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.

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

public class Solution {
 public boolean isBalanced(TreeNode root) {
  return getHeight(root) != -1;
 }

 private int getHeight(TreeNode node) {
  if (node == null) {
   return 0;
  }

  int leftHeight = getHeight(node.left);
  if (leftHeight == -1) {
   return -1;
  }
```

```
    int rightHeight = getHeight(node.right);
    if (rightHeight == -1) {
      return -1;
    }

    int heightDiff = Math.abs(leftHeight - rightHeight);
    if (heightDiff > 1) {
      return -1;
    }

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

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.

```
class TrieNode {
  public char val;
  public boolean isWord;
  public TrieNode[] children;

  public TrieNode(char val) {
    this.val = val;
    this.isWord = false;
    this.children = new TrieNode[26];
  }
}
```

```
class Trie {
  private TrieNode root;

  public Trie() {
    this.root = new TrieNode(' ');
  }
  public void insert(String word) {
    TrieNode current = root;
    for (char ch : word.toCharArray()) {
      int index = ch - 'a';
      if (current.children[index] == null) {
        current.children[index] = new TrieNode(ch);
      }
      current = current.children[index];
    }
    current.isWord = true;
  }
  public boolean startsWith(String prefix) {
    TrieNode current = root;
    for (char ch : prefix.toCharArray()) {
      int index = ch - 'a';
      if (current.children[index] == null) {
        return false;
      }
      current = current.children[index];
    }
    return true;
  }
}
```

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.

```
public class MinHeap {
 private int[] heap;
 private int size;
 private static final int DEFAULT_CAPACITY = 10;

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

 public MinHeap(int capacity) {
  this.heap = new int[capacity];
  this.size = 0;
 }
 private int getLeftChildIndex(int parentIndex) {
  return 2 * parentIndex + 1;
 }
 private int getRightChildIndex(int parentIndex) {
  return 2 * parentIndex + 2;
 }
 private int getParentIndex(int childIndex) {
  return (childIndex - 1) / 2;
 }
```

```java
private boolean hasLeftChild(int index) {
  return getLeftChildIndex(index) < size;
 }

private boolean hasRightChild(int index) {
  return getRightChildIndex(index) < size;
 }

 private boolean isLeaf(int index) {
  return !hasLeftChild(index);
 }

  private void swap(int index1, int index2) {
  int temp = heap[index1];
  heap[index1] = heap[index2];
  heap[index2] = temp;
 }

  private void heapifyUp(int index) {
  int parentIndex = getParentIndex(index);
  while (index > 0 && heap[parentIndex] > heap[index]) {
    swap(parentIndex, index);
    index = parentIndex;
  }
 }

  private void heapifyDown(int index) {
  while (!isLeaf(index)) {
    int smallerChildIndex = getLeftChildIndex(index);
```

```
    if (hasRightChild(index) && heap[smallerChildIndex] > heap[getRightChildIndex(index)]) {
      smallerChildIndex = getRightChildIndex(index);
    }
    if (heap[index] <= heap[smallerChildIndex]) {
      break;
    }
    swap(index, smallerChildIndex);
    index = smallerChildIndex;
  }
}

 public void insert(int element) {
  if (size == heap.length) {
    int[] newHeap = new int[2 * heap.length];
    System.arraycopy(heap, 0, newHeap, 0, heap.length);
    heap = newHeap;
  }
  heap[size] = element;
  size++;
  heapifyUp(size - 1);
}

public int peek() {
  if (isEmpty()) {
    throw new NoSuchElementException("Heap is empty");
  }
  return heap[0];
}
```

```
  public boolean isEmpty() {
    return size == 0;
  }

  public int extractMin() {
    if (isEmpty()) {
      throw new NoSuchElementException("Heap is empty");
    }
    int min = heap[0];
    heap[0] = heap[size - 1];
    size--;
    heapifyDown(0);
    return min;
  }
}
```

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 int V;
  private List<Integer>[] adjList;

  public Graph(int V) {
    this.V = V;
    adjList = new ArrayList[V];
```

```java
    for (int i = 0; i < V; i++) {
      adjList[i] = new ArrayList<>();
    }
  }

  public void addEdge(int u, int v) {
    adjList[u].add(v);
  }
  private boolean isCyclicUtil(int v, boolean[] visited, boolean[] recStack) {
    if (visited[v]) return false;
    if (recStack[v]) return true;

    visited[v] = true;
    recStack[v] = true;

    for (int neighbor : adjList[v]) {
      if (isCyclicUtil(neighbor, visited, recStack)) {
        return true;
      }
    }

    recStack[v] = false;
    return false;
  }
  public boolean isCyclic(int u, int v) {
    boolean[] visited = new boolean[V];
    boolean[] recStack = new boolean[V];

    addEdge(u, v); // Add the edge temporarily
    if (isCyclicUtil(0, visited, recStack)) {
```

```
      adjList[u].remove(adjList[u].indexOf(v));
      return true;
    }
    adjList[u].remove(adjList[u].indexOf(v));
    return false;
  }
}
```

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 int V;
  private List<Integer>[] adjList;

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

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

```
  public void BFS(int startVertex) {
    boolean[] visited = new boolean[V];

    Queue<Integer> queue = new LinkedList<>();
    queue.add(startVertex);
    visited[startVertex] = true;

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

      for (int neighbor : adjList[currentVertex]) {
        if (!visited[neighbor]) {
          queue.add(neighbor);
          visited[neighbor] = true;
        }
      }
    }
  }
}
```

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.List;

public class Graph {
  private int V;
  private List<Integer>[] adjList;
```

```
  public Graph(int V) {
    this.V = V;
    adjList = new ArrayList[V];
    for (int i = 0; i < V; i++) {
      adjList[i] = new ArrayList<>();
    }
  }

  public void addEdge(int u, int v) {
    adjList[u].add(v);
    adjList[v].add(u);
  }
  public void DFSUtil(int v, boolean[] visited) {
    visited[v] = true;
    System.out.print(v + " ");

    for (int neighbor : adjList[v]) {
      if (!visited[neighbor]) {
        DFSUtil(neighbor, visited);
      }
    }
  }
  public void DFS(int startVertex) {
    boolean[] visited = new boolean[V];
    DFSUtil(startVertex, visited);
  }
}
```