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.

```
class TreeNode {

    int val;

    TreeNode left;

    TreeNode right;

    TreeNode(int x) { val = x; }

}


public class BalancedBinaryTree {


    public static boolean isBalanced(TreeNode root) {

        return checkHeightAndBalance(root).isBalanced;

    }


    private static HeightBalance checkHeightAndBalance(TreeNode node) {

        // Base case: an empty tree is balanced with height -1

        if (node == null) {

            return new HeightBalance(-1, true);

        }


        // Check left subtree

        HeightBalance leftResult = checkHeightAndBalance(node.left);

        if (!leftResult.isBalanced) {

            return new HeightBalance(-1, false);
```

```
        }

        // Check right subtree
        HeightBalance rightResult = checkHeightAndBalance(node.right);
        if (!rightResult.isBalanced) {
            return new HeightBalance(-1, false);
        }

        // Calculate the height and balance status of the current node
        int height = Math.max(leftResult.height, rightResult.height) + 1;
        boolean isBalanced = Math.abs(leftResult.height - rightResult.height) <= 1;

        return new HeightBalance(height, isBalanced);
    }

    private static class HeightBalance {
        int height;
        boolean isBalanced;

        HeightBalance(int height, boolean isBalanced) {
            this.height = height;
            this.isBalanced = isBalanced;
        }
    }
```

```
    public static void main(String[] args) {
        // Example usage:
        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.left.left.left = new TreeNode(8);

        System.out.println("Is the binary tree balanced? " + isBalanced(root)); // Output: 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.

```
import java.util.HashMap;
import java.util.Map;

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

    public TrieNode() {
        children = new HashMap<>();
```

```java
            isEndOfWord = false;
        }
}

public class Trie {
        private final TrieNode root;

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

        // Method to insert a word into the Trie
        public void insert(String word) {
                TrieNode node = root;
                for (char ch : word.toCharArray()) {
                        node.children.putIfAbsent(ch, new TrieNode());
                        node = node.children.get(ch);
                }
                node.isEndOfWord = true;
        }

        // Method to check if a given string is a prefix of any word in the Trie
        public boolean startsWith(String prefix) {
                TrieNode node = root;
                for (char ch : prefix.toCharArray()) {
```

```java
                node = node.children.get(ch);
                if (node == null) {
                    return false;
                }
            }
            return true;
        }

        public static void main(String[] args) {
            Trie trie = new Trie();
            trie.insert("apple");
            trie.insert("app");
            trie.insert("application");

            System.out.println(trie.startsWith("app"));    // Output: true
            System.out.println(trie.startsWith("appl")); // Output: true
            System.out.println(trie.startsWith("banana")); // Output: 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.

```csharp
public class MinHeap {
        private int[] heap;
        private int size;
```

```
private int capacity;

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

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

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

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

// Swap two elements at indices i and j in the heap
private void swap(int i, int j) {
```

```
        int temp = heap[i];

        heap[i] = heap[j];

        heap[j] = temp;

    }


    // Heapify up (used after insertion)

    private void heapifyUp(int i) {

        while (i > 0 && heap[i] < heap[parent(i)]) {

            swap(i, parent(i));

            i = parent(i);

        }

    }


    // Heapify down (used after deletion)

    private void heapifyDown(int i) {

        int minIndex = i;

        int left = leftChild(i);

        int right = rightChild(i);


        if (left < size && heap[left] < heap[minIndex])

            minIndex = left;


        if (right < size && heap[right] < heap[minIndex])

            minIndex = right;
```

```
        if (minIndex != i) {
            swap(i, minIndex);
            heapifyDown(minIndex);
        }
    }

    // Insert an element into the heap
    public void insert(int value) {
        if (size == capacity)
            throw new IllegalStateException("Heap is full");

        heap[size] = value;
        size++;
        heapifyUp(size - 1);
    }

    // Delete the minimum element from the heap
    public int deleteMin() {
        if (size == 0)
            throw new IllegalStateException("Heap is empty");

        int min = heap[0];
        heap[0] = heap[size - 1];
        size--;
        heapifyDown(0);
```

```
        return min;
    }

    // Get the minimum element from the heap
    public int getMin() {
        if (size == 0)
            throw new IllegalStateException("Heap is empty");

        return heap[0];
    }

    public static void main(String[] args) {
        MinHeap minHeap = new MinHeap(10);
        minHeap.insert(10);
        minHeap.insert(20);
        minHeap.insert(5);

        System.out.println("Minimum element in heap: " + minHeap.getMin()); // Output: 5

        minHeap.deleteMin();

        System.out.println("Minimum element in heap after deletion: " + minHeap.getMin()); //
Output: 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.

```
import java.util.*;


public class Graph {

    private Map<Character, List<Character>> adjacencyList;


    public Graph() {

        adjacencyList = new HashMap<>();

    }


    public void addEdge(char source, char destination) {

        adjacencyList.computeIfAbsent(source, k -> new ArrayList<>()).add(destination);

    }


    public boolean hasCycleAfterAddingEdge(char u, char v) {

        addEdge(u, v); // Add the edge first

        Set<Character> visited = new HashSet<>();

        Set<Character> stack = new HashSet<>();


        for (Character node : adjacencyList.keySet()) {

            if (hasCycle(node, visited, stack)) {

                // Remove the added edge if it creates a cycle

                adjacencyList.get(u).remove((Character) v);

                return true;
```

```
            }
        }
        return false;
    }

    private boolean hasCycle(Character node, Set<Character> visited, Set<Character> stack) {
        if (stack.contains(node)) {
            return true;
        }
        if (visited.contains(node)) {
            return false;
        }

        visited.add(node);
        stack.add(node);

        List<Character> neighbors = adjacencyList.getOrDefault(node, new ArrayList<>());
        for (Character neighbor : neighbors) {
            if (hasCycle(neighbor, visited, stack)) {
                return true;
            }
        }

        stack.remove(node);
        return false;
```

```
    }

    public static void main(String[] args) {
        Graph graph = new Graph();
        graph.addEdge('A', 'B');
        graph.addEdge('B', 'C');
        graph.addEdge('C', 'A');

        char u = 'C', v = 'B';
        if (!graph.hasCycleAfterAddingEdge(u, v)) {
            graph.addEdge(u, v);
            System.out.println("Edge (" + u + ", " + v + ") added successfully.");
        } else {
            System.out.println("Adding edge (" + u + ", " + v + ") creates a cycle. Edge not added.");
        }
    }
}
```

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.

```
import java.util.*;

public class Graph {
    private Map<Character, List<Character>> adjacencyList;

    public Graph() {
        adjacencyList = new HashMap<>();
```

```
    }

    public void addEdge(char source, char destination) {
        adjacencyList.computeIfAbsent(source, k -> new ArrayList<>()).add(destination);
        adjacencyList.computeIfAbsent(destination, k -> new ArrayList<>()).add(source); // For
undirected graph
    }

    public void bfs(char startNode) {
        Set<Character> visited = new HashSet<>();
        Queue<Character> queue = new LinkedList<>();
        queue.offer(startNode);

        while (!queue.isEmpty()) {
            char node = queue.poll();
            if (!visited.contains(node)) {
                System.out.println(node);
                visited.add(node);
                List<Character> neighbors = adjacencyList.get(node);
                if (neighbors != null) {
                    for (char neighbor : neighbors) {
                        if (!visited.contains(neighbor)) {
                            queue.offer(neighbor);
                        }
                    }
                }
```

```
            }
        }
    }

    public static void main(String[] args) {
        Graph graph = new Graph();
        graph.addEdge('A', 'B');
        graph.addEdge('A', 'C');
        graph.addEdge('B', 'D');
        graph.addEdge('B', 'E');
        graph.addEdge('C', 'F');
        graph.addEdge('E', 'F');

        char startNode = 'A';
        System.out.println("BFS traversal starting from node " + startNode);
        graph.bfs(startNode);
    }
}
```

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.

```
import java.util.*;

public class Graph {
    private Map<Character, List<Character>> adjacencyList;
```

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

    public void addEdge(char source, char destination) {
        adjacencyList.computeIfAbsent(source, k -> new ArrayList<>()).add(destination);
        adjacencyList.computeIfAbsent(destination, k -> new ArrayList<>()).add(source); // For
undirected graph
    }

    public void dfs(char startNode) {
        Set<Character> visited = new HashSet<>();
        dfsRecursive(startNode, visited);
    }

    private void dfsRecursive(char node, Set<Character> visited) {
        visited.add(node);
        System.out.println(node);

        List<Character> neighbors = adjacencyList.get(node);
        if (neighbors != null) {
            for (char neighbor : neighbors) {
                if (!visited.contains(neighbor)) {
                    dfsRecursive(neighbor, visited);
                }
```

```
            }
        }
    }

    public static void main(String[] args) {
        Graph graph = new Graph();
        graph.addEdge('A', 'B');
        graph.addEdge('A', 'C');
        graph.addEdge('B', 'D');
        graph.addEdge('B', 'E');
        graph.addEdge('C', 'F');
        graph.addEdge('E', 'F');

        char startNode = 'A';
        System.out.println("DFS traversal starting from node " + startNode);
        graph.dfs(startNode);
    }
}
```