Prim's Algorithm and Kruskal Algorithm Implementation

Prim's Algorithm for Minimum Spanning Tree (MST):

```
// A Java program for Prim's Minimum Spanning Tree (MST)
// algorithm. The program is for adjacency matrix (not potimized)
// representation of the graph

import java.io.*;
import java.lang.*;
import java.util.*;

class MST {

    // Number of vertices in the graph
    private static final int V = 5;

    // A utility function to find the vertex with minimum
    // key value, from the set of vertices not yet included
    // in MST
    int minKey(int key[], Boolean mstSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index = -1;

        for (int v = 0; v < V; v++)
            if (mstSet[v] == false && key[v] < min) {
                min = key[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed MST
    // stored in parent[]
    void printMST(int parent[], int graph[][])
    {
        System.out.println("Edge \tWeight");
        for (int i = 1; i < V; i++)
            System.out.println(parent[i] + " - " + i + "\t"
                               + graph[i][parent[i]]);
    }

    // Function to construct and print MST for a graph
    // represented using adjacency matrix representation
    void primMST(int graph[][])
    {
        // Array to store constructed MST
        int parent[] = new int[V];
```

```
// Key values used to pick minimum weight edge in
// cut
int key[] = new int[V];

// To represent set of vertices included in MST
Boolean mstSet[] = new Boolean[V];

// Initialize all keys as INFINITE
for (int i = 0; i < V; i++) {
   key[i] = Integer.MAX_VALUE;
   mstSet[i] = false;
}

// Always include first 1st vertex in MST.
// Make key 0 so that this vertex is
// picked as first vertex
key[0] = 0;

// First node is always root of MST
parent[0] = -1;

// The MST will have V vertices
for (int count = 0; count < V - 1; count++) {

   // Pick the minimum key vertex from the set of
   // vertices not yet included in MST
   int u = minKey(key, mstSet);

   // Add the picked vertex to the MST Set
   mstSet[u] = true;

   // Update key value and parent index of the
   // adjacent vertices of the picked vertex.
   // Consider only those vertices which are not
   // yet included in MST
   for (int v = 0; v < V; v++)

      // graph[u][v] is non zero only for adjacent
      // vertices of m mstSet[v] is false for
      // vertices not yet included in MST Update
      // the key only if graph[u][v] is smaller
      // than key[v]
      if (graph[u][v] != 0 && mstSet[v] == false
         && graph[u][v] < key[v]) {
         parent[v] = u;
         key[v] = graph[u][v];
      }
}

// Print the constructed MST
```

```
        printMST(parent, graph);
    }

    public static void main(String[] args)
    {
        MST t = new MST();
        int graph[][] = new int[][] { { 0, 2, 0, 6, 0 },
                                      { 2, 0, 3, 8, 5 },
                                      { 0, 3, 0, 0, 7 },
                                      { 6, 8, 0, 0, 9 },
                                      { 0, 5, 7, 9, 0 } };

        // Print the solution
        t.primMST(graph);
    }
}
```

Optimized Implementation using Adjacency List Representation (of Graph) and Priority Queue

```
// A Java program for Prim's Minimum Spanning Tree (MST)
// algorithm. The program is for adjacency list
// representation of the graph

import java.io.*;
import java.util.*;

// Class to form pair
class Pair implements Comparable<Pair>
{
    int v;
    int wt;
    Pair(int v,int wt)
    {
        this.v=v;
        this.wt=wt;
    }
    public int compareTo(Pair that)
    {
        return this.wt-that.wt;
    }
}

class GFG {

// Function of spanning tree
static int spanningTree(int V, int E, int edges[][])
    {
         ArrayList<ArrayList<Pair>> adj=new ArrayList<>();
```

```
        for(int i=0;i<V;i++)
        {
          adj.add(new ArrayList<Pair>());
        }
        for(int i=0;i<edges.length;i++)
        {
          int u=edges[i][0];
          int v=edges[i][1];
          int wt=edges[i][2];
          adj.get(u).add(new Pair(v,wt));
          adj.get(v).add(new Pair(u,wt));
        }
        PriorityQueue<Pair> pq = new PriorityQueue<Pair>();
        pq.add(new Pair(0,0));
        int[] vis=new int[V];
        int s=0;
        while(!pq.isEmpty())
        {
          Pair node=pq.poll();
          int v=node.v;
          int wt=node.wt;
          if(vis[v]==1)
          continue;

          s+=wt;
          vis[v]=1;
          for(Pair it:adj.get(v))
          {
            if(vis[it.v]==0)
            {
              pq.add(new Pair(it.v,it.wt));
            }
          }
        }
        return s;
    }

    // Driver code
    public static void main (String[] args) {
      int graph[][] = new int[][] {{0,1,5},
                                   {1,2,3},
                                   {0,2,1}};

      // Function call
      System.out.println(spanningTree(3,3,graph));
    }
}
```

Kruskal's Minimum Spanning Tree (MST) Algorithm

```
// Java program for Kruskal's algorithm

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class KruskalsMST {

  // Defines edge structure
  static class Edge {
    int src, dest, weight;

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

  // Defines subset element structure
  static class Subset {
    int parent, rank;

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

  // Starting point of program execution
  public static void main(String[] args)
  {
    int V = 4;
    List<Edge> graphEdges = new ArrayList<Edge>(
      List.of(new Edge(0, 1, 10), new Edge(0, 2, 6),
          new Edge(0, 3, 5), new Edge(1, 3, 15),
          new Edge(2, 3, 4)));

    // Sort the edges in non-decreasing order
    // (increasing with repetition allowed)
    graphEdges.sort(new Comparator<Edge>() {
      @Override public int compare(Edge o1, Edge o2)
      {
        return o1.weight - o2.weight;
      }
```

```java
    });

    kruskals(V, graphEdges);
  }

  // Function to find the MST
  private static void kruskals(int V, List<Edge> edges)
  {
    int j = 0;
    int noOfEdges = 0;

    // Allocate memory for creating V subsets
    Subset subsets[] = new Subset[V];

    // Allocate memory for results
    Edge results[] = new Edge[V];

    // Create V subsets with single elements
    for (int i = 0; i < V; i++) {
      subsets[i] = new Subset(i, 0);
    }

    // Number of edges to be taken is equal to V-1
    while (noOfEdges < V - 1) {

      // Pick the smallest edge. And increment
      // the index for next iteration
      Edge nextEdge = edges.get(j);
      int x = findRoot(subsets, nextEdge.src);
      int y = findRoot(subsets, nextEdge.dest);

      // If including this edge doesn't cause cycle,
      // include it in result and increment the index
      // of result for next edge
      if (x != y) {
        results[noOfEdges] = nextEdge;
        union(subsets, x, y);
        noOfEdges++;
      }

      j++;
    }

    // Print the contents of result[] to display the
    // built MST
    System.out.println(
      "Following are the edges of the constructed MST:");
    int minCost = 0;
    for (int i = 0; i < noOfEdges; i++) {
      System.out.println(results[i].src + " -- "
```

```java
                        + results[i].dest + " == "
                        + results[i].weight);
            minCost += results[i].weight;
        }
        System.out.println("Total cost of MST: " + minCost);
    }

    // Function to unite two disjoint sets
    private static void union(Subset[] subsets, int x,
                              int y)
    {
        int rootX = findRoot(subsets, x);
        int rootY = findRoot(subsets, y);

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

    // Function to find parent of a set
    private static int findRoot(Subset[] subsets, int i)
    {
        if (subsets[i].parent == i)
            return subsets[i].parent;

        subsets[i].parent
            = findRoot(subsets, subsets[i].parent);
        return subsets[i].parent;
    }
}
```