Data Structure and Operations (Tree,Heap,Graph,DFS,BFS): Day 7 & 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.

```
package wipro.com.assignment06;
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

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

class BinaryTree {

    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) {
        BinaryTree tree = new BinaryTree();

        // Example 1: Balanced Tree
        TreeNode root1 = new TreeNode(1);
        root1.left = new TreeNode(2);
        root1.right = new TreeNode(3);
        root1.left.left = new TreeNode(4);
        root1.left.right = new TreeNode(5);
        root1.right.right = new TreeNode(6);
```

```
        System.out.println("Is the tree balanced? " + tree.isBalanced(root1)); // true

         // Example 2: Unbalanced Tree
         TreeNode root2 = new TreeNode(1);
         root2.left = new TreeNode(2);
         root2.left.left = new TreeNode(3);
         root2.left.left.left = new TreeNode(4);

        System.out.println("Is the tree balanced? " + tree.isBalanced(root2)); //false
    }
}
```

Output:

Is the tree balanced? true
Is the tree balanced? 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.

```
package wipro.com.assignment06;
import java.util.HashMap;
import java.util.Map;
import java.util.HashMap;
import java.util.Map;

class TrieNode {
    Map<Character, TrieNode> children;
    boolean isEndOfWord;

    TrieNode() {
        children = new HashMap<>();
        isEndOfWord = false;
    }
}

class Trie {
    private TrieNode root;

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

    public void insert(String word) {
        TrieNode current = root;
        for (char ch : word.toCharArray()) {
```

```
            current.children.putIfAbsent(ch, new TrieNode());
            current = current.children.get(ch);
        }
        current.isEndOfWord = true;
    }

    public boolean startsWith(String prefix) {
        TrieNode current = root;
        for (char ch : prefix.toCharArray()) {
            if (!current.children.containsKey(ch)) {
                return false;
            }
            current = current.children.get(ch);
        }
        return true;
    }
}

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

        // Insert words into the Trie
        trie.insert("apple");
        trie.insert("app");
        trie.insert("apricot");
        trie.insert("banana");

        // Check if certain prefixes exist in the Trie
        System.out.println(trie.startsWith("app"));    // true
        System.out.println(trie.startsWith("appl"));   // true
        System.out.println(trie.startsWith("ban"));    // true
        System.out.println(trie.startsWith("cat"));    // false
    }
}
```

Output:

true
true
true
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.

```
package wipro.com.assignment06;
import java.util.Arrays;

public class MinHeap {
    private int[] heap;
    private int size;
    private int capacity;

    public MinHeap(int capacity) {
        this.capacity = capacity;
        this.size = 0;
        this.heap = new int[capacity];
    }

    public void insert(int value) {
        if (size == capacity) {
            // If heap is full, resize it
            resizeHeap();
        }

        // Insert the new element at the end
        size++;
        int index = size - 1;
        heap[index] = value;

        // Fix the min heap property if it's violated
        heapifyUp(index);
    }

    private void heapifyUp(int index) {
        int parent = (index - 1) / 2;
        while (index > 0 && heap[index] < heap[parent]) {
            // Swap the current element with its parent if it's smaller
            swap(index, parent);
            index = parent;
            parent = (index - 1) / 2;
        }
    }

    public int extractMin() {
        if (size == 0) {
            throw new IllegalStateException("Heap is empty");
        }

        // Extract the minimum element
        int min = heap[0];
```

```
        // Move the last element to the root
        heap[0] = heap[size - 1];
        size--;

        // Fix the min heap property if it's violated
        heapifyDown(0);

        return min;
    }

    private void heapifyDown(int index) {
        int smallest = index;
        int left = 2 * index + 1;
        int right = 2 * index + 2;

        if (left < size && heap[left] < heap[smallest]) {
            smallest = left;
        }

        if (right < size && heap[right] < heap[smallest]) {
            smallest = right;
        }

        if (smallest != index) {
            swap(index, smallest);
            heapifyDown(smallest);
        }
    }

    public int getMin() {
        if (size == 0) {
            throw new IllegalStateException("Heap is empty");
        }
        return heap[0];
    }

    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }

    private void resizeHeap() {
        capacity *= 2;
        heap = Arrays.copyOf(heap, capacity);
    }

    public static void main(String[] args) {
        MinHeap minHeap = new MinHeap(10);

        minHeap.insert(5);
        minHeap.insert(3);
        minHeap.insert(8);
        minHeap.insert(1);
        minHeap.insert(10);
```

```
        System.out.println("Minimum element: " + minHeap.getMin()); // Output: 1

        minHeap.extractMin(); // Remove the minimum element
        System.out.println("Minimum element after deletion: " + minHeap.getMin()); //
Output: 3
    }
}
```

Output:

Minimum element: 1
Minimum element after deletion: 3

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 wipro.com.assignment06;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

class Graph {
    private int V;
    private List<List<Integer>> adjList;

    public Graph(int vertices) {
        V = vertices;
        setAdjList(new ArrayList<>(V));
        for (int i = 0; i < V; i++) {
            getAdjList().add(new ArrayList<>());
        }
    }

    public void addEdge(int u, int v) {
        getAdjList().get(u).add(v);
        getAdjList().get(v).add(u);

        if (hasCycle(u, new HashSet<>(), -1)) {
            // If adding this edge creates a cycle, remove the edge
            getAdjList().get(u).remove(Integer.valueOf(v));
            getAdjList().get(v).remove(Integer.valueOf(u));
            System.out.println("Adding edge (" + u + ", " + v + ") creates a cycle.
Edge not added.");
        } else {
            System.out.println("Edge (" + u + ", " + v + ") added successfully.");
        }
```

```
    }

    private boolean hasCycle(int vertex, Set<Integer> visited, int parent) {
        visited.add(vertex);

        for (int neighbor : getAdjList().get(vertex)) {
            if (!visited.contains(neighbor)) {
                if (hasCycle(neighbor, visited, vertex)) {
                    return true;
                }
            } else if (neighbor != parent) {
                return true;
            }
        }
        return false;
    }

      public List<List<Integer>> getAdjList() {
            return adjList;
      }

      public void setAdjList(List<List<Integer>> adjList) {
            this.adjList = adjList;
      }
}

public class Main {
    public static void main(String[] args) {
        int V = 4;
        Graph graph = new Graph(V);

        // Add edges
        graph.addEdge(0, 1);
        graph.addEdge(1, 2);
        graph.addEdge(2, 3);
        graph.addEdge(3, 0); // Adding this edge creates a cycle

        // Print adjacency list
        System.out.println("Adjacency List:");
        for (int i = 0; i < V; i++) {
            System.out.print(i + " -> ");
            for (int neighbor : graph.getAdjList().get(i)) {
                System.out.print(neighbor + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
Edge (0, 1) added successfully.
Edge (1, 2) added successfully.
Edge (2, 3) added successfully.
Adding edge (3, 0) creates a cycle. Edge not added.
```

```
Adjacency List:
0 -> 1
1 -> 0 2
2 -> 1 3
3 -> 2
```

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.

```java
package wipro.com.assignment06;
import java.util.*;

class Graph {
    private int V;
    private List<List<Integer>> adjList;

    public Graph(int vertices) {
        V = vertices;
        adjList = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            adjList.add(new ArrayList<>());
        }
    }

    public void addEdge(int u, int v) {
        adjList.get(u).add(v);
        adjList.get(v).add(u);
    }

    public void BFS(int start) {
        boolean[] visited = new boolean[V];
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(start);
        visited[start] = true;

        System.out.println("BFS Traversal starting from node " + start + ":");

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

            for (int neighbor : adjList.get(current)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.offer(neighbor);
                }
            }
        }
        System.out.println();
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        int V = 5;
        Graph graph = new Graph(V);

        // Add edges
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);

        int startNode = 0;
        graph.BFS(startNode);
    }
}
```

Output:

BFS Traversal starting from node 0:
0 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.

```
package wipro.com.assignment06;
import java.util.*;

class Graph {
    private int V;
    private List<List<Integer>> adjList;

    public Graph(int vertices) {
        V = vertices;
        adjList = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            adjList.add(new ArrayList<>());
        }
    }

    public void addEdge(int u, int v) {
        adjList.get(u).add(v);
        adjList.get(v).add(u);
    }
```

```
    public void DFS(int start) {
        boolean[] visited = new boolean[V];
        System.out.println("DFS Traversal starting from node " + start + ":");
        recursiveDFS(start, visited);
        System.out.println();
    }

    private void recursiveDFS(int node, boolean[] visited) {
        visited[node] = true;
        System.out.print(node + " ");

        for (int neighbor : adjList.get(node)) {
            if (!visited[neighbor]) {
                recursiveDFS(neighbor, visited);
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        int V = 5;
        Graph graph = new Graph(V);

        // Add edges
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);

        int startNode = 0;
        graph.DFS(startNode);
    }
}
```

Output:

```
DFS Traversal starting from node 0:
0 1 3 2 4
```