Data Structure and Operations: Day 9 & 10

(Dijkstra's Shortest Path Finder, Kruskal's Algorithm for, Union-Find for Cycle):

Task 1: Dijkstra's Shortest Path Finder

Code Dijkstra's algorithm to find the shortest path from a start node to every other node in a weighted graph with positive weights.

```java
package wipro.com.assignment07;
import java.util.*;

class Dijkstra
{
    static class Node implements Comparable<Node> {
        int vertex;
        int distance;

        Node(int vertex, int distance) {
            this.vertex = vertex;
            this.distance = distance;
        }

        @Override
        public int compareTo(Node other) {
            return Integer.compare(this.distance, other.distance);
        }
    }

    public static void dijkstra(int[][] graph, int start)
    {
        int n = graph.length;
        int[] distances = new int[n];
        boolean[] visited = new boolean[n];

        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[start] = 0;

        PriorityQueue<Node> priorityQueue = new PriorityQueue<>();
        priorityQueue.add(new Node(start, 0));

        while (!priorityQueue.isEmpty()) {
            Node current = priorityQueue.poll();
            int u = current.vertex;

            if (visited[u]) continue;
            visited[u] = true;
            for (int v = 0; v < n; v++) {
                if (graph[u][v] > 0 && !visited[v]) {
                    int newDist = distances[u] + graph[u][v];
                    if (newDist < distances[v]) {
                        distances[v] = newDist;
                        priorityQueue.add(new Node(v, newDist));
                    }
                }
            }
        }

        // Output the shortest distances from the start node to each other node
        System.out.println("Vertex\tDistance from Start");
        for (int i = 0; i < n; i++) {
            System.out.println(i + "\t" + distances[i]);
        }
    }

    public static void main(String[] args) {
        // Example graph represented as an adjacency matrix
        int[][] graph = {
            {0, 10, 20, 0, 0, 0},
            {10, 0, 0, 50, 10, 0},
            {20, 0, 0, 20, 33, 0},
            {0, 50, 20, 0, 20, 2},
            {0, 10, 33, 20, 0, 1},
            {0, 0, 0, 2, 1, 0}
        };

        int startNode = 0;
        dijkstra(graph, startNode);
    }
}
```

Output:

```
Vertex  Distance from Start
0       0
1       10
2       20
3       23
4       20
5       21
```

Task 2: Kruskal’s Algorithm for MST

Implement Kruskal’s algorithm to find the minimum spanning tree of a given connected, undirected graph with non-negative edge weights.

```java
package wipro.com.assignment07;
import java.util.*;

class Kruskal {
    static class Edge implements Comparable<Edge> {
        int src, dest, weight;

        Edge(int src, int dest, int weight) {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }

        @Override
        public int compareTo(Edge other) {
            return Integer.compare(this.weight, other.weight);
        }
    }

    static class Subset {
        int parent, rank;

        Subset(int parent, int rank) {
            this.parent = parent;
            this.rank = rank;
        }
    }

    int V, E; // Number of vertices and edges
    Edge[] edges; // Array of edges

    Kruskal(int v, int e) {
        V = v;
        E = e;
        edges = new Edge[E];
    }

    // Find set of an element i (uses path compression)
    int find(Subset[] subsets, int i) {
        if (subsets[i].parent != i)
            subsets[i].parent = find(subsets, subsets[i].parent);
        return subsets[i].parent;
    }

    // Union of two sets x and y (uses union by rank)
    void union(Subset[] subsets, int x, int y) {
        int rootX = find(subsets, x);
        int rootY = find(subsets, y);

        if (subsets[rootX].rank < subsets[rootY].rank) {
            subsets[rootX].parent = rootY;
        } else if (subsets[rootX].rank > subsets[rootY].rank) {
            subsets[rootY].parent = rootX;
        } else {
            subsets[rootY].parent = rootX;
            subsets[rootX].rank++;
        }
    }

    // Main function to construct MST using Kruskal's algorithm
    void kruskalMST() {
        // Step 1: Sort all the edges in non-decreasing order of their weight
        Arrays.sort(edges);

        // Allocate memory for creating V subsets
        Subset[] subsets = new Subset[V];
        for (int v = 0; v < V; ++v) {
            subsets[v] = new Subset(v, 0);
        }

        // Resulting MST
        ArrayList<Edge> result = new ArrayList<>();

        for (Edge edge : edges) {
            int x = find(subsets, edge.src);
            int y = find(subsets, edge.dest);

            // If including this edge does not cause a cycle
            if (x != y) {
                result.add(edge);
                union(subsets, x, y);
            }
        }

        // Print the contents of result[] to display the MST
        System.out.println("Following are the edges in the constructed MST");
        for (Edge edge : result) {
            System.out.println(edge.src + " -- " + edge.dest + " == " + edge.weight);
        }
    }

    public static void main(String[] args)
    {
        int V = 4; // Number of vertices in graph
        int E = 5; // Number of edges in graph
        Kruskal graph = new Kruskal(V, E);

        // add edge 0-1
        graph.edges[0] = new Edge(0, 1, 10);
        // add edge 0-2
        graph.edges[1] = new Edge(0, 2, 6);
        // add edge 0-3
        graph.edges[2] = new Edge(0, 3, 5);
        // add edge 1-3
        graph.edges[3] = new Edge(1, 3, 15);
        // add edge 2-3
        graph.edges[4] = new Edge(2, 3, 4);
        graph.kruskalMST();
    }
}
```

Output:

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

Task 3: Union-Find for Cycle Detection

Write a Union-Find data structure with path compression. Use this data structure to detect a cycle in an undirected graph.

Union-Find Data Structure with Path Compression:

```
package wipro.com.assignment07;
import java.util.*;
class UnionFind
{
    private int[] parent;
    private int[] rank;

    // Constructor
    public UnionFind(int size) {
        parent = new int[size];
        rank = new int[size];
        for (int i = 0; i < size; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    // Find with path compression
    public int find(int u) {
        if (parent[u] != u) {
            parent[u] = find(parent[u]); // Path compression
        }
        return parent[u];
    }

    // Union by rank
    public void union(int u, int v)
   {
        int rootU = find(u);
        int rootV = find(v);

        if (rootU != rootV) {
            if (rank[rootU] < rank[rootV]) {
                parent[rootU] = rootV;
            } else if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            } else {
                parent[rootV] = rootU;
                rank[rootU]++;
            }
        }
    }
}
```

Cycle Detection in an Undirected Graph:

```
package wipro.com.assignment07;
class Graph
{
    private int V, E; // Number of vertices and edges
    private Edge[] edges; // Array of edges

    static class Edge {
        int src, dest;
    }

    // Constructor
    public Graph(int v, int e) {
        V = v;
        E = e;
        edges = new Edge[E];
        for (int i = 0; i < e; i++) {
            edges[i] = new Edge();
        }
    }

    // Function to detect cycle using Union-Find
    public boolean isCycle() {
        UnionFind unionFind = new UnionFind(V);

        for (int i = 0; i < E; i++) {
            int x = unionFind.find(edges[i].src);
            int y = unionFind.find(edges[i].dest);

            if (x == y) {
                return true; // Cycle detected
            }

            unionFind.union(x, y);
        }
        return false; // No cycle detected
    }

    public static void main(String[] args)
{
        int V = 3, E = 3;
        Graph graph = new Graph(V, E);

        // Adding edge 0-1
        graph.edges[0].src = 0;
        graph.edges[0].dest = 1;

        // Adding edge 1-2
        graph.edges[1].src = 1;
        graph.edges[1].dest = 2;

        // Adding edge 0-2
        graph.edges[2].src = 0;
        graph.edges[2].dest = 2;

        if (graph.isCycle()) {
            System.out.println("Graph contains cycle");
        } else {
            System.out.println("Graph doesn't contain cycle");
        }
    }
}
```

Output:

```
Graph contains cycle
```