```java
package edu.sdsu.cs.datastructures;

import java.util.*;

public class DirectedGraph<V> implements IGraph<V> {

    TreeMap<V, Node> nodesMap;

    public DirectedGraph() {
        nodesMap = new TreeMap<>();
    }

    @Override
    public void add(V vertexName) {
        if (!this.contains(vertexName)) {
            nodesMap.put(vertexName, new Node(vertexName));
        }
    }

    @Override
    public void connect(V start, V destination) {
        Node startVertex = findNodeInnodesMap(start);
        Node endVertex = findNodeInnodesMap(destination);

        if (startVertex != null) {
            if (endVertex != null) {
                if (!startVertex.hasConnectionToNode(destination)) {
                    startVertex.nodes.add(endVertex);
                }
            } else {
                throw new NoSuchElementException("Destination vertex is not
presented");
            }
        } else {
            throw new NoSuchElementException("Starting vertex vertex is not
presented");
        }
    }

    @Override
    public void clear() {
        nodesMap.clear();
    }

    @Override
    public boolean contains(V label) {
        return nodesMap.containsKey(label);
    }

    @Override
    public void disconnect(V start, V destination) {
        Node node = findNodeInnodesMap(start);

        if (node == null)
            throw new NoSuchElementException("Start Node not present");

        boolean executionresult = node.deleteConnection(destination);
        if (executionresult == false)
            throw new NoSuchElementException("Connection not present");
    }

    @Override
    public boolean isConnected(V start, V destination) {
        Node first = findNodeInnodesMap(start);
        Node second = findNodeInnodesMap(destination);

        boolean executionresult = isAtLeatsOneNull(new Node[] { first, second
});
        if (executionresult)
            throw new NoSuchElementException();

        LinkedList<Node> tracker = new LinkedList<>();

        return hasConnection((V) first.value, (V) second.value, tracker);
    }

    private boolean hasConnection(V start, V destination, LinkedList<Node>
tracker) {
        Node first = findNodeInnodesMap(start);
        Node second = findNodeInnodesMap(destination);

        for (Node node : (LinkedList<Node>) first.nodes) {
            if (node.equals(second)) {
                return true;
            }
        }

        tracker.add(first);

        for (Node node : (LinkedList<Node>) first.nodes) {
            if (first.equals(node))
                continue;

            if (tracker.contains(node))
                continue;

            if (!first.equals(node))
                if (hasConnection((V) node.value, destination, tracker))
                    return true;
        }
        return false;
    }

    private boolean isAtLeatsOneNull(Node[] nodes) {
        for (Node node : nodes)
            if (node == null)
                return true;
        return false;
    }

    @Override
    public Iterable<V> neighbors(V vertexName) {
        if (findNodeInnodesMap(vertexName) == null)
```

```
            throw new NoSuchElementException("Neighbour Vertex");

        LinkedList<V> neighborNodes = new LinkedList<>();
        for (Node node : (LinkedList<Node>)
findNodeInnodesMap(vertexName).nodes)
            neighborNodes.add((V) node.value);

        return neighborNodes;
    }

    @Override
    public void remove(V vertexName) {
        if (nodesMap.containsKey(vertexName)) {
            for (Node node : nodesMap.values()) {
                node.deleteConnection(vertexName);
            }
            nodesMap.remove(vertexName);
        } else
            throw new NoSuchElementException("Vertex Name");
    }

    @Override
    public List<V> shortestPath(V start, V destination) {
        Node first = findNodeInnodesMap(start);
        Node second = findNodeInnodesMap(destination);
        if (first == null)
            throw new NoSuchElementException("shortestPath start does not
exist");
        if (second == null)
            throw new NoSuchElementException("shortestPath destination does
not exist");

        if (!isConnected(start, destination)) {
            System.out.println("Nodes are not connected");
            return null;
        }

        if (first.hasConnectionToNode(destination)) {
            LinkedList<V> path = new LinkedList<>();
            path.add(start);
            path.add(destination);
            return path;
        }

        return searchForPath(start, destination);
    }

    private List<V> searchForPath(V start, V destination) {
        Node first = findNodeInnodesMap(start);
        Node second = findNodeInnodesMap(destination);

        TreeMap<V, Node> unchecked = nodesMap;
        LinkedList<Node> checked = new LinkedList<>();

        first.shortestPath = 0;

        PriorityQueue<V> priority_q = new PriorityQueue<>();
        priority_q.add(start);
        while (!priority_q.isEmpty()) {
            Node present = findNodeInnodesMap(priority_q.poll());
            if (!unchecked.containsValue(present)) {
                continue;
            }
            for (Node edge : (LinkedList<Node>) present.nodes) {

                int calculatedDistance = present.shortestPath + 1;
                if (calculatedDistance < edge.shortestPath) {
                    edge.shortestPath = calculatedDistance;
                }
                priority_q.add((V) edge.value);

            }
            checked.add(present);
            unchecked.remove(present.value);
            boolean finished = true;
            for (Node node : unchecked.values()) {
                if (node.shortestPath != Integer.MAX_VALUE) {
                    finished = false;
                }
            }
            if (!unchecked.containsValue(second) || finished) {
                break;
            }
        }
        LinkedList<V> result = new LinkedList<>();
        Node last = checked.get(checked.size() - 1);
        for (int counter = checked.size() - 1; counter > 0; counter--) {
            if (last.shortestPath - 1 == checked.get(counter -
1).shortestPath) {
                result.add((V) last.value);
                last = checked.get(counter - 1);
            }
        }
        result.add(start);
        Collections.reverse(result);
        return result;
    }

    public int size() {
        return nodesMap.size();
    }

    @Override
    public Iterable<V> vertices() {
        LinkedList<V> labelNodes = new LinkedList<>();

        for (Node node : nodesMap.values()) {
            labelNodes.add((V) node.value);
        }

        return labelNodes;
    }

    @Override
```

```java
    public IGraph<V> connectedGraph(V origin) {
        Node originNode = findNodeInnodesMap(origin);
        if (originNode == null)
            throw new NoSuchElementException("Origin");

        LinkedList<Node> visitedNodes = new LinkedList<>();
        visitedNodes.add(originNode);
        connectedGraphHelper(originNode, visitedNodes);

        DirectedGraph<V> newGraph = new DirectedGraph<>();
        for (Node edge : visitedNodes)
            newGraph.add(edge);

        return newGraph;
    }

    private LinkedList<Node> connectedGraphHelper(Node origin,
LinkedList<Node> visited) {
        for (Node node : (LinkedList<Node>) origin.nodes) {
            if (!visited.contains(node)) {
                connectedGraphHelper(node, visited);
                visited.add(node);
            }
        }

        return visited;
    }

    private void add(Node node) {
        nodesMap.put((V) node.value, node);
    }

    private Node findNodeInnodesMap(V value) {
        return nodesMap.containsKey(value) ? nodesMap.get(value) : null;
    }

    private class Node<V> {
        public V value;
        public LinkedList<Node> nodes;
        public int shortestPath = Integer.MAX_VALUE;

        public Node(V value) {
            this.value = value;
            nodes = new LinkedList<>();
        }

        public boolean hasConnectionToNode(V destination) {
            for (Node node : nodes) {
                if (node.value.equals(destination))
                    return true;
            }
            return false;
        }

        public boolean deleteConnection(V nodeToBeRemoved) {
            for (Node node : nodes) {
                if (node.value.equals(nodeToBeRemoved)) {
                    nodes.remove(node);
                    return true;
                }
            }
            return false;
        }
    }
}
```