Design Analysis And Algorithms

Dijkstra's Shortest Path:

```
import java.util.Scanner;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.Collections;
import java.util.HashSet;
class BinaryMinHeap<T> {
   private Map<T, Integer> nodePosition;
   private List<Node> heap;
   private int maxSize;

   public BinaryMinHeap(int maxSize) {
      this.nodePosition = new HashMap<>();
      this.heap = new ArrayList<>(maxSize);
      this.maxSize = maxSize;
   }

   public void add(int weight, T key) {
      if (nodePosition.containsKey(key)) {
         decrease(key, weight);
      } else {
         Node node = new Node();
         node.weight = weight;
         node.key = key;
         heap.add(node);
         nodePosition.put(key, heap.size() - 1);
         heapifyUp(heap.size() - 1);
      }
   }

   public void decrease(T key, int newWeight) {
      int position = nodePosition.get(key);
      Node node = heap.get(position);
      node.weight = newWeight;
      heapifyUp(position);
   }

   public boolean containsData(T key) {
      return nodePosition.containsKey(key);
   }
```

```
public boolean empty() {
   return heap.isEmpty();
}

public Node extractMinNode() {
   Node minNode = heap.get(0);
   Node lastNode = heap.remove(heap.size() - 1);
   nodePosition.remove(minNode.key);
   if (!heap.isEmpty()) {
      heap.set(0, lastNode);
      nodePosition.put(lastNode.key, 0);
      heapifyDown(0);
   }
   return minNode;
}

public int getWeight(T key) {
   Integer position = nodePosition.get(key);
   if (position == null) {
      return Integer.MAX_VALUE;
   } else {
      return heap.get(position).weight;
   }
}

private void heapifyUp(int currentIndex) {
   int parentIndex = (currentIndex - 1) / 2;
   while (currentIndex > 0 && heap.get(currentIndex).weight < heap.get(parentIndex).weight) {
      swap(currentIndex, parentIndex);
      currentIndex = parentIndex;
      parentIndex = (currentIndex - 1) / 2;
   }
}

private void heapifyDown(int currentIndex) {
   int leftIndex = 2 * currentIndex + 1;
   int rightIndex = 2 * currentIndex + 2;
   int smallest = currentIndex;

   if (leftIndex < heap.size() && heap.get(leftIndex).weight < heap.get(smallest).weight) {
      smallest = leftIndex;
   }
   if (rightIndex < heap.size() && heap.get(rightIndex).weight < heap.get(smallest).weight) {
```

```
            smallest = rightIndex;
        }

        if (smallest != currentIndex) {
            swap(currentIndex, smallest);
            heapifyDown(smallest);
        }
    }
    private void swap(int i, int j) {
        Node node1 = heap.get(i);
        Node node2 = heap.get(j);

        nodePosition.put(node1.key, j);
        nodePosition.put(node2.key, i);

        heap.set(i, node2);
        heap.set(j, node1);
    }

    public class Node {
        int weight;
        T key;
    }
}
class Vertex<T> {
    private T data;
    private List<Edge<T>> edges;

    public Vertex(T data) {
        this.data = data;
        this.edges = new ArrayList<>();
    }

    public T getData() {
        return data;
    }

    public List<Edge<T>> getEdges() {
        return edges;
    }

    public void addEdge(Edge<T> edge) {
        edges.add(edge);
    }
```

```
    public void addBidirectionalEdge(Edge<T> edge) {
        edges.add(edge);
        Vertex<T> other = edge.getVertex1() == this ? edge.getVertex2() : edge.getVertex1();
        other.edges.add(edge);
    }

    @Override
    public String toString() {
        return data.toString();
    }
}

class Edge<T> {
    private Vertex<T> vertex1;
    private Vertex<T> vertex2;
    private int weight;

    public Edge(Vertex<T> vertex1, Vertex<T> vertex2, int weight) {
        this.vertex1 = vertex1;
        this.vertex2 = vertex2;
        this.weight = weight;
    }

    public Vertex<T> getVertex1() {
        return vertex1;
    }

    public Vertex<T> getVertex2() {
        return vertex2;
    }

    public int getWeight() {
        return weight;
    }
}

class Graph<T> {
    private List<Vertex<T>> vertices;

    public Graph() {
        vertices = new ArrayList<>();
    }
```

```
    public void addVertex(Vertex<T> vertex) {
        vertices.add(vertex);
    }

    public List<Vertex<T>> getAllVertex() {
        return vertices;
    }
}

public class DijkstraShortestPath {

    public Map<Vertex<Integer>, Map<Integer, List<Vertex<Integer>>>> shortestPath(Graph<Integer>
graph, Vertex<Integer> sourceVertex) {
        BinaryMinHeap<Vertex<Integer>> minHeap = new
BinaryMinHeap<>(graph.getAllVertex().size());
        Map<Vertex<Integer>, Integer> distance = new HashMap<>();
        Map<Vertex<Integer>, Vertex<Integer>> parent = new HashMap<>();
        Map<Vertex<Integer>, Map<Integer, List<Vertex<Integer>>>> paths = new HashMap<>();

        for (Vertex<Integer> vertex : graph.getAllVertex()) {
            int initialDistance = (vertex == sourceVertex) ? 0 : Integer.MAX_VALUE;
            minHeap.add(initialDistance, vertex);
        }

        minHeap.decrease(sourceVertex, 0);
        distance.put(sourceVertex, 0);
        parent.put(sourceVertex, null);

        while (!minHeap.empty()) {
            BinaryMinHeap<Vertex<Integer>>.Node heapNode = minHeap.extractMinNode();
            Vertex<Integer> current = heapNode.key;
            distance.put(current, heapNode.weight);

            for (Edge<Integer> edge : current.getEdges()) {
                Vertex<Integer> adjacent = getVertexForEdge(current, edge);

                if (!minHeap.containsData(adjacent)) {
                    continue;
                }

                int newDistance = distance.get(current) + edge.getWeight();

                if (minHeap.getWeight(adjacent) > newDistance) {
                    minHeap.decrease(adjacent, newDistance);
```

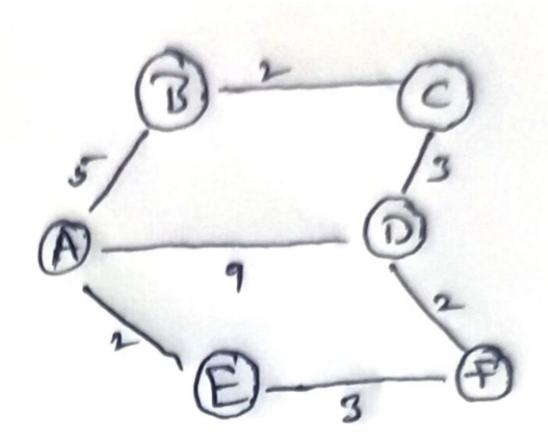
```
                parent.put(adjacent, current);

                List<Vertex<Integer>> path = new ArrayList<>();
                Vertex<Integer> temp = adjacent;
                while (temp != sourceVertex) {
                    path.add(temp);
                    temp = parent.get(temp);
                }
                path.add(sourceVertex);
                Collections.reverse(path);

                Map<Integer, List<Vertex<Integer>>> pathsForAdjacent = new HashMap<>();
                pathsForAdjacent.put(newDistance, path);
                paths.put(adjacent, pathsForAdjacent);
            }
        }
    }

    return paths;
}

private Vertex<Integer> getVertexForEdge(Vertex<Integer> v, Edge<Integer> e) {
    return e.getVertex1().equals(v) ? e.getVertex2() : e.getVertex1();
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    Graph<Integer> graph = new Graph<>();
    Map<Integer, Vertex<Integer>> vertexMap = new HashMap<>();

    System.out.print("Enter the number of vertices: ");
    int numVertices = scanner.nextInt();

    System.out.println("Enter the vertices (integer values):");
    for (int i = 0; i < numVertices; i++) {
        int vertexValue = scanner.nextInt();
        Vertex<Integer> vertex = new Vertex<>(vertexValue);
        graph.addVertex(vertex);
        vertexMap.put(vertexValue, vertex);
    }

    System.out.print("Enter the number of edges: ");
    int numEdges = scanner.nextInt();
```

```
        System.out.println("Enter the edges in the format 'vertex1 vertex2 weight':");
        for (int i = 0; i < numEdges; i++) {
            int vertex1 = scanner.nextInt();
            int vertex2 = scanner.nextInt();
            int weight = scanner.nextInt();
           Vertex<Integer> v1 = vertexMap.get(vertex1);
            Vertex<Integer> v2 = vertexMap.get(vertex2);
            Edge<Integer> edge = new Edge<>(v1, v2, weight);
            v1.addEdge(edge);
            v2.addEdge(edge);
        }

        DijkstraShortestPath dijkstra = new DijkstraShortestPath();

        System.out.print("Enter the source vertex: ");
        int sourceVertexValue = scanner.nextInt();
        Vertex<Integer> sourceVertex = vertexMap.get(sourceVertexValue);

        if (sourceVertex == null) {
            System.out.println("Invalid source vertex.");
            return;
        }

        Map<Vertex<Integer>, Map<Integer, List<Vertex<Integer>>>> shortestPaths =
dijkstra.shortestPath(graph, sourceVertex);

        System.out.println("Shortest paths and their total distances from " + sourceVertexValue + ":");
        for (Map.Entry<Vertex<Integer>, Map<Integer, List<Vertex<Integer>>>> entry :
shortestPaths.entrySet()) {
            Vertex<Integer> destination = entry.getKey();
            System.out.println("To " + destination.getData() + ":");
            Map<Integer, List<Vertex<Integer>>> paths = entry.getValue();
            for (Map.Entry<Integer, List<Vertex<Integer>>> pathEntry : paths.entrySet()) {
                System.out.println("  Distance: " + pathEntry.getKey() + " Path: " + pathEntry.getValue());
            }
        }
    }
}
```

Output:

```
root@IPhone12promax:~# java DijkstraShortestPath
Enter the number of vertices: 6
Enter the vertices (characters a, b, c, ...):
a b c d e f
Enter the number of edges: 7
Enter the edges in the format 'vertex1 vertex2 weight':
a b 5
a d 9
a e 2
b c 2
c d 3
d f 2
e f 3
Enter the source vertex: b
Shortest paths and their total distances from b:
To c:
  Distance: 2 Path: [b, c]
To e:
  Distance: 7 Path: [b, a, e]
To a:
  Distance: 5 Path: [b, a]
To f:
  Distance: 7 Path: [b, c, d, f]
To d:
  Distance: 5 Path: [b, c, d]
root@IPhone12promax:~#
```

Graph:



Design Analysis And Algorithms

Prims Algorithm:

```
import java.util.Scanner;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Collections;
import java.util.HashSet;
 import java.util.Set;
class BinaryMinHeap<T> {
    private Map<T, Integer> nodePosition;
    private List<Node> heap;
    private int maxSize;

    public BinaryMinHeap(int maxSize) {
        this.nodePosition = new HashMap<>();
        this.heap = new ArrayList<>(maxSize);
        this.maxSize = maxSize;
    }

    public void add(int weight, T key) {
        if (nodePosition.containsKey(key)) {
            decrease(key, weight);
        } else {
            Node node = new Node();
            node.weight = weight;
            node.key = key;
            heap.add(node);
            nodePosition.put(key, heap.size() - 1);
            heapifyUp(heap.size() - 1);
        }
    }

    public void decrease(T key, int newWeight) {
        int position = nodePosition.get(key);
        Node node = heap.get(position);
        node.weight = newWeight;
        heapifyUp(position);
    }

    public boolean containsData(T key) {
```

```
    return nodePosition.containsKey(key);
}

public boolean empty() {
    return heap.isEmpty();
}

public Node extractMinNode() {
    Node minNode = heap.get(0);
    Node lastNode = heap.remove(heap.size() - 1);
    nodePosition.remove(minNode.key);
    if (!heap.isEmpty()) {
        heap.set(0, lastNode);
        nodePosition.put(lastNode.key, 0);
        heapifyDown(0);
    }
    return minNode;
}

public int getWeight(T key) {
    Integer position = nodePosition.get(key);
    if (position == null) {
        return Integer.MAX_VALUE;
    } else {
        return heap.get(position).weight;
    }
}

private void heapifyUp(int currentIndex) {
    int parentIndex = (currentIndex - 1) / 2;
    while (currentIndex > 0 && heap.get(currentIndex).weight < heap.get(parentIndex).weight) {
        swap(currentIndex, parentIndex);
        currentIndex = parentIndex;
        parentIndex = (currentIndex - 1) / 2;
    }
}

private void heapifyDown(int currentIndex) {
    int leftIndex = 2 * currentIndex + 1;
    int rightIndex = 2 * currentIndex + 2;
    int smallest = currentIndex;

    if (leftIndex < heap.size() && heap.get(leftIndex).weight < heap.get(smallest).weight) {
        smallest = leftIndex;
```

```
        }
        if (rightIndex < heap.size() && heap.get(rightIndex).weight < heap.get(smallest).weight) {
            smallest = rightIndex;
        }

        if (smallest != currentIndex) {
            swap(currentIndex, smallest);
            heapifyDown(smallest);
        }
    }

    private void swap(int i, int j) {
        Node node1 = heap.get(i);
        Node node2 = heap.get(j);

        nodePosition.put(node1.key, j);
        nodePosition.put(node2.key, i);

        heap.set(i, node2);
        heap.set(j, node1);
    }

    public class Node {
        int weight;
        T key;
    }
}

class Vertex<T> {
    private T data;
    private List<Edge<T>> edges;

    public Vertex(T data) {
        this.data = data;
        this.edges = new ArrayList<>();
    }

    public T getData() {
        return data;
    }

    public List<Edge<T>> getEdges() {
        return edges;
    }
```

```
    public void addEdge(Edge<T> edge) {
        edges.add(edge);
    }

    public void addBidirectionalEdge(Edge<T> edge) {
        edges.add(edge);
        Vertex<T> other = edge.getVertex1() == this ? edge.getVertex2() : edge.getVertex1();
        other.edges.add(edge);
    }

    @Override
    public String toString() {
        return data.toString();
    }
}

class Edge<T> {
    private Vertex<T> vertex1;
    private Vertex<T> vertex2;
    private int weight;

    public Edge(Vertex<T> vertex1, Vertex<T> vertex2, int weight) {
        this.vertex1 = vertex1;
        this.vertex2 = vertex2;
        this.weight = weight;
    }

    public Vertex<T> getVertex1() {
        return vertex1;
    }

    public Vertex<T> getVertex2() {
        return vertex2;
    }

    public int getWeight() {
        return weight;
    }
}

class Graph<T> {
    private List<Vertex<T>> vertices;
```

```
    public Graph() {
        vertices = new ArrayList<>();
    }

    public void addVertex(Vertex<T> vertex) {
        vertices.add(vertex);
    }

    public List<Vertex<T>> getAllVertex() {
        return vertices;
    }
}

class PrimMinimumSpanningTree<T> {
    private Map<Vertex<T>, Vertex<T>> parent;
    private Map<Vertex<T>, Integer> key;
    private Set<Vertex<T>> visitedVertices;
    private BinaryMinHeap<Vertex<T>> minHeap;

    public Map<Vertex<T>, List<Edge<T>>> minimumSpanningTree(Graph<T> graph, Vertex<T>
startVertex) {
        this.parent = new HashMap<>();
        this.key = new HashMap<>();
        this.visitedVertices = new HashSet<>();
        this.minHeap = new BinaryMinHeap<>(graph.getAllVertex().size());

        Map<Vertex<T>, List<Edge<T>>> minimumSpanningTree = new HashMap<>();

        for (Vertex<T> vertex : graph.getAllVertex()) {
            key.put(vertex, Integer.MAX_VALUE);
            minHeap.add(Integer.MAX_VALUE, vertex);
        }

        minHeap.decrease(startVertex, 0);
        key.put(startVertex, 0);
        parent.put(startVertex, null);

        while (!minHeap.empty()) {
            BinaryMinHeap<Vertex<T>>.Node heapNode = minHeap.extractMinNode();
            Vertex<T> currentVertex = heapNode.key;
            visitedVertices.add(currentVertex);

            if (parent.get(currentVertex) != null) {
                Vertex<T> start = parent.get(currentVertex);
```

```
            Edge<T> minimumSpanningTreeEdge = findEdge(graph, start, currentVertex);
            minimumSpanningTree.computeIfAbsent(start, k -> new
ArrayList<>()).add(minimumSpanningTreeEdge);
        }

        for (Edge<T> edge : currentVertex.getEdges()) {
            Vertex<T> adjacentVertex = getOtherVertex(edge, currentVertex);
            if (!visitedVertices.contains(adjacentVertex)) {
                int weight = edge.getWeight();
                if (key.get(adjacentVertex) > weight) {
                    minHeap.decrease(adjacentVertex, weight);
                    key.put(adjacentVertex, weight);
                    parent.put(adjacentVertex, currentVertex);
                }
            }         }}
        return minimumSpanningTree;
    }

    public Vertex<T> getOtherVertex(Edge<T> edge, Vertex<T> currentVertex) {
        return edge.getVertex1().equals(currentVertex) ? edge.getVertex2() : edge.getVertex1();
    }

    private Edge<T> findEdge(Graph<T> graph, Vertex<T> start, Vertex<T> end) {
        for (Edge<T> edge : start.getEdges()) {
            Vertex<T> adjacent = getOtherVertex(edge, start);
            if (adjacent.equals(end)) {
                return edge;
            }
        }
        return null;
    }
}
public class PrimAlgorithm {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Graph<Character> graph = new Graph<>();
        Map<Character, Vertex<Character>> vertexMap = new HashMap<>();
        System.out.print("Enter the number of vertices: ");
        int numVertices = scanner.nextInt();
        System.out.println("Enter the vertices (characters a, b, c, ...):");
        for (int i = 0; i < numVertices; i++) {
            char vertexValue = scanner.next().charAt(0);
            Vertex<Character> vertex = new Vertex<>(vertexValue);
            graph.addVertex(vertex);
```

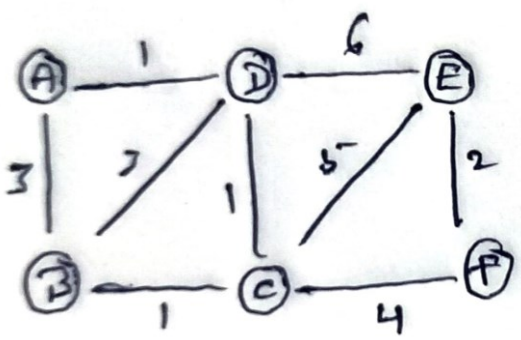
```
            vertexMap.put(vertexValue, vertex);
        }
        System.out.print("Enter the number of edges: ");
        int numEdges = scanner.nextInt();
        System.out.println("Enter the edges in the format 'vertex1 vertex2 weight':");
        for (int i = 0; i < numEdges; i++) {
            char vertex1 = scanner.next().charAt(0);
            char vertex2 = scanner.next().charAt(0);
            int weight = scanner.nextInt();

            Vertex<Character> v1 = vertexMap.get(vertex1);
            Vertex<Character> v2 = vertexMap.get(vertex2);

            Edge<Character> edge = new Edge<>(v1, v2, weight);
            v1.addEdge(edge);
            v2.addEdge(edge);
        }
        PrimMinimumSpanningTree<Character> prim = new PrimMinimumSpanningTree<>();

        System.out.print("Enter the source vertex: ");
        char sourceVertexValue = scanner.next().charAt(0);
        Vertex<Character> sourceVertex = vertexMap.get(sourceVertexValue);

        if (sourceVertex == null) {
            System.out.println("Invalid source vertex.");
            return;
        }

        Map<Vertex<Character>, List<Edge<Character>>> minimumSpanningTree =
prim.minimumSpanningTree(graph, sourceVertex);

        System.out.println("Minimum Spanning Tree Edges from " + sourceVertexValue + ":");
        for (Map.Entry<Vertex<Character>, List<Edge<Character>>> entry :
minimumSpanningTree.entrySet()) {
            Vertex<Character> startVertex = entry.getKey();
            System.out.println("From " + startVertex.getData() + ":");
            List<Edge<Character>> edges = entry.getValue();
            for (Edge<Character> edge : edges) {
                Vertex<Character> otherVertex = prim.getOtherVertex(edge, startVertex);
                System.out.println("  To " + otherVertex.getData() + " Weight: " + edge.getWeight());
            }
        } }}
```

Design Analysis And Algorithms

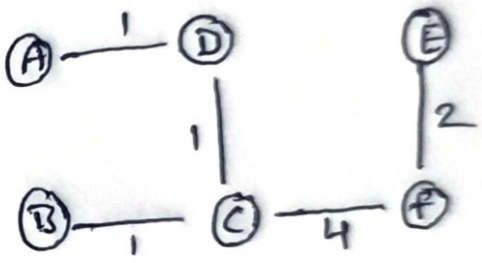
Output:

```
root@IPhone12promax:~# java PrimAlgorithm
Enter the number of vertices: 6
Enter the vertices (characters a, b, c, ...):
a b c d e f
Enter the number of edges: 9
Enter the edges in the format 'vertex1 vertex2 weight':
a b 3
a d 1
b c 1
b d 3
c d 1
c e 5
c f 4
d e 6
e f 2
Enter the source vertex: c
Minimum Spanning Tree Edges from c:
From d:
  To a Weight: 1
From f:
  To e Weight: 2
From c:
  To b Weight: 1
  To d Weight: 1
  To f Weight: 4
```

Graph:



Minimum Spanning Tree:



Design Analysis And Algorithms

Kruskals Algorithm:

```
import java.util.Scanner;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Collections;


class Edge<T> {
    private Vertex<T> vertex1;
    private Vertex<T> vertex2;
    private int weight;

    public Edge(Vertex<T> vertex1, Vertex<T> vertex2, int weight) {
        this.vertex1 = vertex1;
        this.vertex2 = vertex2;
        this.weight = weight;
    }

    public Vertex<T> getVertex1() {
        return vertex1;
    }

    public Vertex<T> getVertex2() {
        return vertex2;
    }

    public int getWeight() {
        return weight;
    }
}

class Vertex<T> {
    private T data;
    private List<Edge<T>> edges;

    public Vertex(T data) {
        this.data = data;
        this.edges = new ArrayList<>();
    }
```

```
    public T getData() {
        return data;
    }

    public List<Edge<T>> getEdges() {
        return edges;
    }

    public void addEdge(Edge<T> edge) {
        edges.add(edge);
    }
}

class BinaryMinHeap<T> {
    // Private inner class to hold the edges and their positions
    private class Node {
        Edge<T> edge;
        int position;
    }

    private Map<Edge<T>, Integer> nodePosition;
    private List<Node> heap;

    public BinaryMinHeap() {
        this.nodePosition = new HashMap<>();
        this.heap = new ArrayList<>();
    }

    public void add(Edge<T> edge) {
        Node node = new Node();
        node.edge = edge;
        nodePosition.put(edge, heap.size());
        heap.add(node);
        heapifyUp(heap.size() - 1);
    }

    public Edge<T> extractMin() {
        Node minNode = heap.get(0);
        Edge<T> minEdge = minNode.edge;
        Node lastNode = heap.remove(heap.size() - 1);

        if (!heap.isEmpty()) {
            heap.set(0, lastNode);
            nodePosition.put(lastNode.edge, 0);
```

```
            heapifyDown(0);
        }

        return minEdge;
    }

    public boolean isEmpty() {
        return heap.isEmpty();
    }

    private void heapifyUp(int currentIndex) {
        int parentIndex = (currentIndex - 1) / 2;
        while (currentIndex > 0 && heap.get(currentIndex).edge.getWeight() <
heap.get(parentIndex).edge.getWeight()) {
            swap(currentIndex, parentIndex);
            currentIndex = parentIndex;
            parentIndex = (currentIndex - 1) / 2;
        }
    }

    private void heapifyDown(int currentIndex) {
        int leftIndex = 2 * currentIndex + 1;
        int rightIndex = 2 * currentIndex + 2;
        int smallest = currentIndex;

        if (leftIndex < heap.size() && heap.get(leftIndex).edge.getWeight() <
heap.get(smallest).edge.getWeight()) {
            smallest = leftIndex;
        }
        if (rightIndex < heap.size() && heap.get(rightIndex).edge.getWeight() <
heap.get(smallest).edge.getWeight()) {
            smallest = rightIndex;
        }

        if (smallest != currentIndex) {
            swap(currentIndex, smallest);
            heapifyDown(smallest);
        }
    }

    private void swap(int i, int j) {
        Node node1 = heap.get(i);
        Node node2 = heap.get(j);
```

```
            nodePosition.put(node1.edge, j);
            nodePosition.put(node2.edge, i);

            heap.set(i, node2);
            heap.set(j, node1);
        }
    }

    class DisjointSet<T> {
        private Map<Vertex<T>, Vertex<T>> parent;
        private Map<Vertex<T>, Integer> rank;

        public DisjointSet(List<Vertex<T>> vertices) {
            parent = new HashMap<>();
            rank = new HashMap<>();

            for (Vertex<T> vertex : vertices) {
                parent.put(vertex, vertex);
                rank.put(vertex, 0);
            }
        }

        public Vertex<T> find(Vertex<T> vertex) {
            if (vertex != parent.get(vertex)) {
                parent.put(vertex, find(parent.get(vertex)));
            }
            return parent.get(vertex);
        }

        public void union(Vertex<T> x, Vertex<T> y) {
            Vertex<T> rootX = find(x);
            Vertex<T> rootY = find(y);

            if (rootX == rootY) {
                return;
            }

            if (rank.get(rootX) < rank.get(rootY)) {
                parent.put(rootX, rootY);
            } else if (rank.get(rootX) > rank.get(rootY)) {
                parent.put(rootY, rootX);
            } else {
                parent.put(rootY, rootX);
                rank.put(rootX, rank.get(rootX) + 1);
```

```
        }
    }
}

class KruskalMinimumSpanningTree<T> {
    public List<Edge<T>> minimumSpanningTree(List<Vertex<T>> vertices, List<Edge<T>> edges) {
        BinaryMinHeap<T> minHeap = new BinaryMinHeap<>();

        for (Edge<T> edge : edges) {
            minHeap.add(edge);
        }

        List<Edge<T>> mst = new ArrayList<>();
        DisjointSet<T> disjointSet = new DisjointSet<>(vertices);

        while (!minHeap.isEmpty()) {
            Edge<T> edge = minHeap.extractMin();
            Vertex<T> root1 = disjointSet.find(edge.getVertex1());
            Vertex<T> root2 = disjointSet.find(edge.getVertex2());

            if (root1 != root2) {
                mst.add(edge);
                disjointSet.union(root1, root2);
            }
        }

        return mst;
    }
}

public class KruskalAlgorithm {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        List<Vertex<Character>> vertices = new ArrayList<>();
        List<Edge<Character>> edges = new ArrayList<>();

        System.out.print("Enter the number of vertices (a, b, c, ...): ");
        int numVertices = scanner.nextInt();

        System.out.println("Enter the vertices (character values):");
        for (int i = 0; i < numVertices; i++) {
            char vertexValue = scanner.next().charAt(0);
            Vertex<Character> vertex = new Vertex<>(vertexValue);
```

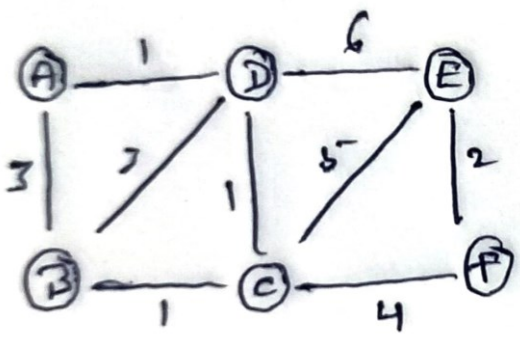
```
            vertices.add(vertex);
        }

        System.out.print("Enter the number of edges: ");
        int numEdges = scanner.nextInt();

        System.out.println("Enter the edges in the format 'vertex1 vertex2 weight':");
        for (int i = 0; i < numEdges; i++) {
            char vertex1 = scanner.next().charAt(0);
            char vertex2 = scanner.next().charAt(0);
            int weight = scanner.nextInt();

            Vertex<Character> v1 = null;
            Vertex<Character> v2 = null;
            for (Vertex<Character> v : vertices) {
                if (v.getData() == vertex1) {
                    v1 = v;
                }
                if (v.getData() == vertex2) {
                    v2 = v;
                }
            }

            if (v1 != null && v2 != null) {
                Edge<Character> edge = new Edge<>(v1, v2, weight);
                edges.add(edge);
            } else {
                System.out.println("Invalid edge vertices!");
            }
        }

        KruskalMinimumSpanningTree<Character> kruskal = new KruskalMinimumSpanningTree<>();
        List<Edge<Character>> minimumSpanningTree = kruskal.minimumSpanningTree(vertices,
edges);

        System.out.println("Minimum Spanning Tree Edges:");
        for (Edge<Character> edge : minimumSpanningTree) {
            System.out.println("From " + edge.getVertex1().getData() + " To " +
edge.getVertex2().getData() + " Weight: " + edge.getWeight());
        }
    }
}
```

Design Analysis And Algorithms

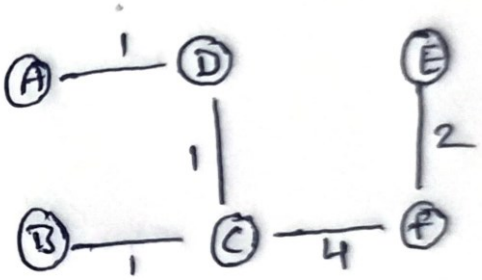
Output:

```
root@IPhone12promax:~# javac KruskalAlgorithm.java
root@IPhone12promax:~# java KruskalAlgorithm
Enter the number of vertices (a, b, c, ...): 6
Enter the vertices (character values):
a b c d e f
Enter the number of edges: 9
Enter the edges in the format 'vertex1 vertex2 weight':
a b 3
a d 1
b c 1
b d 3
c d 1
c e 5
c f 4
d e 6
e f 2
Minimum Spanning Tree Edges:
From a To d Weight: 1
From c To d Weight: 1
From b To c Weight: 1
From e To f Weight: 2
From c To f Weight: 4
```

Graph:



Minimum Spanning Tree:



By
Aari Eswar
21VV1A1201