```
import java.util.HashMap;
import java.util.LinkedList;

import heap.Heap;

/**
 * Represents a weighted undirected graph with edges stored in adjacency
 * lists. Nodes are represented using integer id's.
 * Undirected edges are represented as opposite-direction pairs of directed
 * edges, one in each direction, where one lives in the source nodes.
 */
public class Graph {
    // maps node id's to their edge lists
    private HashMap<Integer, LinkedList<Edge>> nodes;

    /** Constructor: empty graph */
    public Graph() {
        nodes = new HashMap<Integer,LinkedList<Edge>>();
    }

    /** Add a node with the given id and no outgoing edges */
    public void addNode(int nodeId) {
        nodes.put(nodeId, new LinkedList<Edge>());
    }

    /** Returns whether the given node is in the graph */
    public boolean hasNode(int nodeId) {
        return nodes.containsKey(nodeId);
    }

    /** Returns the edge from node1 to node2, or null either node doesn't
     * exist or there's no edge between them. */
    public Edge getEdge(int node1, int node2) {
        if (!nodes.containsKey(node1) || !nodes.containsKey(node2)) {
            return null;
        }
        for (Edge e : nodes.get(node1)) {
            if (e.end == node2) {
                return e;
            }
        }
        return null;
    }

    /** Adds an edge between node1 and node2.
     * Pre: node1 and node2 are nodes in the graph. */
    public void addEdge(int node1, int node2, int weight) {
        nodes.get(node1).add(new Edge(node1, node2, weight));
        nodes.get(node2).add(new Edge(node2, node1, weight));
    }

    /** Get the list of edges leaving a node
     * Pre: nodeId is a node in the graph. */
    public LinkedList<Edge> getEdges(int nodeId) {
        return nodes.get(nodeId);
```

```
}

/** Returns the number of nodes in the graph. */
public int numNodes() {
    return nodes.size();
}

/** Returns a new Graph representing a minimum spanning tree of this
 * graph, computed using Prim's algorithm.*/
public Graph prim(int startNode) {
    Graph mst = new Graph();
    mst.addNode(startNode);

    Heap<Edge,Integer> S = new Heap<Edge,Integer>();
    for (Edge e : this.getEdges(startNode)) {
        S.add(e, e.weight);
    }

    //inv: mst is a tree with a subset of this graph's nodes and
    //     edges (u,v) is in S iff u is in mst and v is not
    while (mst.numNodes() < this.numNodes()) {
        Edge e = S.poll();
        mst.addNode(e.end);
        mst.addEdge(e.start, e.end, e.weight);
        for (Edge vw : this.getEdges(e.end)) {
            if (!mst.hasNode(vw.end)) {
                S.add(vw, vw.weight);
            }
        }
    }
    return mst;
}

/** An edge in the Graph */
public class Edge {
    public int start; // one node
    public int end; // the other node
    public int weight; // weight of the edge

    /** Constructor: initializes endpoints and weight */
    public Edge(int start, int end, int weight) {
        this.start = start;
        this.end = end;
        this.weight = weight;
    }
}
```