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.

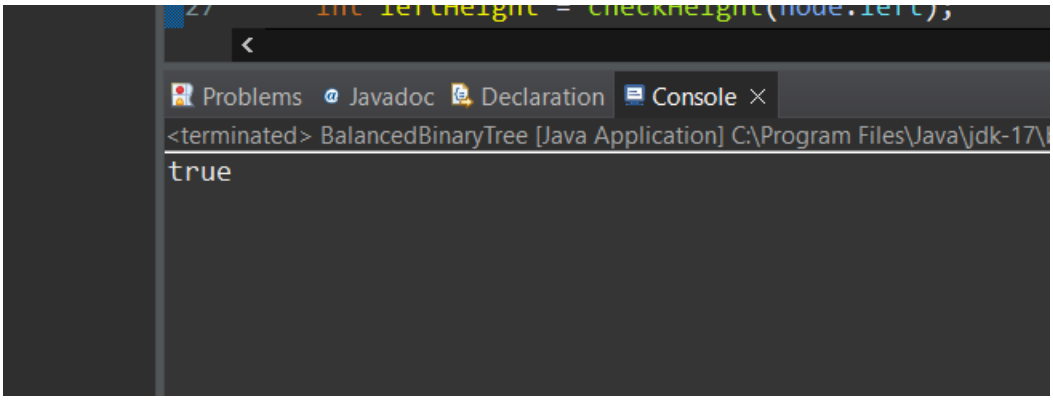
Ans:

```
package AssignmentDay7_8;



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

 TreeNode(int x) {
    val = x;
    left = null;
    right = null;
 }
}

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) {

    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.right = new TreeNode(6);
    root.left.left.left = new TreeNode(7);

    BalancedBinaryTree tree = new BalancedBinaryTree();
```

```
    System.out.println(tree.isBalanced(root));
 }
}
```

Output:



Task 2: Trie for Prefix Checking
Implement a trie data structure in Java that supports insertion of strings and provides a method to check if a given string is a prefix of any word in the trie.

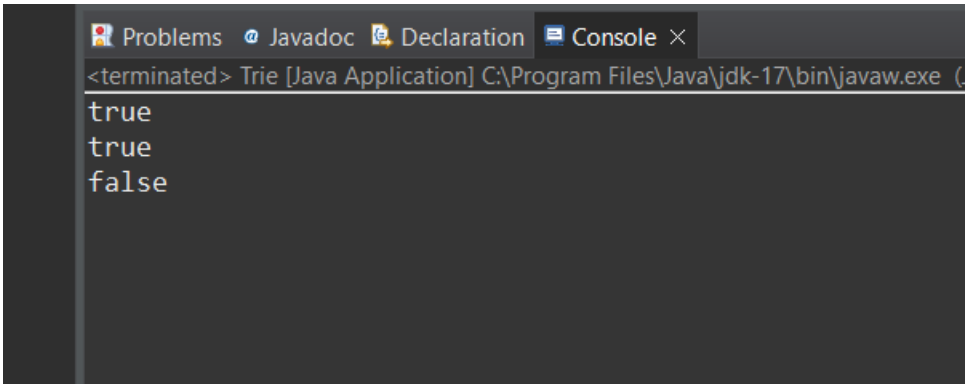
Ans:

```
package AssignmentDay7_8;


class TrieNode {


 TrieNode[] children = new TrieNode[26];

 boolean isEndOfWord;


 public TrieNode() {

    isEndOfWord = false;

    for (int i = 0; i < 26; i++) {

        children[i] = null;
```

```
    }
  }
}



public class Trie {
  private TrieNode root;


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

  public void insert(String word) {
    TrieNode currentNode = root;
    for (int i = 0; i < word.length(); i++) {
      int index = word.charAt(i) - 'a';
      if (currentNode.children[index] == null) {
        currentNode.children[index] = new TrieNode();
      }
      currentNode = currentNode.children[index];
    }
    currentNode.isEndOfWord = true;
  }


  public boolean startsWith(String prefix) {
    TrieNode currentNode = root;
    for (int i = 0; i < prefix.length(); i++) {
```

```
        int index = prefix.charAt(i) - 'a';
        if (currentNode.children[index] == null) {
            return false;
        }
        currentNode = currentNode.children[index];
    }
    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"));
    System.out.println(trie.startsWith("ban"));
    System.out.println(trie.startsWith("cat"));
}
}
```

Output:



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:

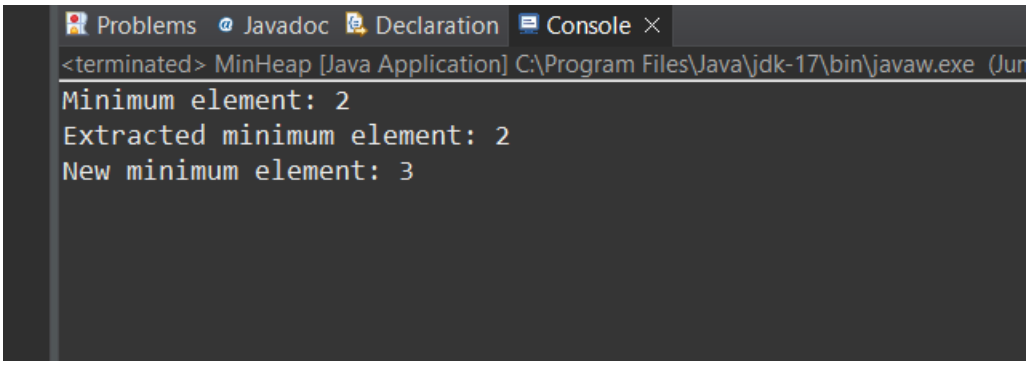
```
package AssignmentDay7_8;


import java.util.ArrayList;


public class MinHeap {
   private ArrayList<Integer> heap;



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


   private int parent(int i) {
      return (i - 1) / 2;
   }
   private int leftChild(int i) {
      return 2 * i + 1;
   }


   private int rightChild(int i) {
      return 2 * i + 2;
   }


   private void swap(int i, int j) {
      int temp = heap.get(i);
```

```
    heap.set(i, heap.get(j));
    heap.set(j, temp);
}

private void siftUp(int i) {
    while (i > 0 && heap.get(parent(i)) > heap.get(i)) {
        swap(i, parent(i));
        i = parent(i);
    }
}

private void siftDown(int i) {
    int minIndex = i;
    int left = leftChild(i);
    if (left < heap.size() && heap.get(left) < heap.get(minIndex)) {
        minIndex = left;
    }
    int right = rightChild(i);
    if (right < heap.size() && heap.get(right) < heap.get(minIndex)) {
        minIndex = right;
    }
    if (i != minIndex) {
        swap(i, minIndex);
        siftDown(minIndex);
    }
}

public void insert(int key) {
    heap.add(key);
```

```
    siftUp(heap.size() - 1);
}

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

public int extractMin() {
    if (heap.isEmpty()) {
        throw new IllegalStateException("Heap is empty");
    }
    int result = heap.get(0);
    heap.set(0, heap.get(heap.size() - 1));
    heap.remove(heap.size() - 1);
    if (!heap.isEmpty()) {
        siftDown(0);
    }
    return result;
}

public static void main(String[] args) {
    MinHeap minHeap = new MinHeap();
    minHeap.insert(3);
    minHeap.insert(2);
    minHeap.insert(15);
```

```
        minHeap.insert(5);
        minHeap.insert(4);
        minHeap.insert(45);

        System.out.println("Minimum element: " + minHeap.getMin());
        System.out.println("Extracted minimum element: " + minHeap.extractMin());
        System.out.println("New minimum element: " + minHeap.getMin());
    }
}
```

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.

Ans:

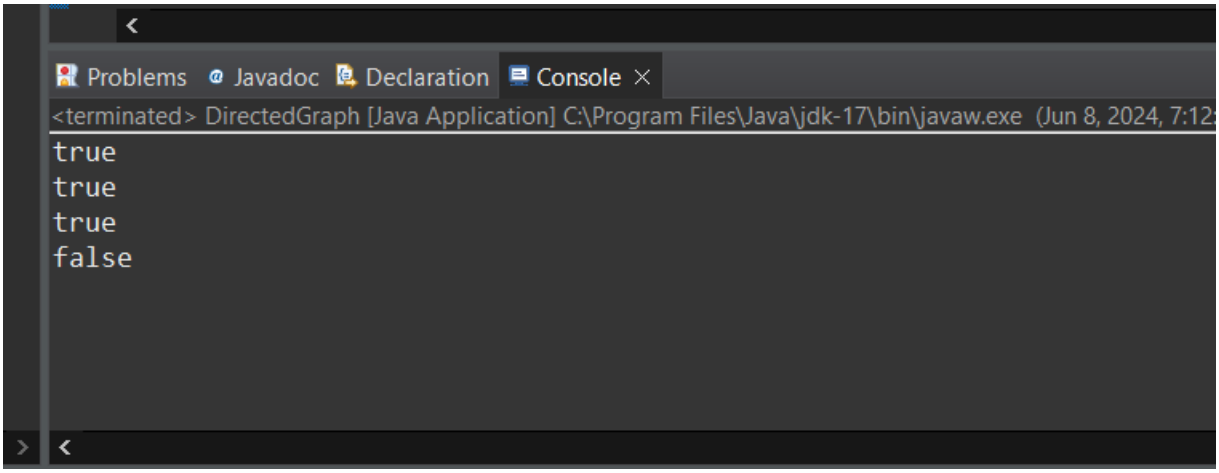
```
package AssignmentDay7_8;


import java.util.*;


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

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

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

public boolean addEdge(int from, int to) {
    adjList.putIfAbsent(from, new ArrayList<>());
    adjList.putIfAbsent(to, new ArrayList<>());
    adjList.get(from).add(to);

    if (isCyclic()) {
        adjList.get(from).remove((Integer) to);
        return false;
    }
    return true;
}

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

    for (Integer node : adjList.keySet()) {
        if (isCyclicUtil(node, visited, recStack)) {
            return true;
        }
    }
```

```
    return false;
}

private boolean isCyclicUtil(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 : adjList.get(node)) {
        if (isCyclicUtil(neighbor, visited, recStack)) {
            return true;
        }
    }

    recStack.remove(node);
    return false;
}

public static void main(String[] args) {
    DirectedGraph graph = new DirectedGraph();
    graph.addNode(0);
    graph.addNode(1);
    graph.addNode(2);
```

```
        graph.addNode(3);

        System.out.println(graph.addEdge(0, 1));
        System.out.println(graph.addEdge(1, 2));
        System.out.println(graph.addEdge(2, 3));
        System.out.println(graph.addEdge(3, 1));
    }
}
```

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.

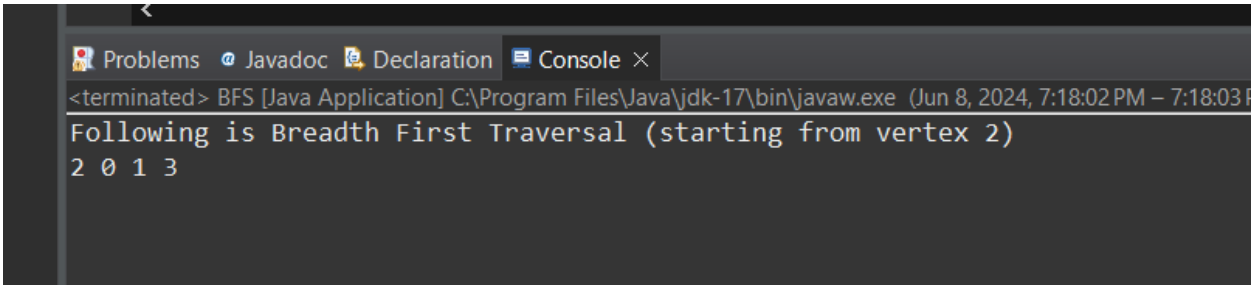
Ans:

```
package AssignmentDay7_8;


import java.util.*;


public class BFS {
    private int V;
```

```
private LinkedList<Integer> adj[];



BFS(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);
}

void BFS(int s) {
   boolean visited[] = new boolean[V];

   LinkedList<Integer> queue = new LinkedList<>();

   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[]) {
      BFS g = new BFS(4);

      g.addEdge(0, 1);
      g.addEdge(0, 2);
      g.addEdge(1, 2);
      g.addEdge(2, 0);
      g.addEdge(2, 3);
      g.addEdge(3, 3);

      System.out.println("Following is Breadth First Traversal " +
                         "(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.

Ans:

```
package AssignmentDay7_8;


import java.util.*;


public class DFS {
   private int V;
   private LinkedList<Integer> adj[];


   // Constructor
   DFS(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);
}

void DFSUtil(int v, boolean visited[]) {
    visited[v] = true;
    System.out.print(v + " ");

    for (Integer n : adj[v]) {
        if (!visited[n])
            DFSUtil(n, visited);
    }
}

void DFS(int v) {
    boolean visited[] = new boolean[V];

    DFSUtil(v, visited);
}

public static void main(String args[]) {
    DFS g = new DFS(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);
```

```
        System.out.println("Following is Depth First Traversal " +
                           "(starting from vertex 2)");

        g.DFS(2);
    }
}
```

Output: