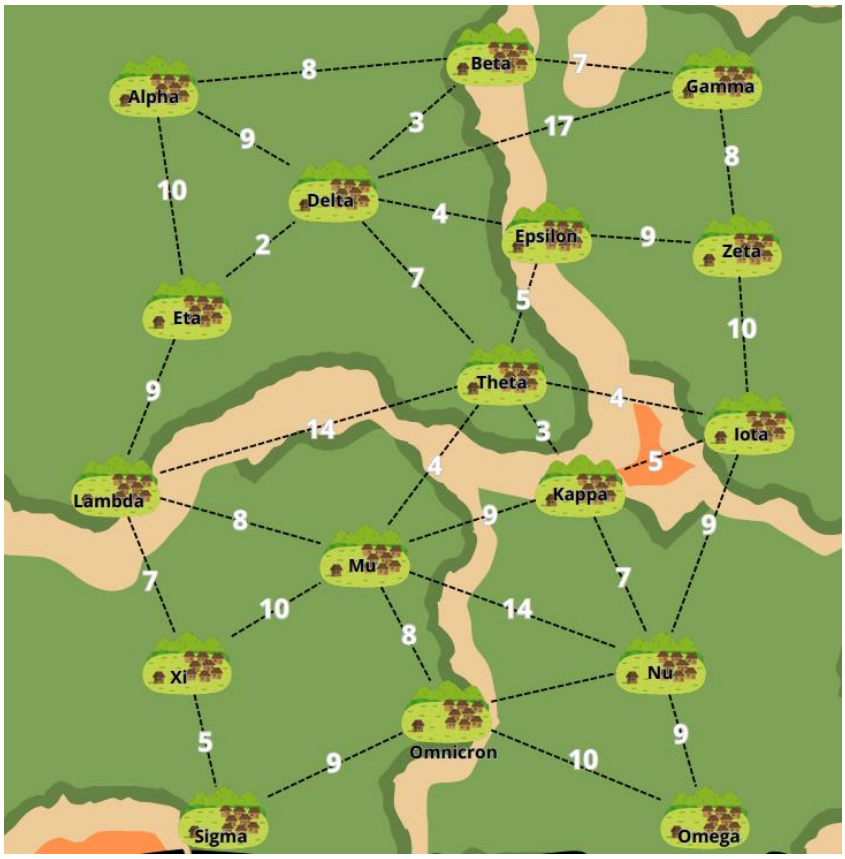
MODUL V

GRAPH

5.1 PERMASALAHAN



Pak ophet adalah calon bupati dari kabupaten merembu, pak ophet berasal dari salah satu desa di kabupaten tersebut dalam rangka melakukan kampanye pak ophet menginginkan supaya bisa mengujungin semua desa dengan rute tercepat yang dimana berarti kalian di minta untuk membuat list jalur dari desa pak ophet ke setiap desa lainnya dalam jalan tercepat

Perlu di perhatikan bahwa jalan tercepat blum tentu berasal dari jalan langsung ke tempatnya, dan bisa saja ada jalan yang lebih pendek dengan mengitari desa lainnya

Perlu di perhatikan bahwa juga untuk di urutkan desa terdekat terlebih dahulu karena pak ophet menginginkan untuk kampane di desa desa terdekatnya terlebih dahuluMengikuti urutan Pre-Order Transversal

a. Apabila rumah pak ophet di Theta

```
Theta
Theta => Kappa
Theta => Iota
Theta => Mu
Theta => Epsilon
Theta => Delta
Theta => Delta => Eta
Theta => Delta => Beta
Theta => Kappa => Nu
Theta => Mu => Lambda
Theta => Mu => Omnicron
Theta => Epsilon => Zeta
Theta => Mu => Xi
Theta => Delta => Alpha
Theta => Delta => Beta => Gamma
Theta => Kappa => Nu => Omega
Theta => Mu => Xi => Sigma
```

b. Apabila rumah pak ophet di Alpha

```
Alpha
Alpha => Beta
Alpha => Delta
Alpha => Eta
Alpha => Delta => Epsilon
Alpha => Beta => Gamma
Alpha => Delta => Theta
Alpha => Eta => Lambda
Alpha => Delta => Theta => Kappa
Alpha => Delta => Theta => Iota
Alpha => Delta => Theta => Mu
Alpha => Delta => Epsilon => Zeta
Alpha => Delta => Theta => Kappa => Nu
Alpha => Eta => Lambda => Xi
Alpha => Delta => Theta => Mu => Omnicron
Alpha => Eta => Lambda => Xi => Sigma
Alpha => Delta => Theta => Kappa => Nu => Omega
```

c. Apabila rumah pak ophet di Mu

```
Mu
Mu => Theta
Mu => Theta => Kappa
Mu => Theta => Iota
Mu => Lambda
Mu => Omnicron
Mu => Theta => Epsilon
Mu => Xi
Mu => Theta => Delta
Mu => Omnicron => Nu
Mu => Theta => Delta => Eta
Mu => Theta => Delta => Beta
Mu => Xi => Sigma
Mu => Theta => Epsilon => Zeta
Mu => Omnicron => Omega
Mu => Theta => Delta => Alpha
Mu => Theta => Delta => Beta => Gamma
```

5.2 HASIL PERCOBAAN

5.2.1 Pemilihan Bupati Merembu

1. Algoritma
   a. Buat kelas Node yang yang merepresentasikan simpul dalam graf. Kelas ini memiliki atribut seperti nama simpul (name), jarak ke simpul lain (distance), status kunjungan (visited), daftar sisi (edges), simpul berikutnya (next), dan simpul induk (parent).
   b. Membuat kelas Edge untuk merepresentasikan sisi (edge) dalam graf. Sisi memiliki atribut seperti target simpul (target), bobot (weight), dan referensi ke sisi berikutnya (next) untuk membentuk daftar sisi.
   c. Membuat kelas EdgeList untuk menyimpan daftar sisi yang terhubung dengan simpul tertentu. Metode addEdge digunakan untuk menambahkan sisi baru ke dalam daftar, dengan traversal hingga akhir daftar jika sisi sudah ada.
   d. Membuat kelas Graph untuk merepresentasikan graf. Graf berisi daftar simpul (nodes), fungsi untuk menambahkan simpul (addNode), menambahkan sisi (addEdge), dan algoritma Dijkstra untuk menghitung jalur terpendek dari simpul
   e. Mengimplementasikan Algoritma Dijkstra yang dimulai dengan mengatur jarak simpul awal menjadi nol. Simpul dengan jarak terkecil yang belum dikunjungi diproses, dan jarak simpul tetangga diperbarui jika jalur baru lebih pendek dari jarak sebelumnya.
2. *Source Code*

```
public class Node {
    String name;
    boolean visited;
    int distance;
    EdgeList edges;
    Node next;
    Node parent;

    Node(String name) {
        this.name = name;
        this.visited = false;
        this.distance = Integer.MAX_VALUE;
        this.edges = new EdgeList();
        this.next = null;
        this.parent = null;
    }
}

class Edge {
    Node target;
    int weight;
    Edge next;

    Edge(Node target, int weight) {
        this.target = target;
```

```
        this.weight = weight;
        this.next = null;
    }
}

class EdgeList {
    Edge head;

    void addEdge(Edge edge) {
        if (head == null) {
            head = edge;
        } else {
            Edge current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = edge;
        }
    }
}
public class NodeList {
    Node head;

    void addNode(Node node) {
        if (head == null) {
            head = node;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = node;
        }
    }

    Node findNode(String name) {
        Node current = head;
        while (current != null) {
            if (current.name.equals(name)) {
                return current;
            }
            current = current.next;
        }
        return null;
    }
}
class Edge {
    Node target;
    int weight;
    Edge next;

    Edge(Node target, int weight) {
        this.target = target;
        this.weight = weight;
        this.next = null;
    }
}
public class Graph {
    NodeList nodes;
    String[][] paths = new String[100][100];
    int[] pathLengths = new int[100];
```

```
    int pathCount = 0;

    Graph() {
        nodes = new NodeList();
    }

    void addNode(String name) {
        nodes.addNode(new Node(name));
    }

    void addEdge(String from, String to, int weight) {
        Node fromNode = nodes.findNode(from);
        Node toNode = nodes.findNode(to);
        if (fromNode != null && toNode != null) {
            fromNode.edges.addEdge(new Edge(toNode, weight));
        }
    }

    void dijkstra(String startName) {
        Node startNode = nodes.findNode(startName);
        if (startNode == null) return;

        startNode.distance = 0;

        while (true) {
            Node currentNode = null;

            Node temp = nodes.head;
            while (temp != null) {
                if (!temp.visited && (currentNode == null ||
temp.distance < currentNode.distance)) {
                    currentNode = temp;
                }
                temp = temp.next;
            }

            if (currentNode == null) break;

            currentNode.visited = true;

            Edge edge = currentNode.edges.head;
            while (edge != null) {
                Node neighbor = edge.target;
                int newDistance = currentNode.distance + edge.weight;
                if (newDistance < neighbor.distance) {
                    neighbor.distance = newDistance;
                    neighbor.parent = currentNode;
                }
                edge = edge.next;
            }
        }
    }

    void collectPath(Node target) {
        String[] path = new String[100];
        int length = 0;

        while (target != null) {
            path[length++] = target.name;
            target = target.parent;
        }
```

```
        paths[pathCount] = new String[length];
        for (int i = 0; i < length; i++) {
            paths[pathCount][length - 1 - i] = path[i];
        }
        pathLengths[pathCount] = length;
        pathCount++;
    }

    void sortPaths() {
        for (int i = 0; i < pathCount - 1; i++) {
            for (int j = 0; j < pathCount - i - 1; j++) {
                if (pathLengths[j] > pathLengths[j + 1]) {

                    String[] tempPath = paths[j];
                    paths[j] = paths[j + 1];
                    paths[j + 1] = tempPath;

                    int tempLength = pathLengths[j];
                    pathLengths[j] = pathLengths[j + 1];
                    pathLengths[j + 1] = tempLength;
                }
            }
        }
    }

    void printPaths(String startName) {
        Node startNode = nodes.findNode(startName);
        if (startNode == null) return;

        Node current = nodes.head;
        while (current != null) {
            if (!current.name.equals(startName)) {
                collectPath(current);
            }
            current = current.next;
        }

        sortPaths();

        for (int i = 0; i < pathCount; i++) {
            System.out.print("");
            for (int j = 0; j < pathLengths[i]; j++) {
                System.out.print(paths[i][j]);
                if (j < pathLengths[i] - 1) {
                    System.out.print(" => ");
                }
            }
            System.out.println();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Graph graph = new Graph();
        Graph graph1 = new Graph();
        Graph graph2 = new Graph();


        graph.addNode("Alpha");
```

```
        graph.addNode("Beta");
        graph.addNode("Gamma");
        graph.addNode("Delta");
        graph.addNode("Epsilon");
        graph.addNode("Zeta");
        graph.addNode("Eta");
        graph.addNode("Theta");
        graph.addNode("Iota");
        graph.addNode("Kappa");
        graph.addNode("Lambda");
        graph.addNode("Mu");
        graph.addNode("Nu");
        graph.addNode("Xi");
        graph.addNode("omnicron");
        graph.addNode("Sigma");
        graph.addNode("Omega");

        graph.addEdge("Alpha", "Beta", 8);
        graph.addEdge("Alpha", "Delta", 9);
        graph.addEdge("Alpha", "Eta", 10);
        graph.addEdge("Beta", "Alpha", 8);
        graph.addEdge("Beta", "Delta", 3);
        graph.addEdge("Beta", "Gamma", 7);
        graph.addEdge("Gamma", "Beta", 7);
        graph.addEdge("Gamma", "Delta", 17);
        graph.addEdge("Gamma", "Zeta", 8);
        graph.addEdge("Delta", "Alpha", 9);
        graph.addEdge("Delta", "Beta", 3);
        graph.addEdge("Delta", "Gamma", 17);
        graph.addEdge("Delta", "Epsilon", 4);
        graph.addEdge("Delta", "Theta", 7);
        graph.addEdge("Delta", "Eta", 2);
        graph.addEdge("Eta", "Alpha", 10);
        graph.addEdge("Eta", "Delta", 2);
        graph.addEdge("Eta", "Lambda", 9);
        graph.addEdge("Lambda", "Eta", 9);
        graph.addEdge("Lambda", "Mu", 8);
        graph.addEdge("Theta", "Delta", 7);
        graph.addEdge("Theta", "Epsilon", 4);
        graph.addEdge("Theta", "Iota", 3);
        graph.addEdge("Theta", "Kappa", 9);
        graph.addEdge("Mu", "Lambda", 8);
        graph.addEdge("Mu", "Kappa", 14);
        graph.addEdge("Mu", "Xi", 8);
        graph.addEdge("Xi", "Mu", 8);
        graph.addEdge("Xi", "omnicron", 5);
        graph.addEdge("omnicron", "Xi", 5);
        graph.addEdge("omnicron", "Sigma", 9);
        graph.addEdge("omnicron", "Nu", 9);
        graph.addEdge("Nu", "omnicron", 9);
        graph.addEdge("Nu", "Kappa", 9);
        graph.addEdge("Nu", "Omega", 10);
        graph.addEdge("Omega", "Nu", 10);
        graph.addEdge("Sigma", "omnicron", 9);
        graph.addEdge("Kappa", "Theta", 9);
        graph.addEdge("Kappa", "Mu", 14);
        graph.addEdge("Kappa", "Nu", 9);
        graph.addEdge("Kappa", "Iota", 5);
        graph.addEdge("Epsilon", "Delta", 4);
        graph.addEdge("Epsilon", "Theta", 4);
        graph.addEdge("Epsilon", "Zeta", 9);
```

```
        graph.addEdge("Zeta", "Epsilon", 9);
        graph.addEdge("Zeta", "Gamma", 8);
        graph.addEdge("Zeta", "Iota", 10);
        graph.addEdge("Iota", "Zeta", 10);
        graph.addEdge("Iota", "Theta", 3);
        graph.addEdge("Iota", "Kappa", 5);

        graph.dijkstra("Alpha");
        graph.printPaths("Alpha");

        graph1.addNode("Alpha");
        graph1.addNode("Beta");
        graph1.addNode("Gamma");
        graph1.addNode("Delta");
        graph1.addNode("Epsilon");
        graph1.addNode("Zeta");
        graph1.addNode("Eta");
        graph1.addNode("Theta");
        graph1.addNode("Iota");
        graph1.addNode("Kappa");
        graph1.addNode("Lambda");
        graph1.addNode("Mu");
        graph1.addNode("Nu");
        graph1.addNode("Xi");
        graph1.addNode("Omnicron");
        graph1.addNode("Sigma");
        graph1.addNode("Omega");

        graph1.addEdge("Alpha", "Beta", 8);
        graph1.addEdge("Alpha", "Delta", 9);
        graph1.addEdge("Alpha", "Eta", 10);
        graph1.addEdge("Beta", "Alpha", 8);
        graph1.addEdge("Beta", "Delta", 3);
        graph1.addEdge("Beta", "Gamma", 7);
        graph1.addEdge("Gamma", "Beta", 7);
        graph1.addEdge("Gamma", "Delta", 17);
        graph1.addEdge("Gamma", "Zeta", 8);
        graph1.addEdge("Delta", "Alpha", 9);
        graph1.addEdge("Delta", "Beta", 3);
        graph1.addEdge("Delta", "Gamma", 17);
        graph1.addEdge("Delta", "Epsilon", 4);
        graph1.addEdge("Delta", "Theta", 7);
        graph1.addEdge("Delta", "Eta", 2);
        graph1.addEdge("Eta", "Alpha", 10);
        graph1.addEdge("Eta", "Delta", 2);
        graph1.addEdge("Eta", "Lambda", 9);
        graph1.addEdge("Lambda", "Eta", 9);
        graph1.addEdge("Lambda", "Mu", 8);
        graph1.addEdge("Lambda", "Xi", 7);
        graph1.addEdge("Lambda", "Theta", 14);
        graph1.addEdge("Theta", "Delta", 7);
        graph1.addEdge("Theta", "Epsilon", 5);
        graph1.addEdge("Theta", "Mu", 4);
        graph1.addEdge("Theta", "Iota", 4);
        graph1.addEdge("Theta", "Kappa", 3);
        graph1.addEdge("Theta", "Lambda", 14);
        graph1.addEdge("Mu", "Lambda", 8);
        graph1.addEdge("Mu", "Kappa", 9);
        graph1.addEdge("Mu", "Xi", 10);
        graph1.addEdge("Mu", "Omnicron", 8);
        graph1.addEdge("Mu", "Theta", 4);
```

```
        graph1.addEdge("Mu", "Nu", 14);
        graph1.addEdge("Xi", "Mu", 10);
        graph1.addEdge("Xi", "Lambda",  7);
        graph1.addEdge("Xi", "Sigma", 9);
        graph1.addEdge("Omnicron", "Mu", 8);
        graph1.addEdge("Omnicron", "Sigma", 9);
        graph1.addEdge("Omnicron", "Nu", 7);
        graph1.addEdge("omnicron", "Omega", 10);
        graph1.addEdge("Nu", "omnicron", 7);
        graph1.addEdge("Nu", "Kappa", 7);
        graph1.addEdge("Nu", "Omega", 9);
        graph1.addEdge("Nu", "Iota", 9);
        graph1.addEdge("Nu", "Mu", 14);
        graph1.addEdge("Omega", "Nu", 9);
        graph1.addEdge("Omega", "Omnicron", 10);
        graph1.addEdge("Sigma", "omnicron", 9);
        graph1.addEdge("Sigma", "Xi", 5);
        graph1.addEdge("Kappa", "Theta", 3);
        graph1.addEdge("Kappa", "Mu", 9);
        graph1.addEdge("Kappa", "Nu", 7);
        graph1.addEdge("Kappa", "Iota", 5);
        graph1.addEdge("Epsilon", "Delta", 4);
        graph1.addEdge("Epsilon", "Theta", 5);
        graph1.addEdge("Epsilon", "Zeta", 9);
        graph1.addEdge("Zeta", "Epsilon", 9);
        graph1.addEdge("Zeta", "Gamma", 8);
        graph1.addEdge("Zeta", "Iota", 10);
        graph1.addEdge("Iota", "Zeta", 10);
        graph1.addEdge("Iota", "Theta", 4);
        graph1.addEdge("Iota", "Kappa", 5);
        graph1.addEdge("Iota", "Theta", 4);


System.out.println("\n================================================
=");
        graph1.dijkstra("Theta");
        graph1.printPaths("Theta");

        graph2.addNode("Alpha");
        graph2.addNode("Beta");
        graph2.addNode("Gamma");
        graph2.addNode("Delta");
        graph2.addNode("Epsilon");
        graph2.addNode("Zeta");
        graph2.addNode("Eta");
        graph2.addNode("Theta");
        graph2.addNode("Iota");
        graph2.addNode("Kappa");
        graph2.addNode("Lambda");
        graph2.addNode("Mu");
        graph2.addNode("Nu");
        graph2.addNode("Xi");
        graph2.addNode("Omnicron");
        graph2.addNode("Sigma");
        graph2.addNode("Omega");

        graph2.addEdge("Alpha", "Beta", 8);
        graph2.addEdge("Alpha", "Delta", 9);
        graph2.addEdge("Alpha", "Eta", 10);
        graph2.addEdge("Beta", "Alpha", 8);
        graph2.addEdge("Beta", "Delta", 3);
```

```
        graph2.addEdge("Beta", "Gamma", 7);
        graph2.addEdge("Gamma", "Beta", 7);
        graph2.addEdge("Gamma", "Delta", 17);
        graph2.addEdge("Gamma", "Zeta", 8);
        graph2.addEdge("Delta", "Alpha", 9);
        graph2.addEdge("Delta", "Beta", 3);
        graph2.addEdge("Delta", "Gamma", 17);
        graph2.addEdge("Delta", "Epsilon", 4);
        graph2.addEdge("Delta", "Theta", 7);
        graph2.addEdge("Delta", "Eta", 2);
        graph2.addEdge("Eta", "Alpha", 10);
        graph2.addEdge("Eta", "Delta", 2);
        graph2.addEdge("Eta", "Lambda", 9);
        graph2.addEdge("Lambda", "Eta", 9);
        graph2.addEdge("Lambda", "Mu", 8);
        graph2.addEdge("Lambda", "Xi", 7);
        graph2.addEdge("Lambda", "Theta", 14);
        graph2.addEdge("Theta", "Delta", 7);
        graph2.addEdge("Theta", "Epsilon", 5);
        graph2.addEdge("Theta", "Mu", 4);
        graph2.addEdge("Theta", "Iota", 4);
        graph2.addEdge("Theta", "Kappa", 3);
        graph2.addEdge("Theta", "Lambda", 14);
        graph2.addEdge("Mu", "Lambda", 8);
        graph2.addEdge("Mu", "Kappa", 9);
        graph2.addEdge("Mu", "Xi", 10);
        graph2.addEdge("Mu", "Omnicron", 8);
        graph2.addEdge("Mu", "Theta", 4);
        graph2.addEdge("Mu", "Nu", 14);
        graph2.addEdge("Xi", "Mu", 10);
        graph2.addEdge("Xi", "Lambda",  7);
        graph2.addEdge("Xi", "Sigma", 5);
        graph2.addEdge("Omnicron", "Mu", 8);
        graph2.addEdge("Omnicron", "Sigma", 9);
        graph2.addEdge("Omnicron", "Nu", 5);
        graph2.addEdge("omnicron", "Omega", 10);
        graph2.addEdge("Nu", "omnicron", 5);
        graph2.addEdge("Nu", "Kappa", 7);
        graph2.addEdge("Nu", "Omega", 9);
        graph2.addEdge("Nu", "Iota", 9);
        graph2.addEdge("Nu", "Mu", 14);
        graph2.addEdge("Omega", "Nu", 9);
        graph2.addEdge("Omega", "Omnicron", 10);
        graph2.addEdge("Sigma", "omnicron", 9);
        graph2.addEdge("Sigma", "Xi", 5);
        graph2.addEdge("Kappa", "Theta", 3);
        graph2.addEdge("Kappa", "Mu", 9);
        graph2.addEdge("Kappa", "Nu", 7);
        graph2.addEdge("Kappa", "Iota", 5);
        graph2.addEdge("Epsilon", "Delta", 4);
        graph2.addEdge("Epsilon", "Theta", 5);
        graph2.addEdge("Epsilon", "Zeta", 9);
        graph2.addEdge("Zeta", "Epsilon", 9);
        graph2.addEdge("Zeta", "Gamma", 8);
        graph2.addEdge("Zeta", "Iota", 10);
        graph2.addEdge("Iota", "Zeta", 10);
        graph2.addEdge("Iota", "Theta", 4);
        graph2.addEdge("Iota", "Kappa", 5);
        graph2.addEdge("Iota", "Theta", 4);
```

```
System.out.println("\n=============================================
=");
        graph2.dijkstra("Mu");
        graph2.printPaths("Mu");


    }
}
```

5.2 ANALISIS DATA

```
public class Node {
    String name;
    boolean visited;
    int distance;
    EdgeList edges;
    Node next;
    Node parent;

    Node(String name) {
        this.name = name;
        this.visited = false;
        this.distance = Integer.MAX_VALUE;
        this.edges = new EdgeList();
        this.next = null;
        this.parent = null;
    }
}

class Edge {
    Node target;
    int weight;
    Edge next;

    Edge(Node target, int weight) {
        this.target = target;
        this.weight = weight;
        this.next = null;
    }
}

class EdgeList {
    Edge head;

    void addEdge(Edge edge) {
        if (head == null) {
            head = edge;
        } else {
            Edge current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = edge;
        }
    }
}
```

Script ini merupakan implementasi sederhana dari struktur data graf berbasis linked list untuk merepresentasikan simpul node dan sisi edge. Kelas Node menyimpan informasi

simpul, seperti nama, status kunjungan (visited), jarak terpendek (distance), daftar sisi yang terhubung (edges), simpul berikutnya dalam daftar (next), dan simpul induk dalam jalur terpendek (parent). Kelas Edge merepresentasikan sisi dengan informasi simpul tujuan (target), bobot (weight), dan sisi berikutnya dalam daftar (next). Kelas EdgeList adalah daftar berantai dari objek Edge yang menyediakan metode untuk menambahkan sisi baru ke akhir daftar. Struktur ini berguna untuk algoritma graf, seperti pencarian jalur terpendek menggunakan BFS, DFS, atau Dijkstra.

```
public class NodeList {
    Node head;

    void addNode(Node node) {
        if (head == null) {
            head = node;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = node;
        }
    }

    Node findNode(String name) {
        Node current = head;
        while (current != null) {
            if (current.name.equals(name)) {
                return current;
            }
            current = current.next;
        }
        return null;
    }
}
```

Script ini merupakan implementasi sederhana dari linked list untuk menyimpan dan mengelola node. Kelas NodeList memiliki atribut head, yang menunjuk ke simpul pertama dalam daftar. Metode addNode digunakan untuk menambahkan simpul baru ke akhir daftar, dengan melakukan iterasi hingga menemukan simpul terakhir. Metode findNode mencari simpul berdasarkan nama (name) dengan melakukan iterasi mulai dari simpul pertama hingga menemukan simpul yang cocok atau hingga akhir daftar, dan mengembalikan simpul tersebut jika ditemukan, atau null jika tidak.

```
class Edge {
    Node target;
    int weight;
    Edge next;

    Edge(Node target, int weight) {
```

```
        this.target = target;
        this.weight = weight;
        this.next = null;
    }
}
    }
```

Script ini merepresentasikan struktur data untuk sebuah edge dalam graf berbasis linked list. Kelas Edge memiliki tiga atribut yaitu target, yang menunjuk ke node tujuan yang terhubung oleh edge tersebut. weight, yang menyimpan bobot atau nilai yang terkait dengan edge tersebut dan next, yang menunjuk ke sisi berikutnya dalam daftar berantai. Konstruktor Edge digunakan untuk menginisialisasi sisi dengan simpul tujuan (target) dan bobot (weight), sementara next diatur ke null sebagai nilai awal.

```
public class Graph {
    NodeList nodes;
    String[][] paths = new String[100][100];
    int[] pathLengths = new int[100];
    int pathCount = 0;

    Graph() {
        nodes = new NodeList();
    }

    void addNode(String name) {
        nodes.addNode(new Node(name));
    }

    void addEdge(String from, String to, int weight) {
        Node fromNode = nodes.findNode(from);
        Node toNode = nodes.findNode(to);
        if (fromNode != null && toNode != null) {
            fromNode.edges.addEdge(new Edge(toNode, weight));
        }
    }
```

Script diatas merupakan implementasi awal dari sebuah graf yang mendukung representasi simpul node dan edge dengan kemampuan untuk mencatat jalur. Kelas Graph memiliki atribut nodes berisi daftar simpul menggunakan NodeList, array paths untuk menyimpan jalur sebagai pasangan nama simpul, array pathLengths untuk menyimpan panjang jalur, dan variabel pathCount untuk menghitung jumlah jalur yang tercatat. Konstruktor Graph menginisialisasi daftar simpul kosong. Metode addNode menambahkan simpul baru ke graf berdasarkan nama, sementara metode addEdge membuat edge baru yang menghubungkan dua simpul dengan bobot tertentu, menggunakan metode pencarian dari NodeList untuk mendapatkan simpul asal fromNode dan tujuan toNode.

```
void dijkstra(String startName) {
        Node startNode = nodes.findNode(startName);
        if (startNode == null) return;
```

```
        startNode.distance = 0;

        while (true) {
            Node currentNode = null;

            Node temp = nodes.head;
            while (temp != null) {
                if (!temp.visited && (currentNode == null ||
temp.distance < currentNode.distance)) {
                    currentNode = temp;
                }
                temp = temp.next;
            }

            if (currentNode == null) break;

            currentNode.visited = true;

            Edge edge = currentNode.edges.head;
            while (edge != null) {
                Node neighbor = edge.target;
                int newDistance = currentNode.distance + edge.weight;
                if (newDistance < neighbor.distance) {
                    neighbor.distance = newDistance;
                    neighbor.parent = currentNode;
                }
                edge = edge.next;
            }
        }
    }
```

Script diatas adalah metode dijkstra yang dimana mengimplementasikan algoritma Dijkstra untuk mencari jalur terpendek dari simpul awal ke semua simpul lainnya dalam sebuah graf berbobot. Metode ini dimulai dengan menginisialisasi jarak (distance) simpul awal menjadi 0, sedangkan simpul lainnya tetap pada nilai maksimum (Integer.MAX_VALUE). Dalam setiap iterasi, simpul yang belum dikunjungi dengan jarak terkecil dipilih sebagai currentNode. Setelah itu, semua tetangga (neighbor) simpul tersebut diperiksa melalui sisi-sisi yang terhubung. Jika ditemukan jarak baru yang lebih pendek ke tetangga tersebut, jarak tersebut diperbarui, dan simpul induk (parent) diatur ke simpul saat ini. Proses ini berulang hingga semua simpul telah dikunjungi atau tidak ada lagi simpul yang dapat diproses, memastikan setiap simpul memiliki jarak minimum dari simpul awal.

```
void collectPath(Node target) {
        String[] path = new String[100];
        int length = 0;

        while (target != null) {
            path[length++] = target.name;
            target = target.parent;
        }

        paths[pathCount] = new String[length];
        for (int i = 0; i < length; i++) {
            paths[pathCount][length - 1 - i] = path[i];
        }
        pathLengths[pathCount] = length;
        pathCount++;
    }

    void sortPaths() {
        for (int i = 0; i < pathCount - 1; i++) {
            for (int j = 0; j < pathCount - i - 1; j++) {
                if (pathLengths[j] > pathLengths[j + 1]) {

                    String[] tempPath = paths[j];
                    paths[j] = paths[j + 1];
                    paths[j + 1] = tempPath;

                    int tempLength = pathLengths[j];
                    pathLengths[j] = pathLengths[j + 1];
                    pathLengths[j + 1] = tempLength;
                }
            }
        }
    }
```

Script tersebut merupakan metode collectPath dan sortPaths digunakan untuk mengelola jalur-jalur yang ditemukan dalam graf. Metode collectPath membangun jalur dari simpul target (target) hingga simpul awal dengan mengikuti referensi induk (parent). Jalur tersebut disimpan secara terbalik dalam array sementara path, kemudian diurutkan ulang dan disimpan ke array paths bersama dengan panjang jalur di pathLengths. Metode sortPaths mengurutkan semua jalur berdasarkan panjangnya menggunakan bubble sort, sehingga jalur dengan panjang terpendek berada di awal.

```
void printPaths(String startName) {
        Node startNode = nodes.findNode(startName);
        if (startNode == null) return;

        Node current = nodes.head;
        while (current != null) {
            if (!current.name.equals(startName)) {
                collectPath(current);
            }
            current = current.next;
        }

        sortPaths();
```

```
        for (int i = 0; i < pathCount; i++) {
            System.out.print("");
            for (int j = 0; j < pathLengths[i]; j++) {
                System.out.print(paths[i][j]);
                if (j < pathLengths[i] - 1) {
                    System.out.print(" => ");
                }
            }
            System.out.println();
        }
    }
```

Script diatas merupakan metode printPaths yang digunakan untuk mencetak semua jalur dari simpul awal tertentu ke setiap simpul lain dalam graf. Pertama, metode mencari simpul awal berdasarkan namanya startName. Untuk setiap simpul dalam graf yang bukan simpul awal, metode collectPath dipanggil untuk membangun jalur dari simpul tersebut ke simpul awal. Setelah semua jalur dikumpulkan, metode sortPaths dipanggil untuk mengurutkan jalur berdasarkan panjangnya. Kemudian, setiap jalur dicetak ke konsol dalam format yang menunjukkan urutan simpul dengan pemisah " => ", memberikan representasi yang jelas dari jalur-jalur dalam graf.

```
public class Main {
    public static void main(String[] args) {
        Graph graph = new Graph();
        Graph graph1 = new Graph();
        Graph graph2 = new Graph();

        graph.addNode("Alpha");
        graph.addNode("Beta");
        graph.addNode("Gamma");
        graph.addNode("Delta");
        graph.addNode("Epsilon");
        graph.addNode("Zeta");
        graph.addNode("Eta");
        graph.addNode("Theta");
        graph.addNode("Iota");
        graph.addNode("Kappa");
        graph.addNode("Lambda");
        graph.addNode("Mu");
        graph.addNode("Nu");
        graph.addNode("Xi");
        graph.addNode("omnicron");
        graph.addNode("Sigma");
        graph.addNode("Omega");

        graph.addEdge("Alpha", "Beta", 8);
        graph.addEdge("Alpha", "Delta", 9);
        graph.addEdge("Alpha", "Eta", 10);
        graph.addEdge("Beta", "Alpha", 8);
        graph.addEdge("Beta", "Delta", 3);
        graph.addEdge("Beta", "Gamma", 7);
        graph.addEdge("Gamma", "Beta", 7);
        graph.addEdge("Gamma", "Delta", 17);
```

```
        graph.addEdge("Gamma", "Zeta", 8);
        graph.addEdge("Delta", "Alpha", 9);
        graph.addEdge("Delta", "Beta", 3);
        graph.addEdge("Delta", "Gamma", 17);
        graph.addEdge("Delta", "Epsilon", 4);
        graph.addEdge("Delta", "Theta", 7);
        graph.addEdge("Delta", "Eta", 2);
        graph.addEdge("Eta", "Alpha", 10);
        graph.addEdge("Eta", "Delta", 2);
        graph.addEdge("Eta", "Lambda", 9);
        graph.addEdge("Lambda", "Eta", 9);
        graph.addEdge("Lambda", "Mu", 8);
        graph.addEdge("Theta", "Delta", 7);
        graph.addEdge("Theta", "Epsilon", 4);
        graph.addEdge("Theta", "Iota", 3);
        graph.addEdge("Theta", "Kappa", 9);
        graph.addEdge("Mu", "Lambda", 8);
        graph.addEdge("Mu", "Kappa", 14);
        graph.addEdge("Mu", "Xi", 8);
        graph.addEdge("Xi", "Mu", 8);
        graph.addEdge("Xi", "omnicron", 5);
        graph.addEdge("omnicron", "Xi", 5);
        graph.addEdge("omnicron", "Sigma", 9);
        graph.addEdge("omnicron", "Nu", 9);
        graph.addEdge("Nu", "omnicron", 9);
        graph.addEdge("Nu", "Kappa", 9);
        graph.addEdge("Nu", "Omega", 10);
        graph.addEdge("Omega", "Nu", 10);
        graph.addEdge("Sigma", "omnicron", 9);
        graph.addEdge("Kappa", "Theta", 9);
        graph.addEdge("Kappa", "Mu", 14);
        graph.addEdge("Kappa", "Nu", 9);
        graph.addEdge("Kappa", "Iota", 5);
        graph.addEdge("Epsilon", "Delta", 4);
        graph.addEdge("Epsilon", "Theta", 4);
        graph.addEdge("Epsilon", "Zeta", 9);
        graph.addEdge("Zeta", "Epsilon", 9);
        graph.addEdge("Zeta", "Gamma", 8);
        graph.addEdge("Zeta", "Iota", 10);
        graph.addEdge("Iota", "Zeta", 10);
        graph.addEdge("Iota", "Theta", 3);
        graph.addEdge("Iota", "Kappa", 5);

        graph.dijkstra("Alpha");
        graph.printPaths("Alpha");
```

Script diatas adalah Kelas Main yang mendemonstrasikan pembuatan objek Graph dan menambahkan simpul seperti "Alpha", "Beta", hingga "Omega". Selanjutnya, hubungan antar simpul ditentukan menggunakan metode addEdge, di mana setiap sisi memiliki bobot tertentu yang merepresentasikan jarak atau biaya. Setelah graf terbentuk, algoritma Dijkstra dipanggil melalui Dijkstra ("Alpha") untuk menghitung jalur terpendek dari simpul "Alpha" ke semua simpul lain. Hasil jalur terpendek kemudian ditampilkan menggunakan metode printPaths. Program ini mendemonstrasikan bagaimana graf berarah berbobot dikelola dan algoritma Dijkstra diterapkan untuk analisis jalur.