[] args) {
        Graph graph = new Graph();
        graph.addNode(1);
        graph.addNode(2);
        graph.addNode(3);

        System.out.println(graph.addEdge(1, 2)); // true
        System.out.println(graph.addEdge(2, 3)); // true
        System.out.println(graph.addEdge(3, 1)); // false, creates a cycle
    }
}
```



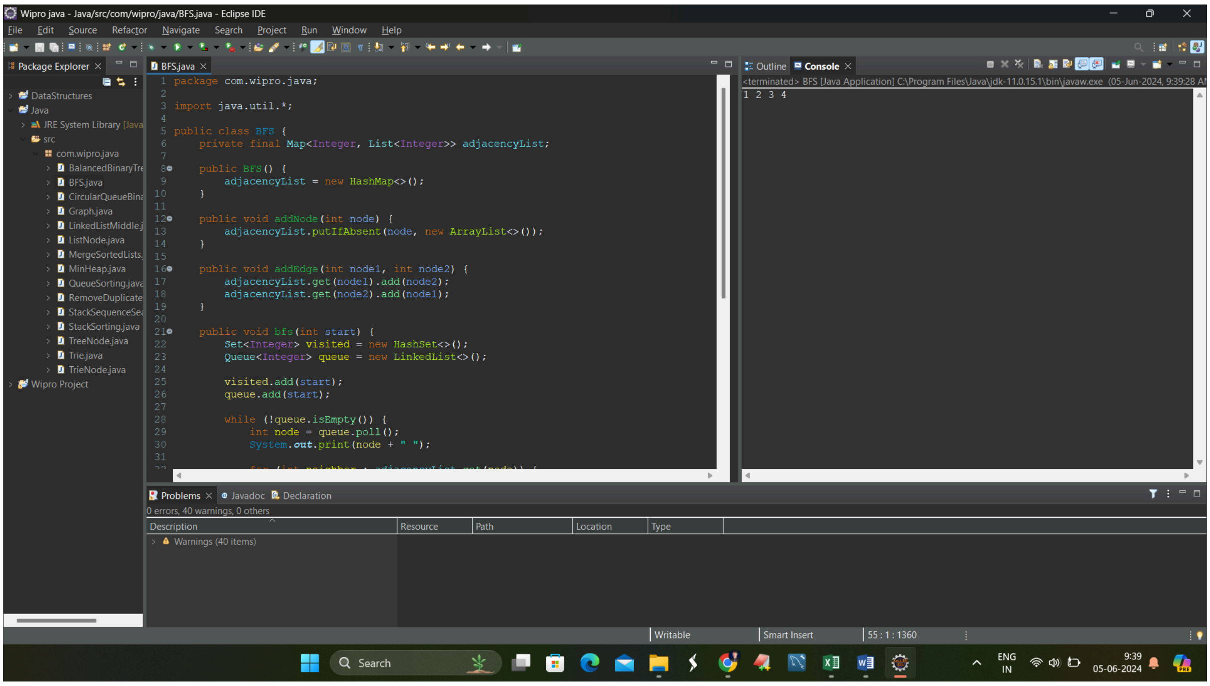
Task 5: Breadth-First Search (BFS) Implementation

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.

```
 import java.util.*;

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

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

    public void addNode(int node) {
        adjacencyList.putIfAbsent(node, new ArrayList<>());
    }

    public void addEdge(int node1, int node2) {
```

```
        adjacencyList.get(node1).add(node2);
        adjacencyList.get(node2).add(node1);
    }

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

        visited.add(start);
        queue.add(start);

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

            for (int neighbor : adjacencyList.get(node)) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.add(neighbor);
                }
            }
        }
    }

    public static void main(String[] args) {
        BFS graph = new BFS();
        graph.addNode(1);
        graph.addNode(2);
        graph.addNode(3);
        graph.addNode(4);

        graph.addEdge(1, 2);
```

```
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);

        graph.bfs(1); // Output: 1 2 3 4
    }
}
```



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.

```
import java.util.*;

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

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

```
public void addNode(int node) {
    adjacencyList.putIfAbsent(node, new ArrayList<>());
}

public void addEdge(int node1, int node2) {
    adjacencyList.get(node1).add(node2);
    adjacencyList.get(node2).add(node1);
}

public void dfs(int start) {
    Set<Integer> visited = new HashSet<>();
    dfsRecursive(start, visited);
}

private void dfsRecursive(int node, Set<Integer> visited) {
    if (visited.contains(node)) {
        return;
    }

    visited.add(node);
    System.out.print(node + " ");

    for (int neighbor : adjacencyList.get(node)) {
        if (!visited.contains(neighbor)) {
            dfsRecursive(neighbor, visited);
        }
    }
}

public static void main(String[] args) {
    DFS graph = new DFS();
```

```
        graph.addNode(1);
        graph.addNode(2);
        graph.addNode(3);
        graph.addNode(4);

        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);

        graph.dfs(1); // Output: 1 2 4 3
    }
}
```