Day 7 and 8: Manisha Assignment

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.

Explanation

1. TreeNode Class:

- A simple class to represent each node in the binary tree. Each node has a value (`val`), a left child (`left`), and a right child (`right`).

2. BalancedBinaryTree Class:

- Contains the main logic to check if the binary tree is balanced.

3. checkHeight Method:

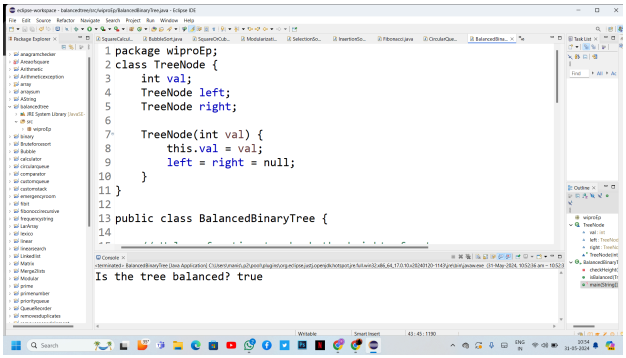
- This is a helper method that recursively checks the height of the tree.
- If a subtree is unbalanced, it returns `-1`.
- Otherwise, it returns the height of the tree.

4. isBalanced Method:

- This method uses `checkHeight` to determine if the tree is balanced.
- It returns `true` if the tree is balanced and `false` otherwise.

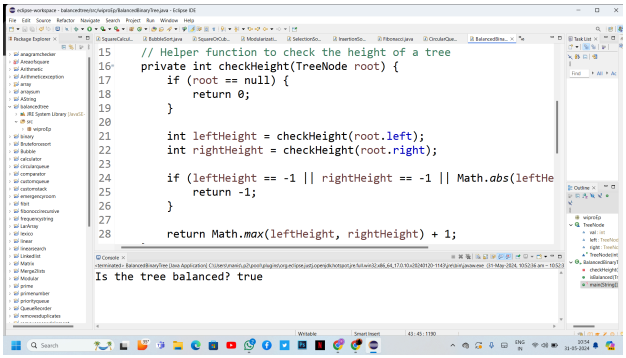
5. main Method:

- Constructs a sample balanced binary tree.
- Calls the `isBalanced` method to check if the tree is balanced.
- Prints the result.



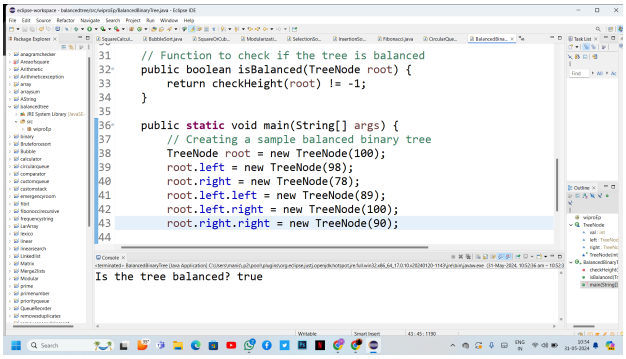
```
package wiproEp;
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int val) {
        this.val = val;
        left = right = null;
    }
}
public class BalancedBinaryTree {
```

```
Is the tree balanced? true
```



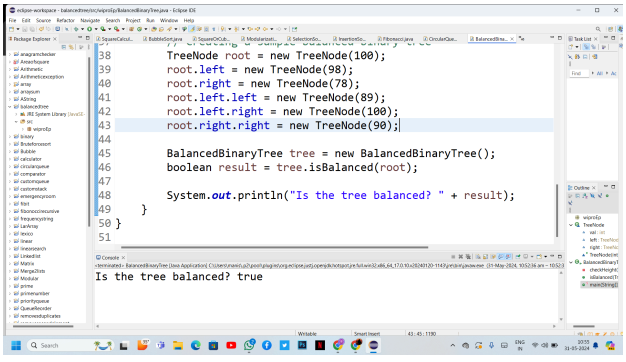
```
    // Helper function to check the height of a tree
    private int checkHeight(TreeNode root) {
        if (root == null) {
            return 0;
        }

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

        if (leftHeight == -1 || rightHeight == -1 || Math.abs(leftHe
            return -1;
        }

        return Math.max(leftHeight, rightHeight) + 1;
```

```
Is the tree balanced? true
```



```
    // Function to check if the tree is balanced
    public boolean isBalanced(TreeNode root) {
        return checkHeight(root) != -1;
    }

    public static void main(String[] args) {
        // Creating a sample balanced binary tree
        TreeNode root = new TreeNode(100);
        root.left = new TreeNode(98);
        root.right = new TreeNode(78);
        root.left.left = new TreeNode(89);
        root.left.right = new TreeNode(100);
        root.right.right = new TreeNode(90);
```

```
Is the tree balanced? true
```



```
        TreeNode root = new TreeNode(100);
        root.left = new TreeNode(98);
        root.right = new TreeNode(78);
        root.left.left = new TreeNode(89);
        root.left.right = new TreeNode(100);
        root.right.right = new TreeNode(90);

        BalancedBinaryTree tree = new BalancedBinaryTree();
        boolean result = tree.isBalanced(root);

        System.out.println("Is the tree balanced? " + result);
    }
}
```

```
Is the tree balanced? 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.

Explanation

1. TrieNode Class:

- Represents each node in the trie.
- Contains an array of `TrieNode` references for each letter in the alphabet (26 for lowercase English letters).
- A boolean `isEndOfWord` to mark the end of a word.

2. Trie Class:

- Contains the root node and methods to insert words and check for prefixes.

3. insert Method:

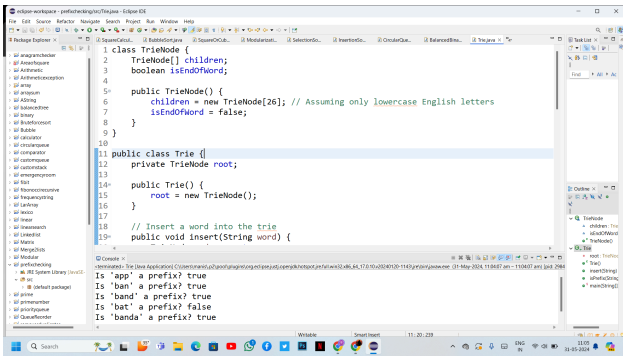
- Inserts a word into the trie by iterating through each character, creating new nodes as needed, and marking the end of the word.

4. isPrefix Method:

- Checks if a given string is a prefix by traversing the trie nodes according to the characters in the prefix.
- Returns `true` if all characters in the prefix are found in the trie, otherwise `false`.

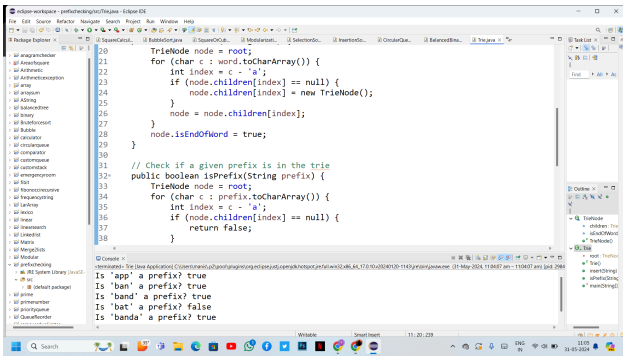
5. main Method:

- Demonstrates how to use the `Trie` class.
- Inserts several words into the trie.
- Tests the `isPrefix` method with different prefixes and prints the results.



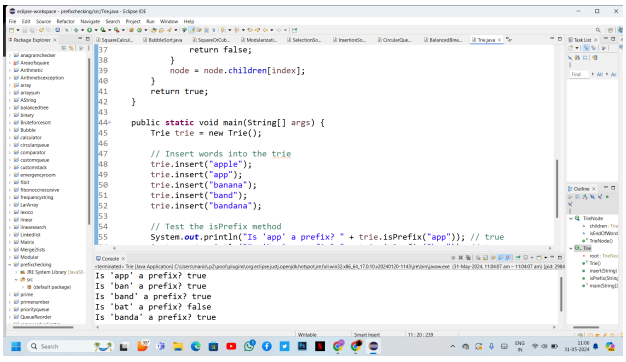
```
class TrieNode {
    TrieNode[] children;
    boolean isEndOfWord;

    public TrieNode() {
        children = new TrieNode[26]; // Assuming only lowercase English letters
        isEndOfWord = false;
    }
}

public class Trie {
    private TrieNode root;

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

    // Insert a word into the trie
    public void insert(String word) {
```

```
Is 'app' a prefix? true
Is 'ban' a prefix? true
Is 'band' a prefix? true
Is 'bat' a prefix? false
Is 'banda' a prefix? true
```



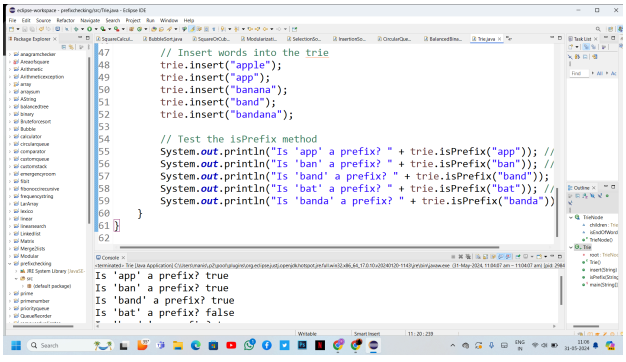
```
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                node.children[index] = new TrieNode();
            }
            node = node.children[index];
        }
        node.isEndOfWord = true;
    }

    // Check if a given prefix is in the trie
    public boolean isPrefix(String prefix) {
        TrieNode node = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (node.children[index] == null) {
                return false;
            }
```

```
Is 'app' a prefix? true
Is 'ban' a prefix? true
Is 'band' a prefix? true
Is 'bat' a prefix? false
Is 'banda' a prefix? true
```



```
                return false;
            }
            node = node.children[index];
        }
        return true;
    }

    public static void main(String[] args) {
        Trie trie = new Trie();

        // Insert words into the trie
        trie.insert("apple");
        trie.insert("app");
        trie.insert("banana");
        trie.insert("band");
        trie.insert("bandana");

        // Test the isPrefix method
        System.out.println("Is 'app' a prefix? " + trie.isPrefix("app")); // true
```

```
Is 'app' a prefix? true
Is 'ban' a prefix? true
Is 'band' a prefix? true
Is 'bat' a prefix? false
Is 'banda' a prefix? true
```



```
        // Insert words into the trie
        trie.insert("apple");
        trie.insert("app");
        trie.insert("banana");
        trie.insert("band");
        trie.insert("bandana");

        // Test the isPrefix method
        System.out.println("Is 'app' a prefix? " + trie.isPrefix("app")); //
        System.out.println("Is 'ban' a prefix? " + trie.isPrefix("ban")); //
        System.out.println("Is 'band' a prefix? " + trie.isPrefix("band"));
        System.out.println("Is 'bat' a prefix? " + trie.isPrefix("bat")); //
        System.out.println("Is 'banda' a prefix? " + trie.isPrefix("banda"))
    }
}
```

```
Is 'app' a prefix? true
Is 'ban' a prefix? true
Is 'band' a prefix? true
Is 'bat' a prefix? 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.

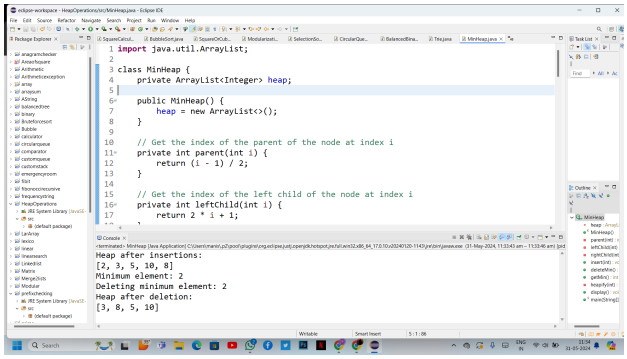
Explanation

1. MinHeap Class:

- **Constructor:** Initializes an empty ArrayList to store heap elements.
- parent, leftChild, rightChild: Helper methods to calculate parent and child indices.
- **insert**: Adds a new value to the heap and maintains the heap property by comparing and swapping values up the tree.
- **deleteMin**: Removes and returns the minimum element (root) from the heap. Replaces the root with the last element and heapifies down to maintain the heap property.
- **getMin:** Returns the minimum element without removing it.
- **heapify**: Ensures the heap property is maintained by comparing parent and child nodes and swapping if necessary.
- display: Prints the elements of the heap.

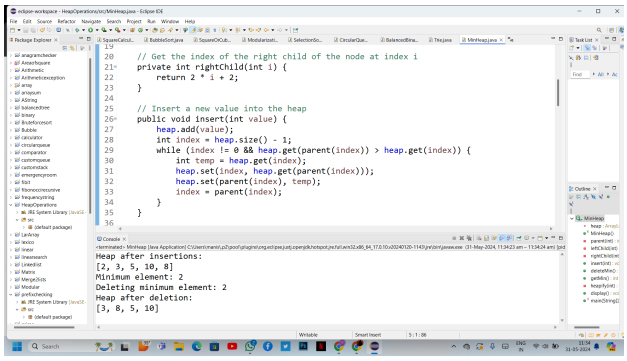
2. Main Method:

- Demonstrates inserting elements into the heap, fetching the minimum element, and deleting the minimum element.
- Displays the heap after each operation.



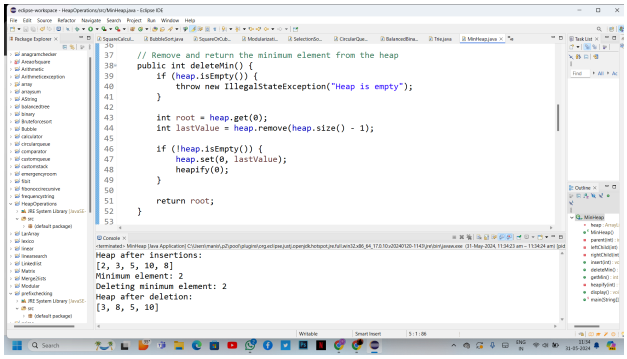
```
import java.util.ArrayList;

class MinHeap {
    private ArrayList<Integer> heap;

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

    // Get the index of the parent of the node at index i
    private int parent(int i) {
        return (i - 1) / 2;
    }

    // Get the index of the left child of the node at index i
    private int leftChild(int i) {
        return 2 * i + 1;
```

```
Heap after insertions:
[2, 3, 5, 10, 8]
Minimum element: 2
Deleting minimum element: 2
Heap after deletion:
[3, 8, 5, 10]
```



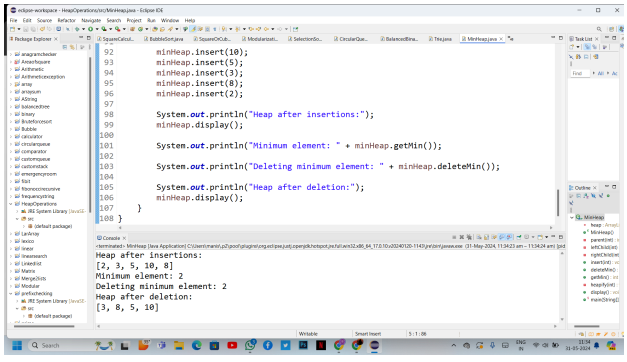
```
    // Get the index of the right child of the node at index i
    private int rightChild(int i) {
        return 2 * i + 2;
    }

    // Insert a new value into the heap
    public void insert(int value) {
        heap.add(value);
        int index = heap.size() - 1;
        while (index != 0 && heap.get(parent(index)) > heap.get(index)) {
            int temp = heap.get(index);
            heap.set(index, heap.get(parent(index)));
            heap.set(parent(index), temp);
            index = parent(index);
        }
    }
```

```
Heap after insertions:
[2, 3, 5, 10, 8]
Minimum element: 2
Deleting minimum element: 2
Heap after deletion:
[3, 8, 5, 10]
```



```
    // Remove and return the minimum element from the heap
    public int deleteMin() {
        if (heap.isEmpty()) {
            throw new IllegalStateException("Heap is empty");
        }

        int root = heap.get(0);
        int lastValue = heap.remove(heap.size() - 1);

        if (!heap.isEmpty()) {
            heap.set(0, lastValue);
            heapify(0);
        }

        return root;
    }
```

```
Heap after insertions:
[2, 3, 5, 10, 8]
Minimum element: 2
Deleting minimum element: 2
Heap after deletion:
[3, 8, 5, 10]
```



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

        System.out.println("Heap after insertions:");
        minHeap.display();

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

        System.out.println("Deleting minimum element: " + minHeap.deleteMin());

        System.out.println("Heap after deletion:");
        minHeap.display();
    }
}
```

```
Heap after insertions:
[2, 3, 5, 10, 8]
Minimum element: 2
Deleting minimum element: 2
Heap after deletion:
[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.

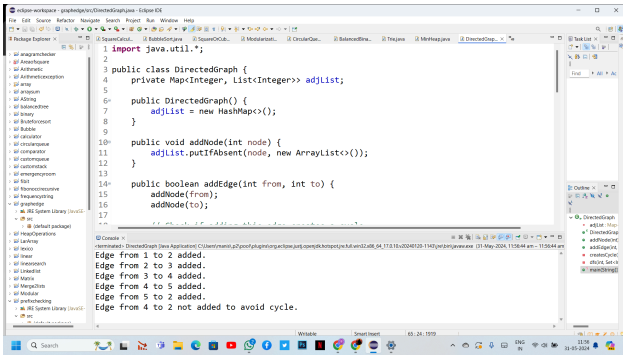
Explanation

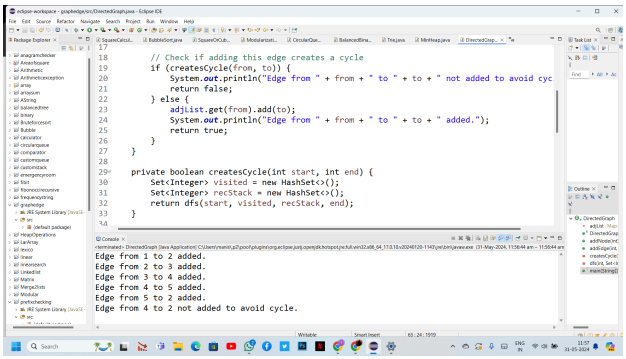
1. **DirectedGraph Class:**
   - adjList: A map representing the adjacency list of the graph.
   - addNode: Adds a node to the adjacency list if it doesn't already exist.
   - addEdge: Adds an edge between two nodes if it doesn't create a cycle. It first checks for potential cycles using the `createsCycle` method.
   - createsCycle: Uses depth-first search (DFS) to check if adding an edge creates a cycle.
   - dfs: A helper method for performing DFS. It checks if a cycle is detected by maintaining a recursion stack (`recStack`).

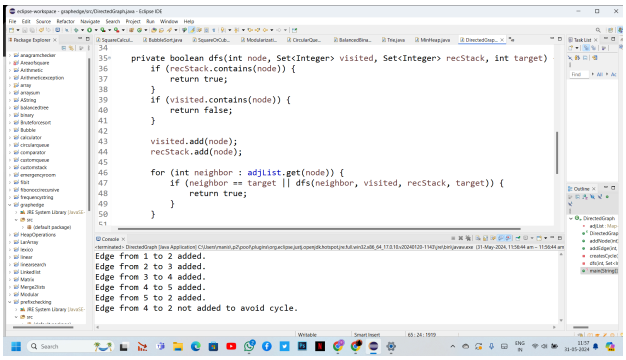
2. **Main Method:**
   - Creates an instance of `DirectedGraph`.
   - Adds edges between nodes.
   - Attempts to add an edge that would create a cycle, demonstrating the cycle detection mechanism.

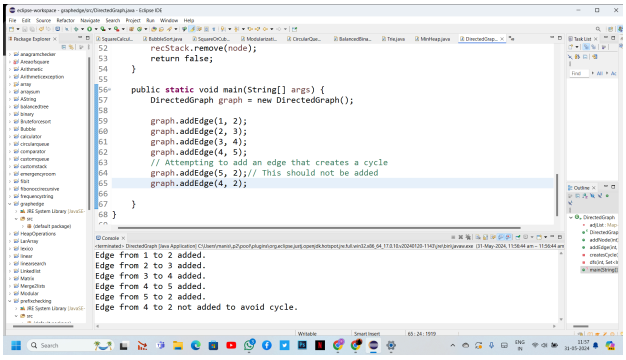
Running the Program in Eclipse

1. Create a New Java Project: In Eclipse, go to `File -> New -> Java Project`. Name the project, e.g., `DirectedGraphExample`.
2. Create a New Java Class: Right-click on the `src` folder in your project, then go to `New -> Class`. Name the class `DirectedGraph` and include the `public static void main(String[] args)` method.
3. Copy and Paste the Code: Copy the provided code and paste it into your `DirectedGraph` class.
4. Run the Program: Right-click on your `DirectedGraph` class in the Eclipse project explorer, then select `Run As -> Java Application`.









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.

Explanation

1. Graph Class:

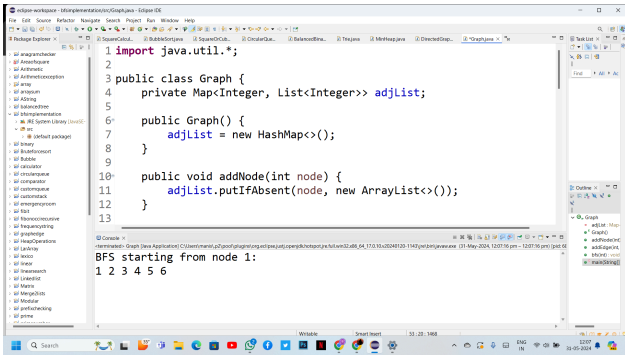
- **adjList**: A map representing the adjacency list of the graph.
- **addNode:** Adds a node to the adjacency list if it doesn't already exist.
- **addEdge:** Adds an undirected edge between two nodes.
- **bfs:** Implements the Breadth-First Search algorithm. It starts from the given node, visits all the adjacent nodes, and prints each node in the order it is visited.

2. Main Method:

- Creates an instance of `Graph`.
- Adds edges between nodes to form an undirected graph.
- Calls the `bfs` method to start the BFS traversal from node 1.

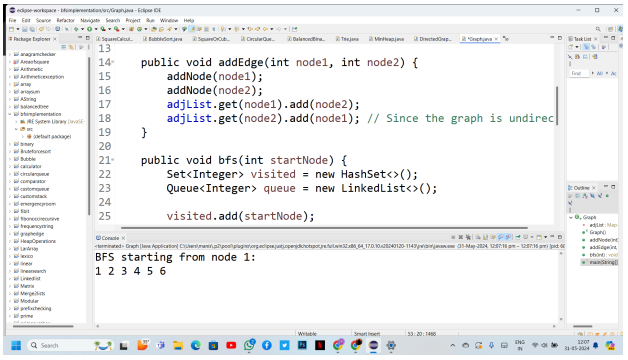
Running the Program in Eclipse

1. **Create a New Java Project:** In Eclipse, go to `File -> New -> Java Project`. Name the project, e.g., `GraphExample`.
2. **Create a New Java Class:** Right-click on the `src` folder in your project, then go to `New -> Class`. Name the class `Graph` and include the `public static void main(String[] args)` method.
3. **Copy and Paste the Code:** Copy the provided code and paste it into your `Graph` class.
4. **Run the Program:** Right-click on your `Graph` class in the Eclipse project explorer, then select `Run As -> Java Application`.



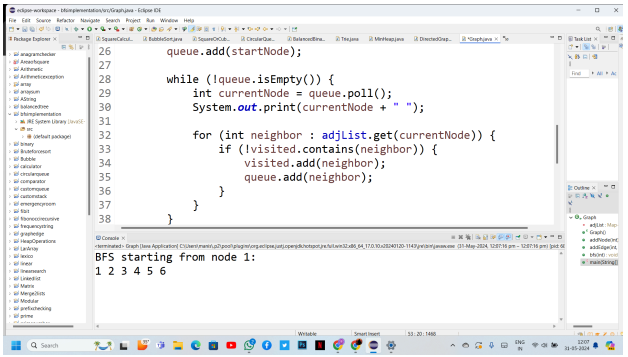
```
import java.util.*;

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

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

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

```
BFS starting from node 1:
1 2 3 4 5 6
```



```
    public void addEdge(int node1, int node2) {
        addNode(node1);
        addNode(node2);
        adjList.get(node1).add(node2);
        adjList.get(node2).add(node1); // Since the graph is undirec
    }

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

        visited.add(startNode);
```

```
BFS starting from node 1:
1 2 3 4 5 6
```



```
        queue.add(startNode);

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

            for (int neighbor : adjList.get(currentNode)) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.add(neighbor);
                }
            }
        }
```

```
BFS starting from node 1:
1 2 3 4 5 6
```

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.

Explanation

- Graph Initialization The graph is initialized with 6 vertices.
- Adding Edge: Edges are added between the vertices to form the graph.
- DFS Traversal: Starting from vertex 0, the `dfs` function recursively visits each vertex, marking it as visited and printing it out.
- Visited Array: The `visited` array ensures that each vertex is visited only once, preventing infinite loops in the case of cycles in the graph.

Running the Program in Eclipse

1. Open Eclipse: Create a new Java project and a new Java class named `GraphDFS`.
2. Copy the Code: Copy the provided code into the `GraphDFS.java` file.
3. Run the Program: Right-click on the file and select `Run As -> Java Application`.
4. View Output: The output will be displayed in the Eclipse console, showing the DFS traversal starting from vertex 0.