Great! You're building a solid foundation by working with all three common **graph representations in Java**. Below are well-structured, clean **template codes** for:

1. **Adjacency List (Unweighted & Weighted)**
2. **Adjacency Matrix (Weighted)**
3. **Edge List (Bonus)**
4. Example representation for each

**🔷 1. Adjacency List – Unweighted Graph**

**✅ Template Code**

java

CopyEdit

import java.util.\*;

public class AdjacencyListGraph {

public static void addEdge(List<List<Integer>> adj, int u, int v) {

adj.get(u).add(v);

adj.get(v).add(u); // Undirected

}

public static void displayGraph(List<List<Integer>> adj) {

for (int i = 0; i < adj.size(); i++) {

System.out.print(i + ": ");

for (int neighbor : adj.get(i)) {

System.out.print(neighbor + " ");

}

System.out.println();

}

}

public static void main(String[] args) {

int V = 4;

List<List<Integer>> adj = new ArrayList<>();

for (int i = 0; i < V; i++) adj.add(new ArrayList<>());

addEdge(adj, 0, 1);

addEdge(adj, 0, 2);

addEdge(adj, 1, 2);

addEdge(adj, 2, 3);

displayGraph(adj);

}

}

**🧠 Output Representation**

makefile

CopyEdit

0: 1 2

1: 0 2

2: 0 1 3

3: 2

**🔷 2. Adjacency List – Weighted Graph**

**✅ Template Code**

java

CopyEdit

import java.util.\*;

public class WeightedAdjacencyListGraph {

static class Edge {

int dest, weight;

Edge(int dest, int weight) {

this.dest = dest;

this.weight = weight;

}

public String toString() {

return "(" + dest + ", " + weight + ")";

}

}

public static void addEdge(List<List<Edge>> adj, int u, int v, int w) {

adj.get(u).add(new Edge(v, w));

adj.get(v).add(new Edge(u, w)); // Undirected

}

public static void displayGraph(List<List<Edge>> adj) {

for (int i = 0; i < adj.size(); i++) {

System.out.print(i + ": ");

for (Edge edge : adj.get(i)) {

System.out.print(edge + " ");

}

System.out.println();

}

}

public static void main(String[] args) {

int V = 4;

List<List<Edge>> adj = new ArrayList<>();

for (int i = 0; i < V; i++) adj.add(new ArrayList<>());

addEdge(adj, 0, 1, 10);

addEdge(adj, 0, 2, 5);

addEdge(adj, 1, 2, 2);

addEdge(adj, 1, 3, 1);

addEdge(adj, 2, 3, 9);

displayGraph(adj);

}

}

**🧠 Output Representation**

makefile

CopyEdit

0: (1, 10) (2, 5)

1: (0, 10) (2, 2) (3, 1)

2: (0, 5) (1, 2) (3, 9)

3: (1, 1) (2, 9)

**🔷 3. Adjacency Matrix – Weighted Graph**

**✅ Template Code**

java

CopyEdit

public class AdjacencyMatrixGraph {

public static void addEdge(int[][] matrix, int u, int v, int weight) {

matrix[u][v] = weight;

matrix[v][u] = weight; // Undirected

}

public static void displayMatrix(int[][] matrix) {

System.out.println("Adjacency Matrix:");

for (int[] row : matrix) {

for (int val : row) {

System.out.print(val + "\t");

}

System.out.println();

}

}

public static void main(String[] args) {

int V = 4;

int[][] matrix = new int[V][V];

addEdge(matrix, 0, 1, 10);

addEdge(matrix, 0, 2, 5);

addEdge(matrix, 1, 2, 2);

addEdge(matrix, 1, 3, 1);

addEdge(matrix, 2, 3, 9);

displayMatrix(matrix);

}

}

**🧠 Output Representation**

yaml

CopyEdit

Adjacency Matrix:

0 10 5 0

10 0 2 1

5 2 0 9

0 1 9 0

**🔷 4. Edge List – Weighted Graph (Bonus for Kruskal’s Algorithm)**

**✅ Template Code**

java

CopyEdit

import java.util.\*;

public class EdgeListGraph {

static class Edge {

int u, v, weight;

Edge(int u, int v, int w) {

this.u = u;

this.v = v;

this.weight = w;

}

public String toString() {

return "(" + u + "-" + v + ", " + weight + ")";

}

}

public static void main(String[] args) {

List<Edge> edges = new ArrayList<>();

edges.add(new Edge(0, 1, 10));

edges.add(new Edge(0, 2, 5));

edges.add(new Edge(1, 2, 2));

edges.add(new Edge(1, 3, 1));

edges.add(new Edge(2, 3, 9));

for (Edge e : edges) {

System.out.println(e);

}

}

}

**🧠 Output Representation**

scss

CopyEdit

(0-1, 10)

(0-2, 5)

(1-2, 2)

(1-3, 1)

(2-3, 9)

**✅ Final Recommendation:**

| **Type** | **Use Case** | **Representation** |
| --- | --- | --- |
| Most Common | Traversal, DFS/BFS | **Adjacency List** |
| Weighted Graphs | Dijkstra/Prim | **Adjacency List (with weights)** |
| Dense Graphs | Floyd-Warshall or quick edge lookup | **Adjacency Matrix** |
| MST (Kruskal) | Edge Sorting | **Edge List** |

Let me know if you want templates for **directed graphs**, **topological sort**, or **graph traversal (BFS/DFS)** based on these structures

**🔍 Edge List vs Adjacency List — Full Comparison**

| **Feature** | **Edge List** | **Adjacency List** |
| --- | --- | --- |
| 🔗 **What it stores** | A list of all edges (u, v, weight) | A list of neighbors for each node |
| 💾 **Structure** | List<Edge> | List<List<Edge>> (or Map<Node, List<Node>>) |
| 📊 **Space Complexity** | O(E) | O(V + E) |
| 📥 **Best for** | Simple graphs, quick iteration over edges | Efficient neighbor lookups, sparse graphs |
| 🔍 **Neighbor Lookup** | ❌ O(E), inefficient | ✅ O(1) if using array |
| 📉 **Suitable for Algorithms** | Kruskal's (MST), sorting by weight | Dijkstra’s, BFS, DFS |
| ➕ **Add edge** | edges.add(new Edge(u, v, w)) | adj[u].add(v) |
| 🌐 **Graph Type Support** | Easy for directed/undirected/weighted | Also supports all types |

**✅ Example Graph**

Let’s take this **weighted directed graph**:

scss

CopyEdit

0 → 1 (10)

0 → 2 (5)

1 → 3 (1)

2 → 3 (9)

**1️⃣ Edge List Representation**

java

CopyEdit

class Edge {

int from, to, weight;

Edge(int from, int to, int weight) {

this.from = from;

this.to = to;

this.weight = weight;

}

}

List<Edge> edgeList = new ArrayList<>();

edgeList.add(new Edge(0, 1, 10));

edgeList.add(new Edge(0, 2, 5));

edgeList.add(new Edge(1, 3, 1));

edgeList.add(new Edge(2, 3, 9));

✅ Best when:

* You want to loop through all edges
* You need to **sort by weight** (e.g., in **Kruskal's Algorithm**)

**2️⃣ Adjacency List Representation**

java

CopyEdit

class Edge {

int to, weight;

Edge(int to, int weight) {

this.to = to;

this.weight = weight;

}

}

List<List<Edge>> adjList = new ArrayList<>();

for (int i = 0; i < 4; i++) adjList.add(new ArrayList<>());

adjList.get(0).add(new Edge(1, 10));

adjList.get(0).add(new Edge(2, 5));

adjList.get(1).add(new Edge(3, 1));

adjList.get(2).add(new Edge(3, 9));

✅ Best when:

* You want to find neighbors of a node quickly (like in **Dijkstra**, **DFS**, **BFS**)
* Graph is **sparse**

**🚀 Which One Should You Use?**

| **You want to…** | **Use this** |
| --- | --- |
| Loop through all edges | Edge List |
| Sort edges by weight (e.g., Kruskal) | Edge List |
| Traverse neighbors (e.g., DFS/BFS) | Adjacency List |
| Run Dijkstra or Prim efficiently | Adjacency List |
| Use less memory on sparse graphs | Adjacency List |
| Model graph simply without traversal | Edge List |

**Want to see Adjacency Matrix vs these too? Or build from an edge list to adjacency list? I can show that as well.**