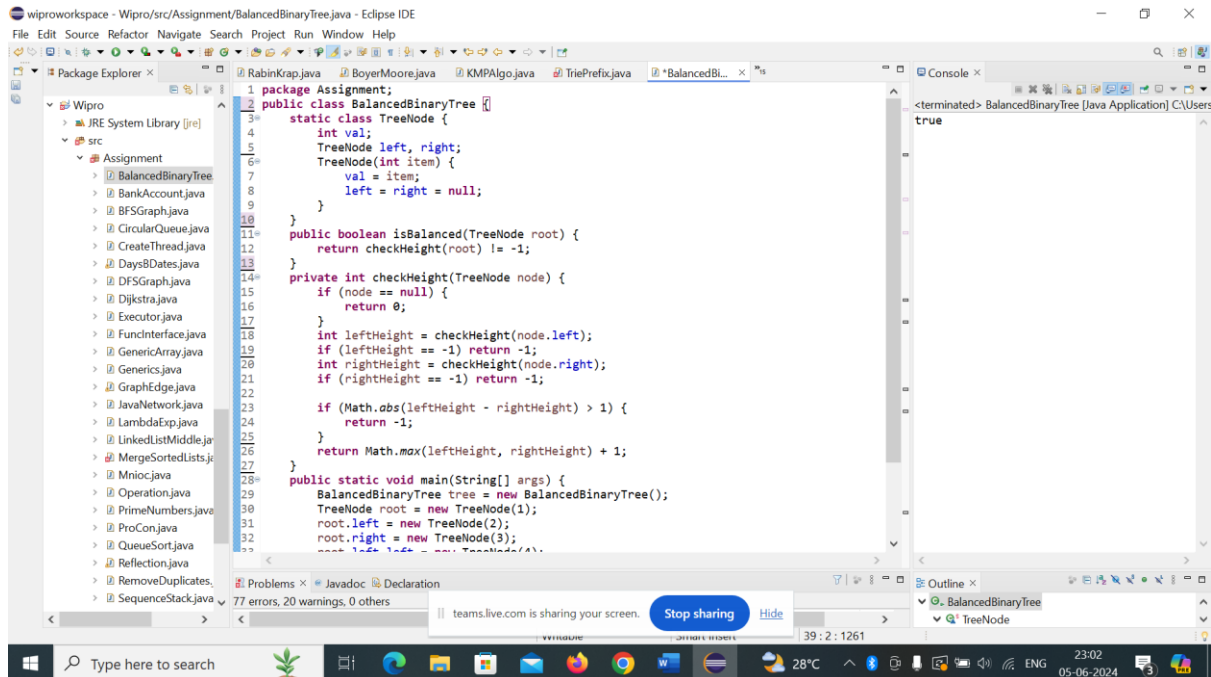


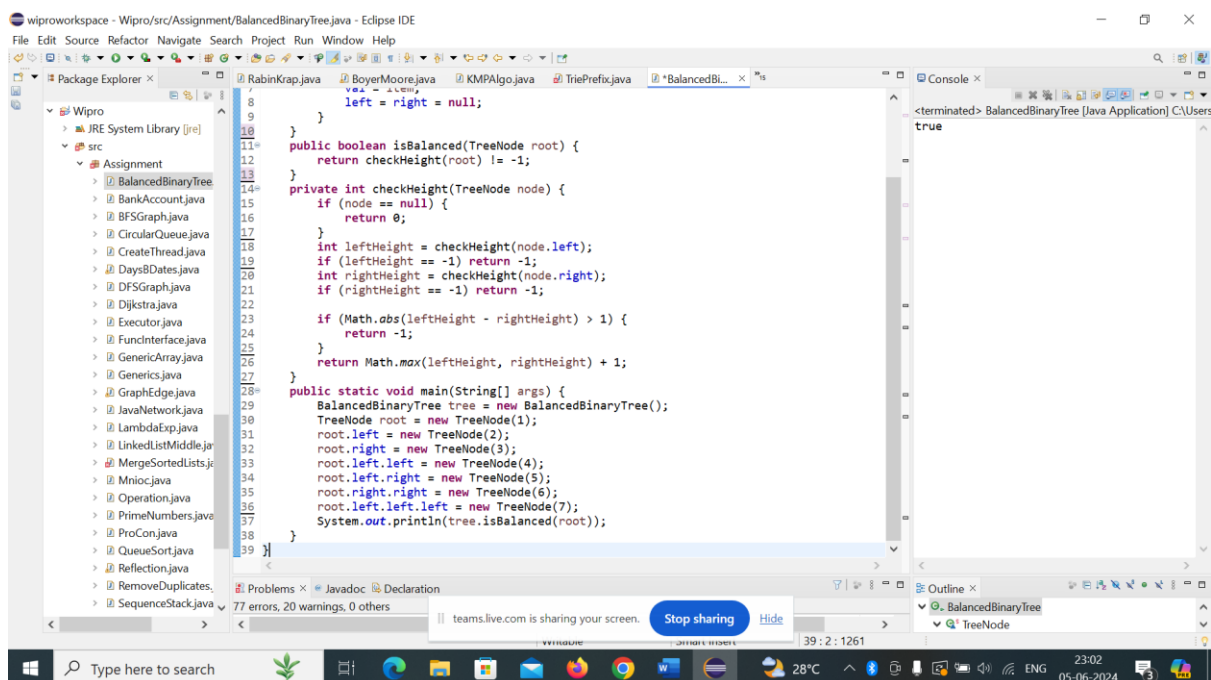
Day 7 and 8

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.



```
1 package Assignment;
2 public class BalancedBinaryTree {
3     static class TreeNode {
4         int val;
5         TreeNode left, right;
6         TreeNode(int item) {
7             val = item;
8             left = right = null;
9         }
10    }
11    public boolean isBalanced(TreeNode root) {
12        return checkHeight(root) != -1;
13    }
14    private int checkHeight(TreeNode node) {
15        if (node == null) {
16            return 0;
17        }
18        int leftHeight = checkHeight(node.left);
19        if (leftHeight == -1) return -1;
20        int rightHeight = checkHeight(node.right);
21        if (rightHeight == -1) return -1;
22        if (Math.abs(leftHeight - rightHeight) > 1) {
23            return -1;
24        }
25        return Math.max(leftHeight, rightHeight) + 1;
26    }
27    public static void main(String[] args) {
28        BalancedBinaryTree tree = new BalancedBinaryTree();
29        TreeNode root = new TreeNode(1);
30        root.left = new TreeNode(2);
31        root.right = new TreeNode(3);
32    }
33 }
```

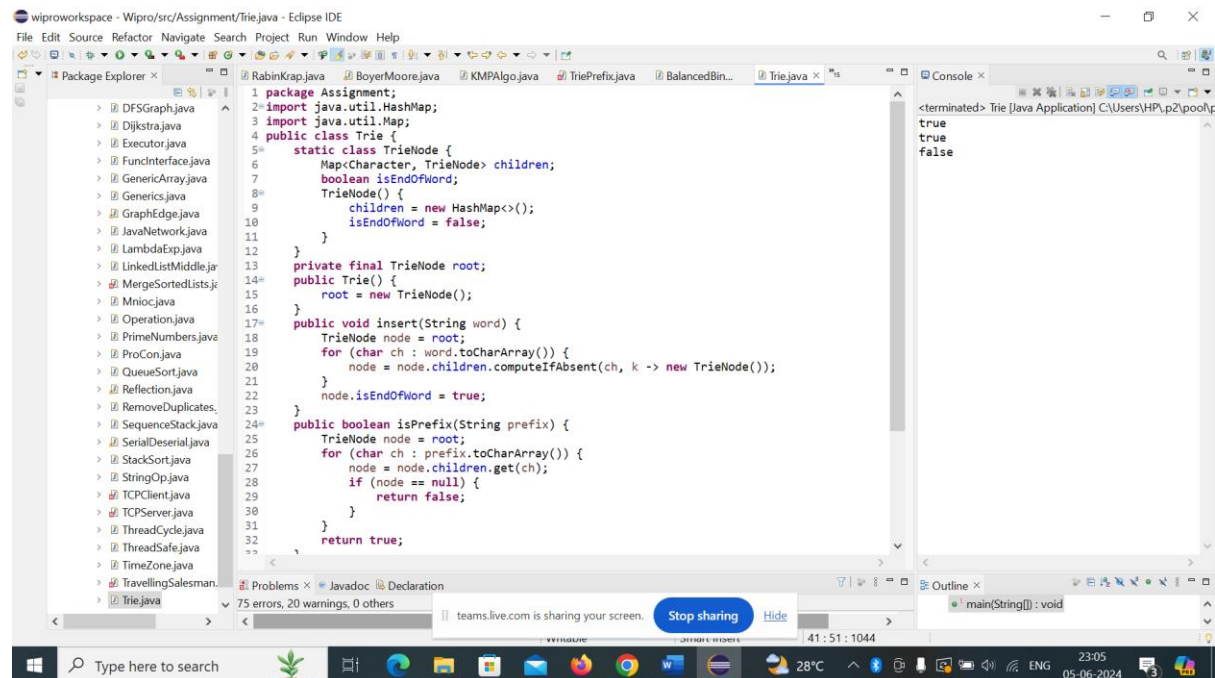


```
8         left = right = null;
9     }
10 }
11 public boolean isBalanced(TreeNode root) {
12     return checkHeight(root) != -1;
13 }
14 private int checkHeight(TreeNode node) {
15     if (node == null) {
16         return 0;
17     }
18     int leftHeight = checkHeight(node.left);
19     if (leftHeight == -1) return -1;
20     int rightHeight = checkHeight(node.right);
21     if (rightHeight == -1) return -1;
22     if (Math.abs(leftHeight - rightHeight) > 1) {
23         return -1;
24     }
25     return Math.max(leftHeight, rightHeight) + 1;
26 }
27 }
28 public static void main(String[] args) {
29     BalancedBinaryTree tree = new BalancedBinaryTree();
30     TreeNode root = new TreeNode(1);
31     root.left = new TreeNode(2);
32     root.right = new TreeNode(3);
33     root.left.left = new TreeNode(4);
34     root.left.right = new TreeNode(5);
35     root.right.right = new TreeNode(6);
36     root.left.left.left = new TreeNode(7);
37     System.out.println(tree.isBalanced(root));
38 }
39 }
```

Console output: true

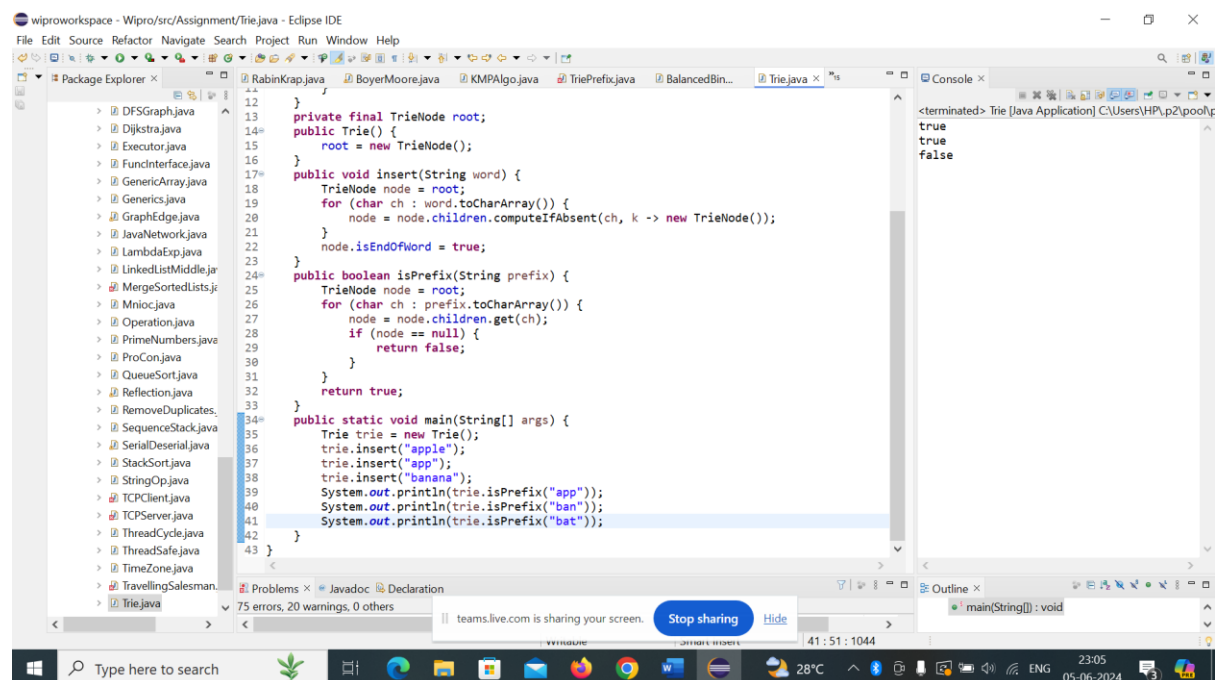
Task 2: Trie for Prefix Checking

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.



The screenshot shows the Eclipse IDE with the 'Trie.java' file open. The code defines a 'Trie' class with a 'TrieNode' inner class. The 'TrieNode' class has a 'Map<Character, TrieNode> children' and a 'boolean isEndOfWord'. The 'Trie' class has a 'root' of type 'TrieNode'. The 'insert' method takes a 'String word' and iterates through its characters, creating new 'TrieNode' objects as needed. The 'isPrefix' method takes a 'String prefix' and checks if it is a prefix of any word in the trie. The console shows the output of the 'main' method, which prints 'true', 'true', and 'false' for the prefixes 'app', 'ban', and 'bat' respectively.

```
1 package Assignment;
2 import java.util.HashMap;
3 import java.util.Map;
4 public class Trie {
5     static class TrieNode {
6         Map<Character, TrieNode> children;
7         boolean isEndOfWord;
8         TrieNode() {
9             children = new HashMap<>();
10            isEndOfWord = false;
11        }
12    }
13    private final TrieNode root;
14    public Trie() {
15        root = new TrieNode();
16    }
17    public void insert(String word) {
18        TrieNode node = root;
19        for (char ch : word.toCharArray()) {
20            node = node.children.computeIfAbsent(ch, k -> new TrieNode());
21        }
22        node.isEndOfWord = true;
23    }
24    public boolean isPrefix(String prefix) {
25        TrieNode node = root;
26        for (char ch : prefix.toCharArray()) {
27            node = node.children.get(ch);
28            if (node == null) {
29                return false;
30            }
31        }
32        return true;
33    }
34}
```

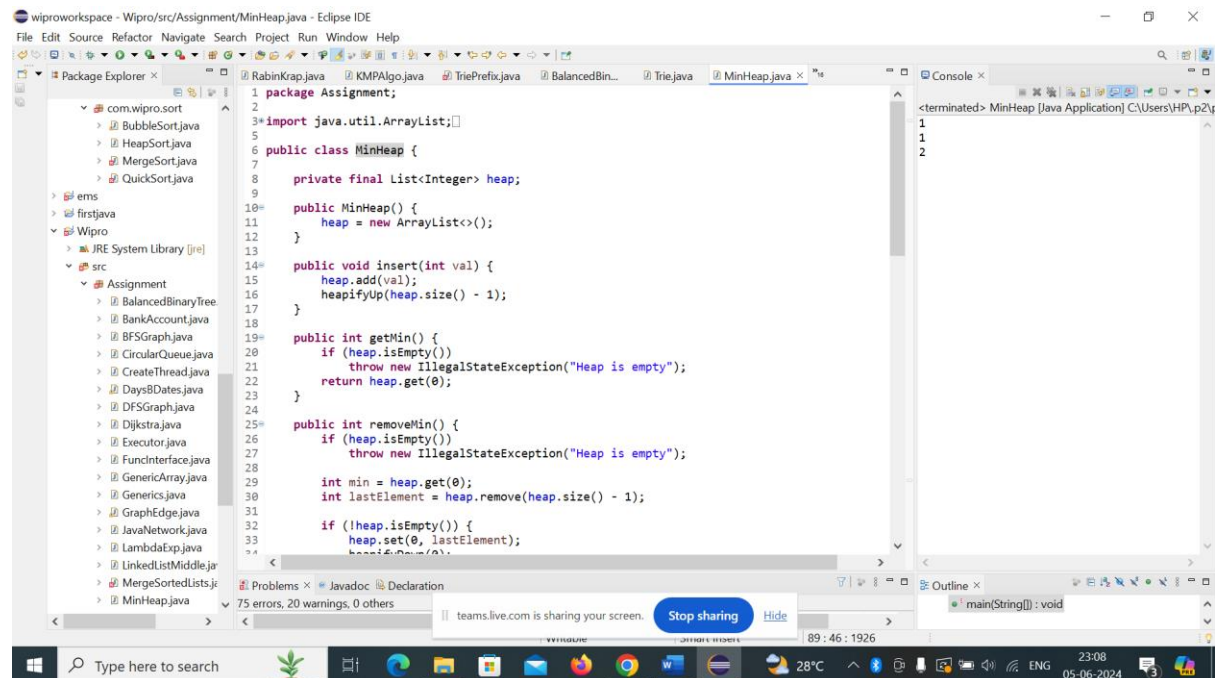


The screenshot shows the Eclipse IDE with the 'Trie.java' file open. The code is the same as the previous screenshot, but it includes a 'main' method that creates a 'Trie' object, inserts the words 'apple' and 'banana', and checks if 'app', 'ban', and 'bat' are prefixes. The console shows the output of the 'main' method, which prints 'true', 'true', and 'false' for the prefixes 'app', 'ban', and 'bat' respectively.

```
1 package Assignment;
2 import java.util.HashMap;
3 import java.util.Map;
4 public class Trie {
5     static class TrieNode {
6         Map<Character, TrieNode> children;
7         boolean isEndOfWord;
8         TrieNode() {
9             children = new HashMap<>();
10            isEndOfWord = false;
11        }
12    }
13    private final TrieNode root;
14    public Trie() {
15        root = new TrieNode();
16    }
17    public void insert(String word) {
18        TrieNode node = root;
19        for (char ch : word.toCharArray()) {
20            node = node.children.computeIfAbsent(ch, k -> new TrieNode());
21        }
22        node.isEndOfWord = true;
23    }
24    public boolean isPrefix(String prefix) {
25        TrieNode node = root;
26        for (char ch : prefix.toCharArray()) {
27            node = node.children.get(ch);
28            if (node == null) {
29                return false;
30            }
31        }
32        return true;
33    }
34    public static void main(String[] args) {
35        Trie trie = new Trie();
36        trie.insert("apple");
37        trie.insert("banana");
38        System.out.println(trie.isPrefix("app"));
39        System.out.println(trie.isPrefix("ban"));
40        System.out.println(trie.isPrefix("bat"));
41    }
42}
```

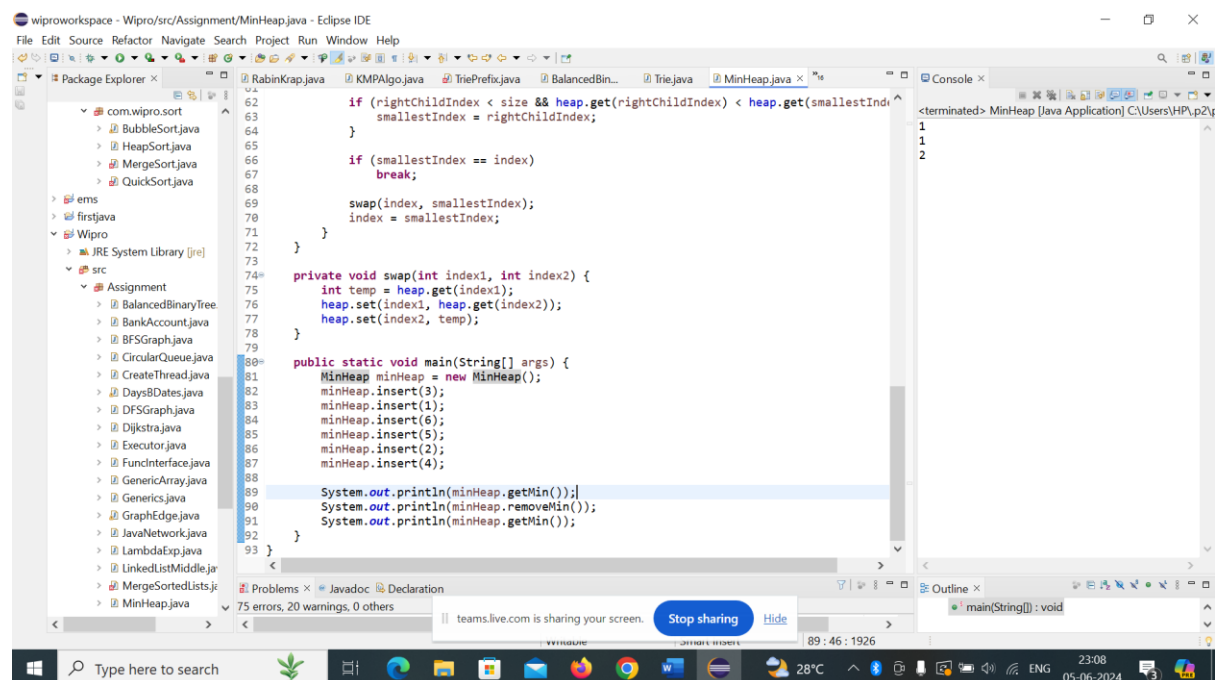
Task 3: Implementing Heap Operations

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.



The screenshot shows the Eclipse IDE with the `MinHeap.java` file open. The code defines a `MinHeap` class that uses an `ArrayList` to store elements. It includes methods for `insert`, `getMin`, and `removeMin`. The `insert` method adds an element and calls `heapifyUp`. The `getMin` method checks if the heap is empty and returns the first element. The `removeMin` method removes the first element and calls `heapifyUp` to maintain the heap property. The console shows the output of the `main` method, which prints the minimum element (1) and the heap after removal (1, 2).

```
1 package Assignment;
2
3 import java.util.ArrayList;
4
5 public class MinHeap {
6
7     private final List<Integer> heap;
8
9     public MinHeap() {
10         heap = new ArrayList<>();
11     }
12
13     public void insert(int val) {
14         heap.add(val);
15         heapifyUp(heap.size() - 1);
16     }
17
18     public int getMin() {
19         if (heap.isEmpty())
20             throw new IllegalStateException("Heap is empty");
21         return heap.get(0);
22     }
23
24     public int removeMin() {
25         if (heap.isEmpty())
26             throw new IllegalStateException("Heap is empty");
27
28         int min = heap.get(0);
29         int lastElement = heap.remove(heap.size() - 1);
30
31         if (!heap.isEmpty()) {
32             heap.set(0, lastElement);
33             heapifyUp(0);
34         }
35     }
36 }
```



The screenshot shows the Eclipse IDE with the `MinHeap.java` file open, displaying the completed code. The `insert` method is now implemented with a `heapifyUp` method that recursively maintains the heap property. The `removeMin` method is also implemented with a `heapifyUp` method. The `main` method is completed, showing the insertion of elements 3, 1, 6, 5, 2, and 4, followed by printing the minimum element (1), removing it, and printing the minimum element again (1). The console shows the output of the `main` method, which prints the minimum element (1) and the heap after removal (1, 2).

```
62         if (rightChildIndex < size && heap.get(rightChildIndex) < heap.get(smallestIndex))
63             smallestIndex = rightChildIndex;
64     }
65
66     if (smallestIndex == index)
67         break;
68
69     swap(index, smallestIndex);
70     index = smallestIndex;
71 }
72
73 private void swap(int index1, int index2) {
74     int temp = heap.get(index1);
75     heap.set(index1, heap.get(index2));
76     heap.set(index2, temp);
77 }
78
79 public static void main(String[] args) {
80     MinHeap minHeap = new MinHeap();
81     minHeap.insert(3);
82     minHeap.insert(1);
83     minHeap.insert(6);
84     minHeap.insert(5);
85     minHeap.insert(2);
86     minHeap.insert(4);
87
88     System.out.println(minHeap.getMin());
89     System.out.println(minHeap.removeMin());
90     System.out.println(minHeap.getMin());
91 }
92 }
```