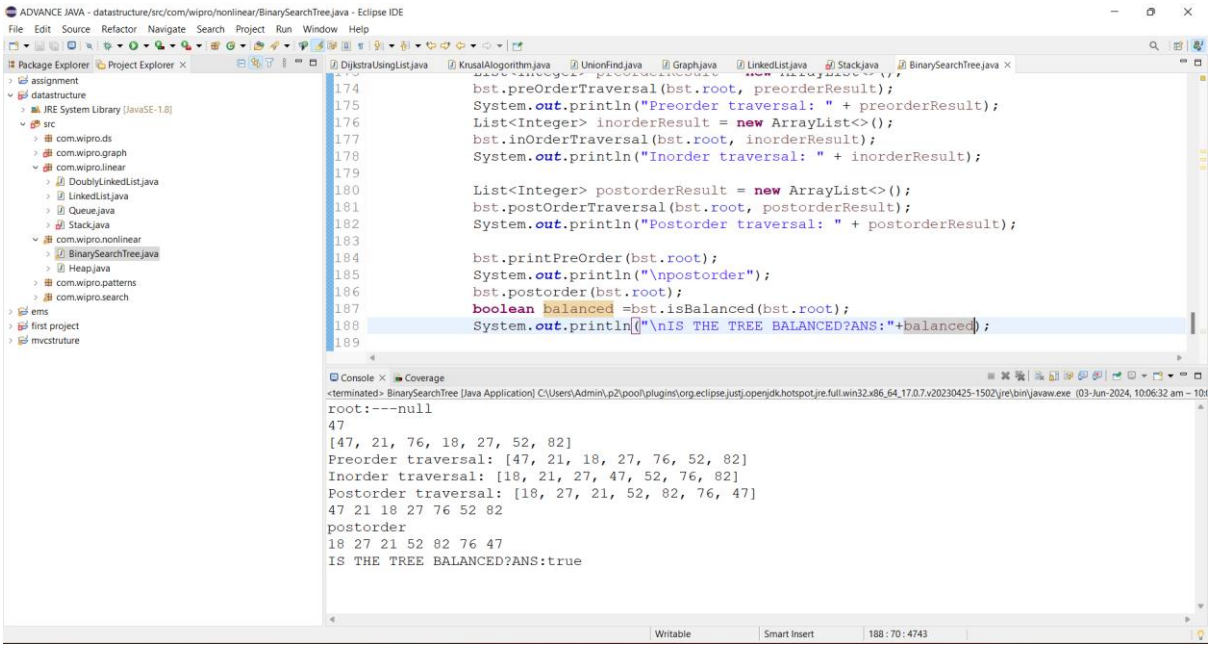
NAME:VISHAKHA AVINASH KALE

DAY7 AND DAY8

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.

```
public boolean isBalanced(Node root) {
 return checkHeight(root) != -1;
 }


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

 int leftHeight = checkHeight(node.left);
 if (leftHeight == -1) {
 return -1; // Left subtree is not balanced
 }

 int rightHeight = checkHeight(node.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; // Return the
height
 }
```

OUTPUT:



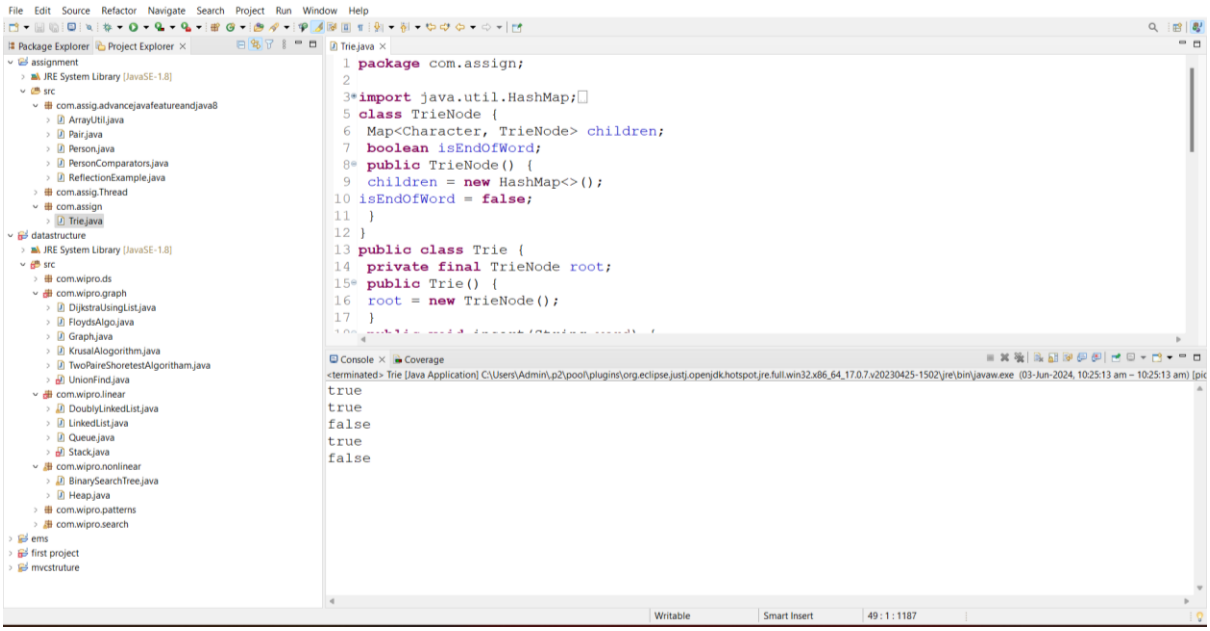
Task 2: Trie for Prefix Checking

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.

```
package com.assig.nonlinear;

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 final TrieNode root;

 public Trie() {
```

```
root = new TrieNode();
}
public void insert(String word) {
TrieNode current = root;
for (char c : word.toCharArray()) {
current.children.putIfAbsent(c, new TrieNode());
current = current.children.get(c);
}
current.isEndOfWord = true;
}
public boolean isPrefix(String prefix) {
TrieNode current = root;
for (char c : prefix.toCharArray()) {
if (!current.children.containsKey(c)) {
return false;
}
current = current.children.get(c);
}
return true;
}
public static void main(String[] args) {
Trie trie = new Trie();
trie.insert("apple");
trie.insert("app");
trie.insert("application");
trie.insert("banana");
System.out.println(trie.isPrefix("app"));
System.out.println(trie.isPrefix("ban"));
System.out.println(trie.isPrefix("bat"));
System.out.println(trie.isPrefix("appl"));
System.out.println(trie.isPrefix("apx"));
}
}
```

OUTPUT:



Task 3: Implementing Heap Operations

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

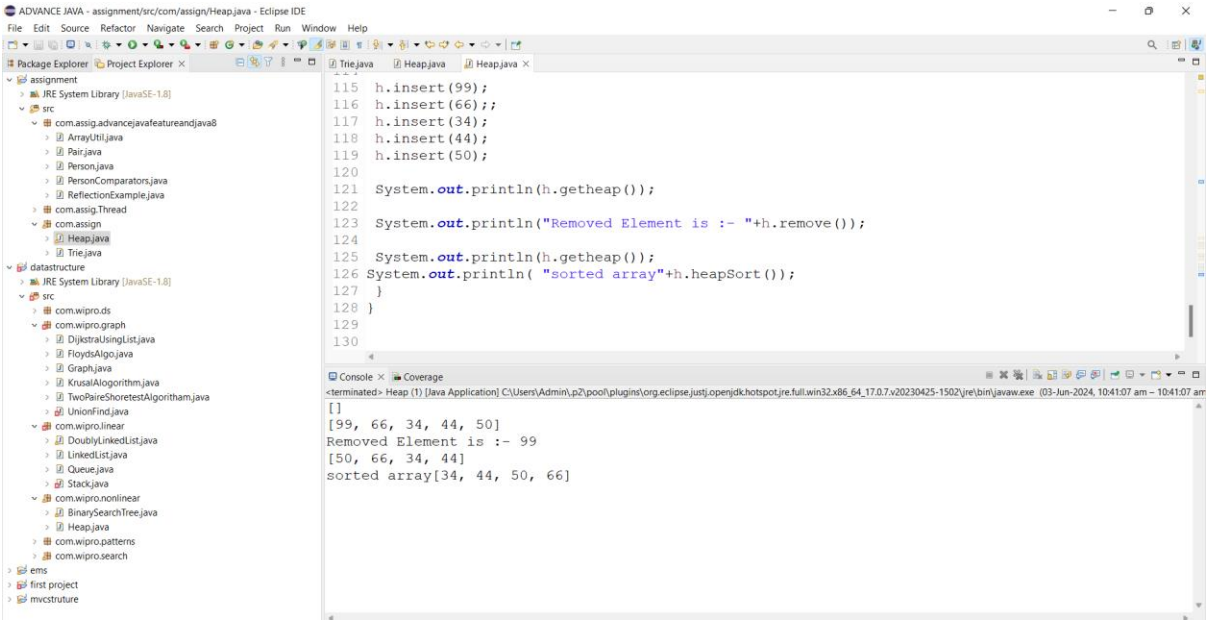
```
package com.ds.tree;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
public class Heap {
private List<Integer>heap;
 public Heap()
 {
 this.heap=new ArrayList<>();

 }

 public List<Integer> getheap()
 {
 return new ArrayList<Integer>(heap);
 }


 public int lefrchild(int index)
 {
 return (index*2)+2;
```

```
 }

 public int rightchild(int index)
 {
 return (index*2)+2;

 }
public int parent(int index)
 {
 return (index-1)/2;

 }

 public void insert(int value)
 {
 heap.add(value);
 int current=heap.size()-1;
 while(current > 0&&
heap.get(current)>heap.get(parent(current)))
 {
 swap(current,parent(current));
 current=parent(current);
 }
 }
 private void swap(int index1, int index2) {
 // TODO Auto-generated method stub

 int temp=heap.get(index1);
 heap.set(index1, heap.get(index2));
 heap.set(index2, temp);
}


 public Integer remove()
 {

 if(heap.size()==0)
 {
 return null;
```

```
 }
 if(heap.size()==1)
 {
 return heap.remove(0);

 }

 int maxvalue=heap.get(0);
 heap.set(0, heap.remove(heap.size()-1));
 sinkDown(0);
 return maxvalue;
}
 private void sinkDown(int index) {

 int maxindex=index;
 int leftindex=lefrchild(index);
 int rightindex=rightchild(index);
 if(leftindex<heap.size()&&heap.get(leftindex)>heap.get(maxind
ex))
 {
 maxindex=leftindex;

 }

 if(rightindex<heap.size()&&heap.get(rightindex)>heap.get(maxi
ndex))
 {
 maxindex=rightindex;

 }
if(maxindex!=index)
 {
 swap(index, maxindex);
 index=maxindex;
 }
 // TODO Auto-generated method stub

 }
public List<Integer> heapSort() {
// List<Integer> sortedList = new ArrayList<>();
```

```
// while (!heap.isEmpty()) {
// sortedList.add(remove());
// }

 Collections.sort(heap);
 return heap;
 }
 public static void main(String[] args) {

 Heap h=new Heap();
System.out.println(h.getheap());

 h.insert(99);
 h.insert(66);;
 h.insert(34);
 h.insert(44);
 h.insert(50);

 System.out.println(h.getheap());

 System.out.println("Removed Element is :- "+h.remove());

 System.out.println(h.getheap());
System.out.println( "sorted array"+h.heapSort());
 }
}
```

OUTPUT:



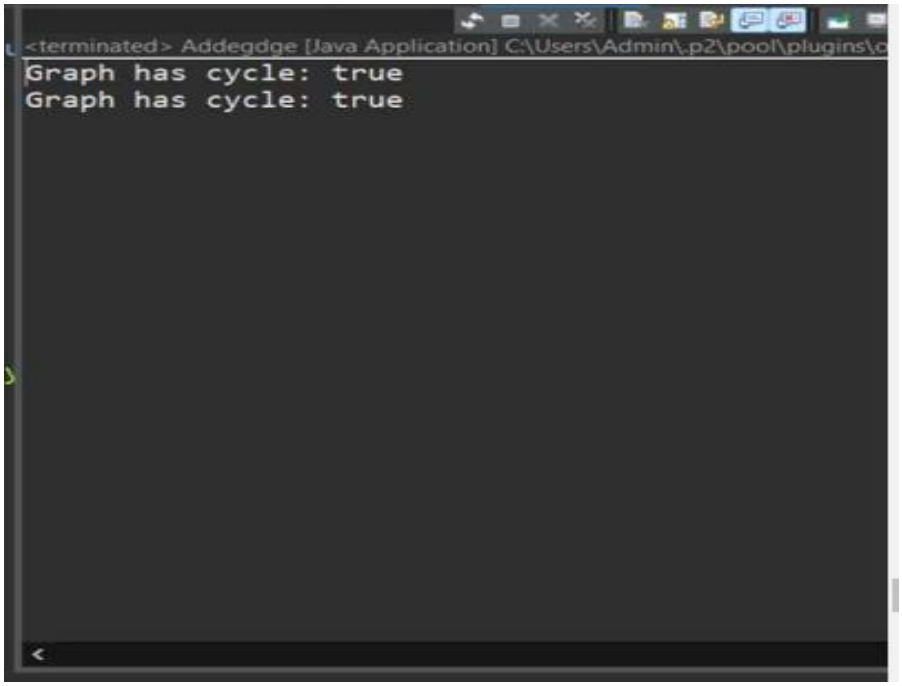
Task 4: Graph Edge Addition Validation

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.

```
package com.assig.nonlinear;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
class Graph {
 private final 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);
 }
 public boolean hasCycle(int from, int to) {
 addEdge(from, to); // Add the edge temporarily
 boolean[] visited = new boolean[adjacencyList.size() + 1];
 boolean[] recursionStack = new boolean[adjacencyList.size() +
1];
 for (int i : adjacencyList.keySet()) {
 if (!visited[i] && isCyclicUtil(i, visited, recursionStack)) {
 // Remove the temporarily added edge
 adjacencyList.get(from).remove(Integer.valueOf(to));
 return true;
 }
 }
// Remove the temporarily added edge
 adjacencyList.get(from).remove(Integer.valueOf(to));
 return false;
 }
 private boolean isCyclicUtil(int v, boolean[] visited, boolean[]
recursionStack) {
 if (recursionStack[v]) {
 return true;
```

```
 }
 if (visited[v]) {
 return false;
 }
 visited[v] = true;
 recursionStack[v] = true;
 List<Integer> neighbors = adjacencyList.getOrDefault(v, new
ArrayList<>());
 for (int neighbor : neighbors) {
 if (isCyclicUtil(neighbor, visited, recursionStack)) {
 return true;
 }
}
 recursionStack[v] = false;
 return false;
 }
}
public class Addegdge {
 public static void main(String[] args) {
 Graph graph = new Graph();
 graph.addEdge(0, 1);
 graph.addEdge(1, 2);
 graph.addEdge(2, 0);
 System.out.println("Graph has cycle: " + graph.hasCycle(2, 0)); //
Output: true
 System.out.println("Graph has cycle: " + graph.hasCycle(3, 5));
 }
}
```

OUTPUT:



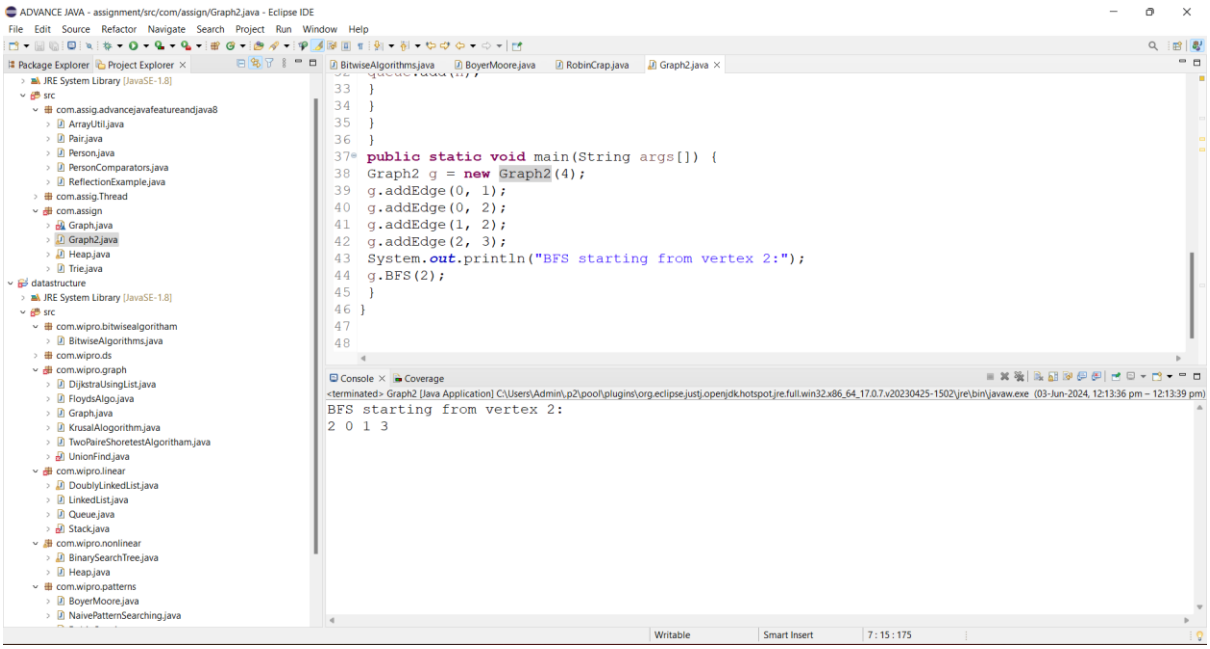
Task 5: Breadth-First Search (BFS) Implementation

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.

```
package com.assig.nonlinear;
import java.util.*;
public class Graph1 {
 private int V; // Number of vertices
 private LinkedList<Integer> adj[]; // Adjacency List public Graph1(int v) {
 V = v;
 adj = new LinkedList[v];
 for (int i = 0; i < v; ++i)
 adj[i] = new LinkedList();
 }
void addEdge(int v, int w) {
 adj[v].add(w);
 adj[w].add(v); // For undirected graph
 }
void BFS(int s) {
 boolean visited[] = new boolean[V];
 LinkedList<Integer> queue = new LinkedList<Integer>();
 visited[s] = true;
 queue.add(s);
 while (queue.size() != 0) {
 s = queue.poll();
 System.out.print(s + " ");
 Iterator<Integer> i = adj[s].listIterator();
 while (i.hasNext()) {
 int n = i.next();
```

```
if (!visited[n]) {
visited[n] = true;
queue.add(n);
}
}
}
}
public static void main(String args[]) {
Graph1 g = new Graph1(4);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 3);
System.out.println("BFS starting from vertex 2:");
g.BFS(2);
}
}
```

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.

```
package com.assig.nonlinear;
import java.util.*;
public class DFS1 {
private int V; // Number of vertices
private LinkedList<Integer> adj[]; // Adjacency List public DFS1(int v) {
V = v;
adj = new LinkedList[v];
for (int i = 0; i < v; ++i)
adj[i] = new LinkedList();
```

```
 }
void addEdge(int v, int w) {
 adj[v].add(w);
 adj[w].add(v); // For undirected graph
 }
void DFSUtil(int v, boolean visited[]) {
 visited[v] = true;
 System.out.print(v + " ");
Iterator<Integer> i = adj[v].listIterator();
 while (i.hasNext()) {
 int n = i.next();
 if (!visited[n])
 DFSUtil(n, visited);
 }
 }
void DFS(int v) {
 boolean visited[] = new boolean[V];
 DFSUtil(v, visited);
 }
public static void main(String args[]) {
 DFS1 g = new DFS1(4);
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 3);
 System.out.println("DFS starting from vertex 2:"); g.DFS(2);
 }
}
```

OUTPUT: