CINEC CAMPUS
Beyond A Graduate

BSc (Hons) in Software Engineering

IT3216: Graph Theory

Week 01

Fundamental concepts of graphs

Learning Outcomes

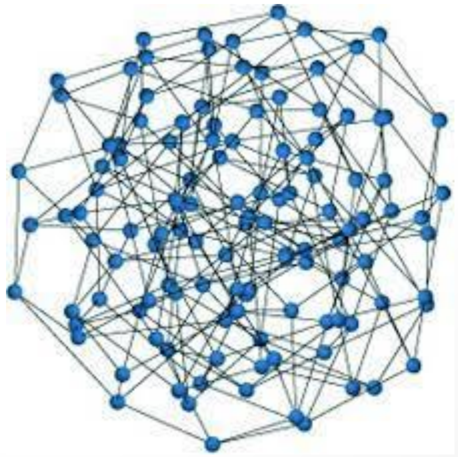
- After completing this course and the essential reading and activities, student should be able to,
- LO1: Apply basic notations of graph theory and fundamental theorems in graph theory
- LO2: formulate and prove central theorems about trees, matching, connectivity, coloring and planar graphs;
- LO3: describe and apply some basic algorithms for graphs;
- LO4: use graph theory as a modelling tool.

Aims of the Lesson

- To introduce the fundamental graph theory topics and results.
- To be familiarize with the techniques for proofs and analysis.

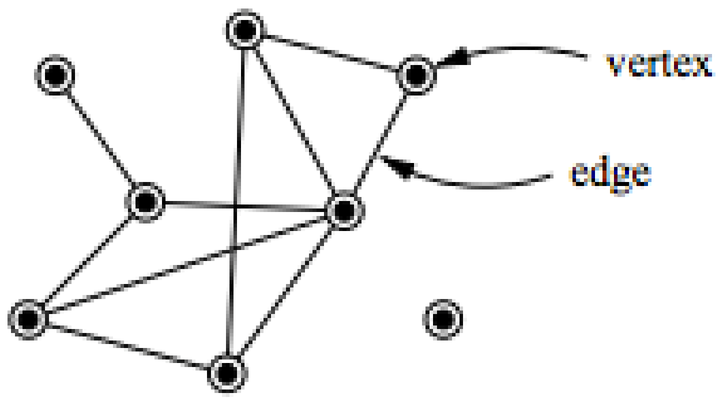
Graph theory

- In computer science, graph theory is the study of graphs, a mathematical structure used to model pair wise relations between objects from a certain collection.
- A graph in this context refers to a collection of vertices or nodes and a collection of edges that connect pairs of vertices

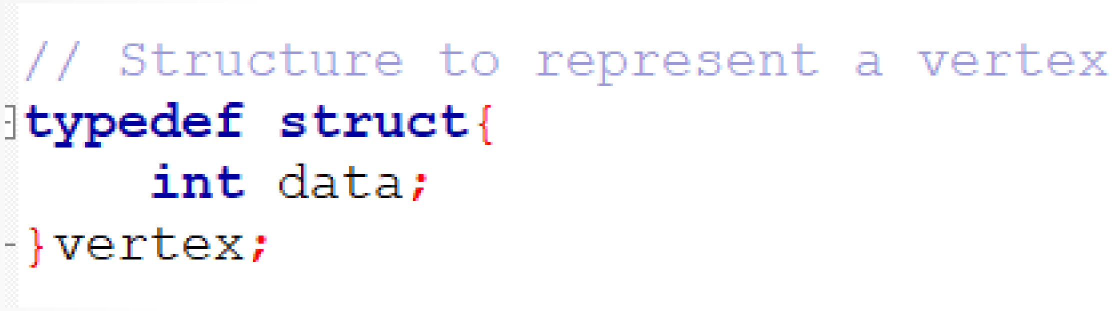


Vertices and Edges

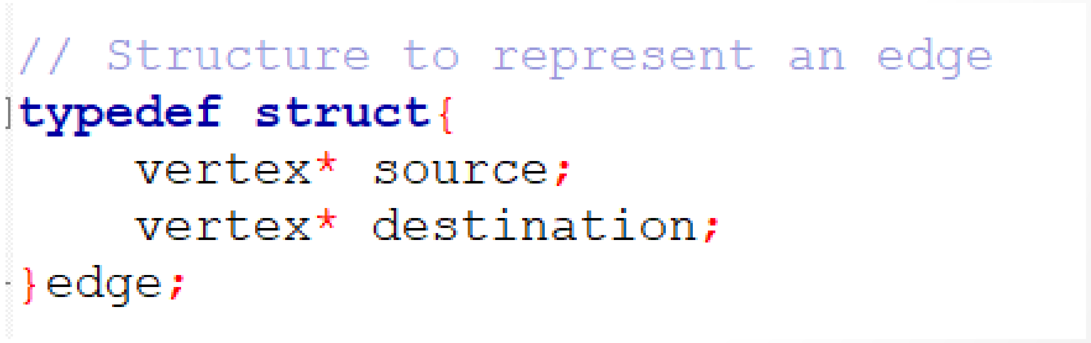
- A graph consists of vertices (also known as nodes) and edges (also known as links).
- Vertices represent entities or points, while edges represent connections between vertices.



Vertices and Edges



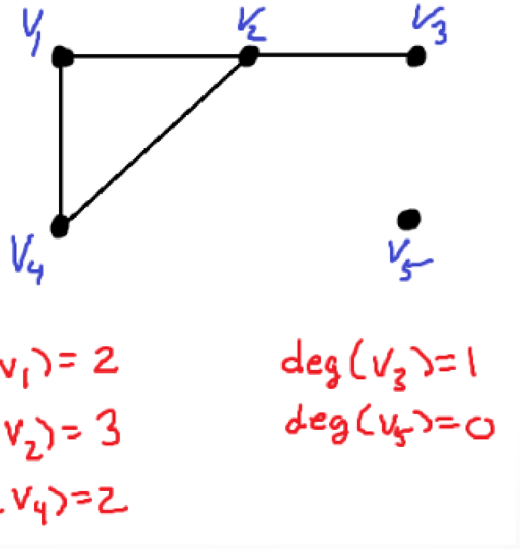
```
// Structure to represent a vertex
typedef struct{
    int data;
}vertex;
```



```
// Structure to represent an edge
typedef struct{
    vertex* source;
    vertex* destination;
}edge;
```

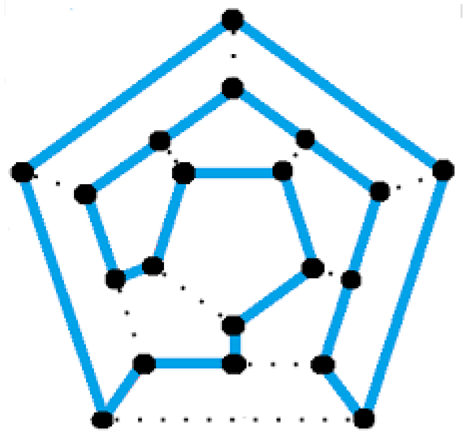
Degree of a Vertex

- The degree of a vertex in an undirected graph is the number of edges connected to it.
- In a directed graph, the in-degree of a vertex is the number of edges pointing to it, and the out-degree is the number of edges pointing from it.



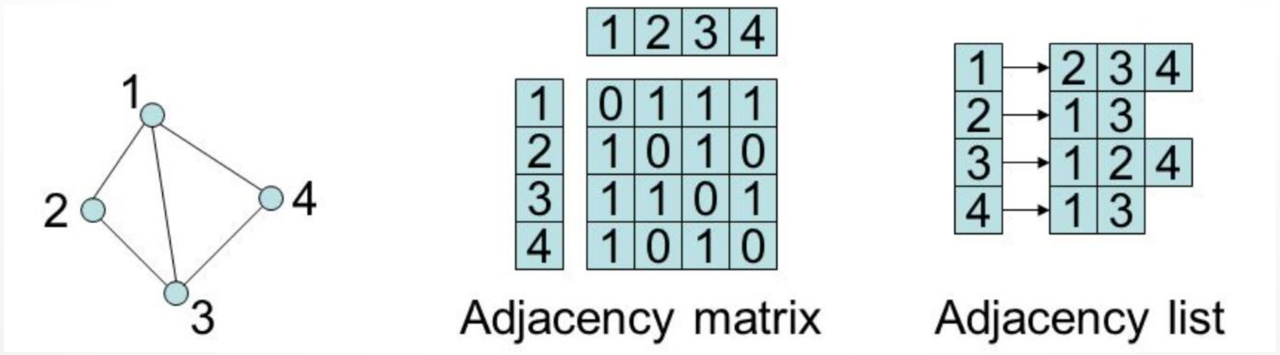
Path and Cycle

- A path in a graph is a sequence of vertices where each adjacent pair is connected by an edge.
- A cycle is a path where the first and last vertices are the same.

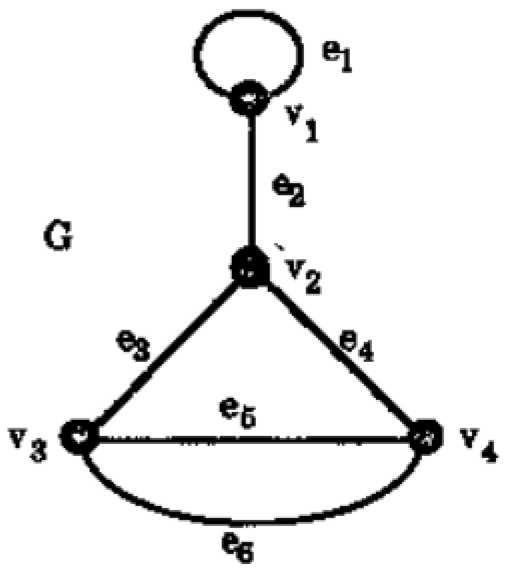


Adjacency Matrix and Adjacency List

- Two common ways to represent graphs are using an adjacency matrix (a 2D array) and an adjacency list (an array of lists or other data structures).
- Adjacency matrix indicates whether an edge exists between vertices, while an adjacency list lists the neighboring vertices for each vertex.



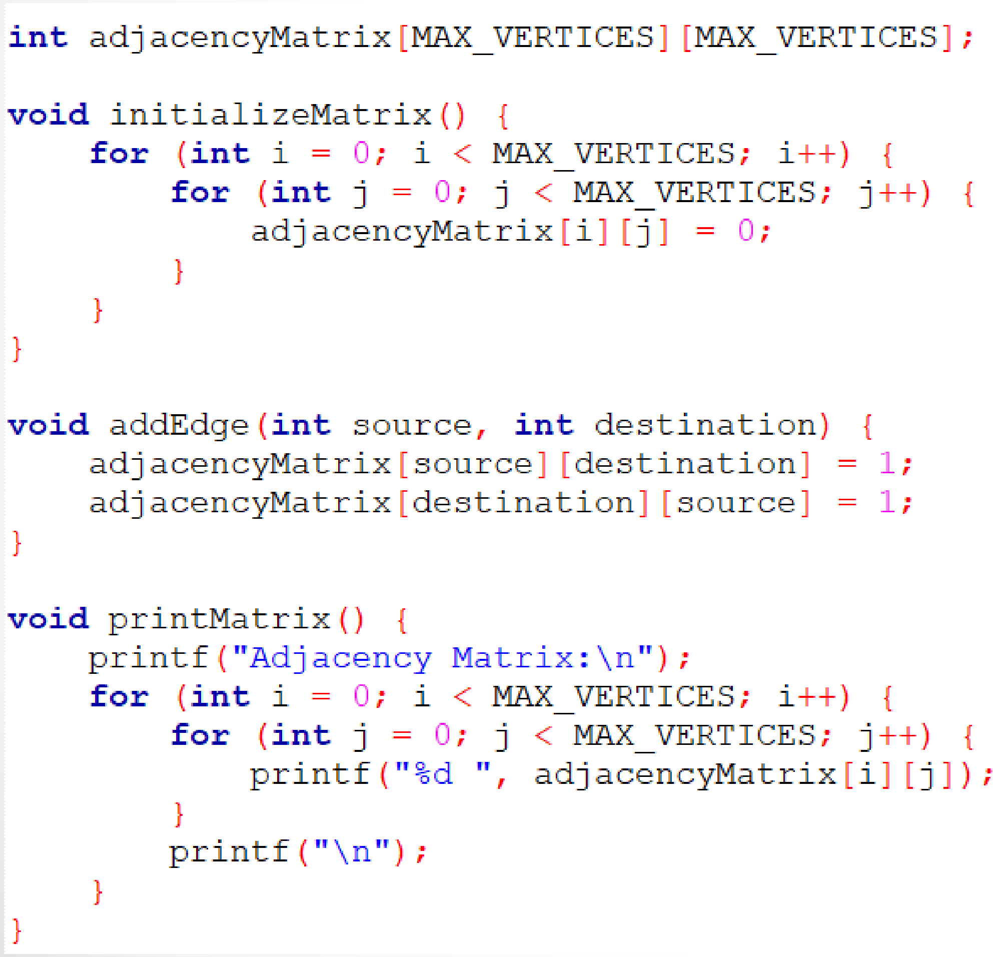
Adjacency Matrix



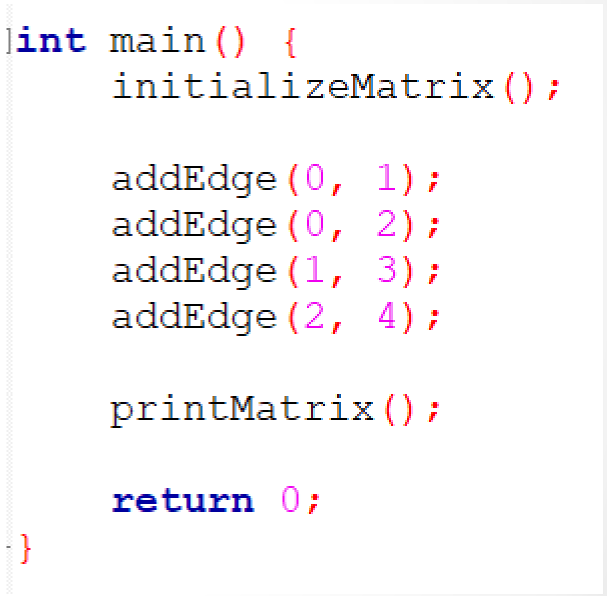
$A(G)$: a 4×4 matrix.

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Adjacency Matrix



```c
int adjacencyMatrix[MAX_VERTICES][MAX_VERTICES];

void initializeMatrix() {
    for (int i = 0; i < MAX_VERTICES; i++) {
        for (int j = 0; j < MAX_VERTICES; j++) {
            adjacencyMatrix[i][j] = 0;
        }
    }
}

void addEdge(int source, int destination) {
    adjacencyMatrix[source][destination] = 1;
    adjacencyMatrix[destination][source] = 1;
}

void printMatrix() {
    printf("Adjacency Matrix:\n");
    for (int i = 0; i < MAX_VERTICES; i++) {
        for (int j = 0; j < MAX_VERTICES; j++) {
            printf("%d ", adjacencyMatrix[i][j]);
        }
        printf("\n");
    }
}
```

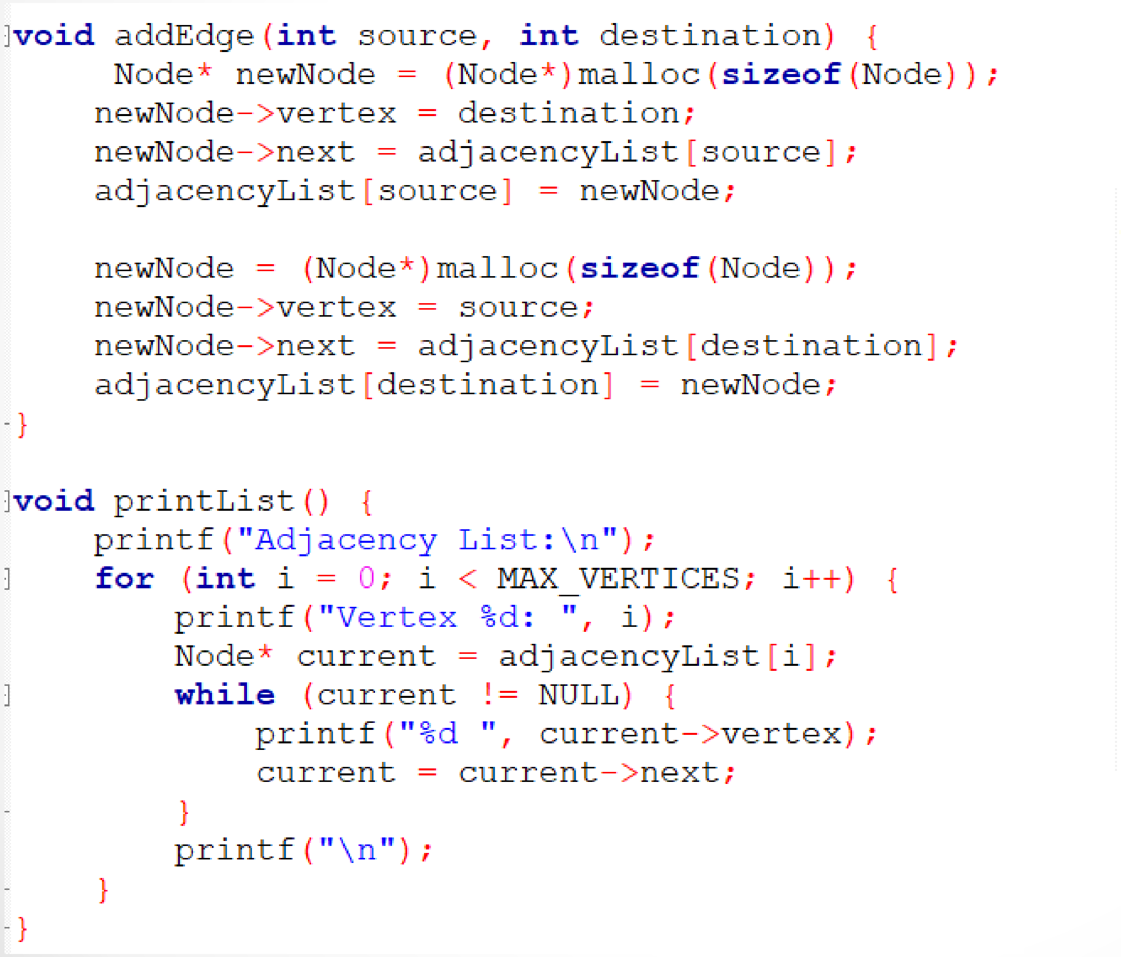


```c
int main() {
    initializeMatrix();

    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 3);
    addEdge(2, 4);

    printMatrix();

    return 0;
}
```

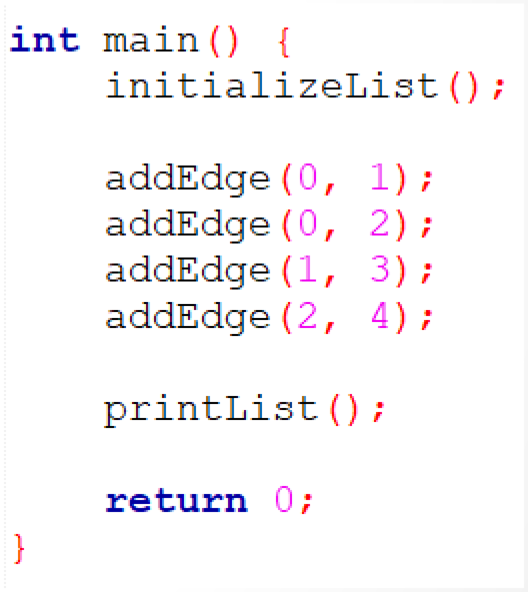
Adjacency List

```
#define MAX_VERTICES 5

typedef struct node {
    int vertex;
    struct node* next;
}Node;

Node* adjacencyList[MAX_VERTICES];

void initializeList() {
    for (int i = 0; i < MAX_VERTICES; i++) {
        adjacencyList[i] = NULL;
    }
}
```

Adjacency List



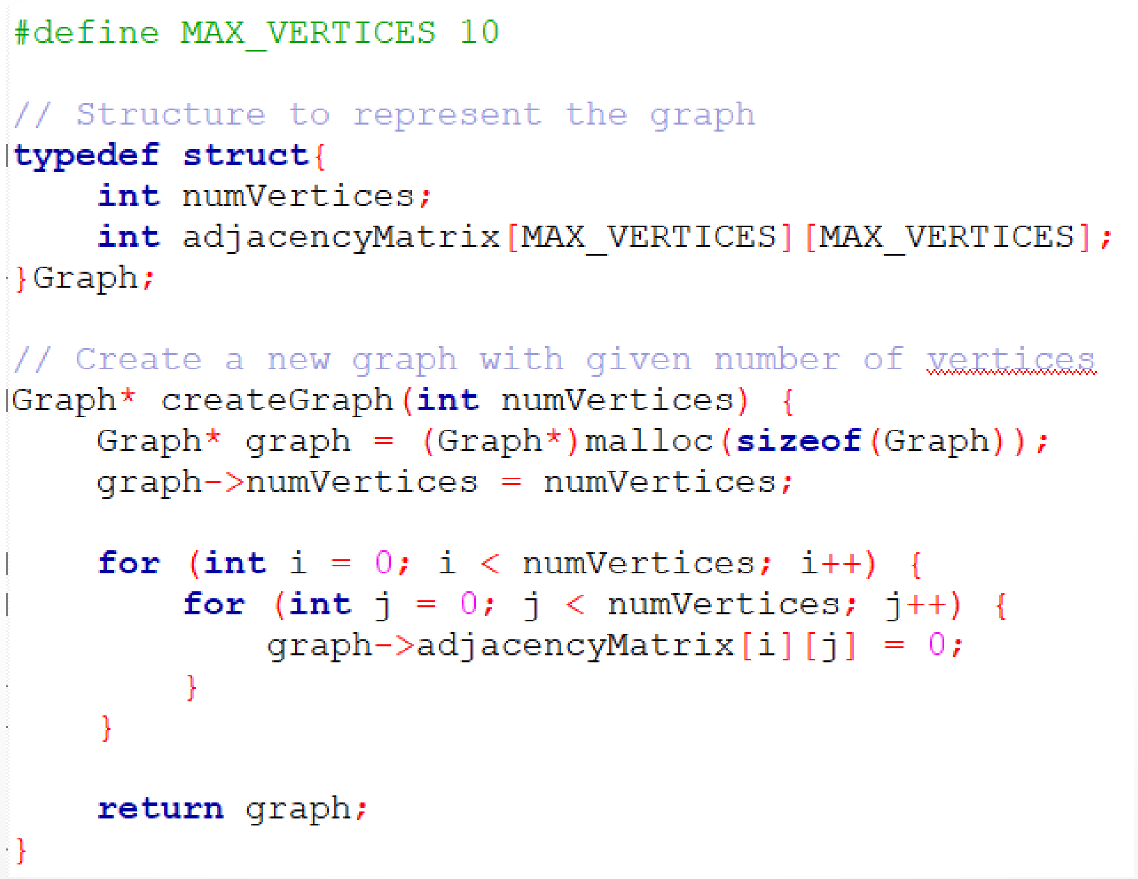
```
void addEdge(int source, int destination) {
     Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = destination;
    newNode->next = adjacencyList[source];
    adjacencyList[source] = newNode;

    newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = source;
    newNode->next = adjacencyList[destination];
    adjacencyList[destination] = newNode;
}

void printList() {
    printf("Adjacency List:\n");
    for (int i = 0; i < MAX_VERTICES; i++) {
        printf("Vertex %d: ", i);
        Node* current = adjacencyList[i];
        while (current != NULL) {
            printf("%d ", current->vertex);
            current = current->next;
        }
        printf("\n");
    }
}
```



```
int main() {
    initializeList();

    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 3);
    addEdge(2, 4);

    printList();

    return 0;
}
```

Graphs

- With use of **Adjacency Matrix**



```
#define MAX_VERTICES 10

// Structure to represent the graph
typedef struct{
    int numVertices;
    int adjacencyMatrix[MAX_VERTICES][MAX_VERTICES];
}Graph;

// Create a new graph with given number of vertices
Graph* createGraph(int numVertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = numVertices;

    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            graph->adjacencyMatrix[i][j] = 0;
        }
    }

    return graph;
}
```

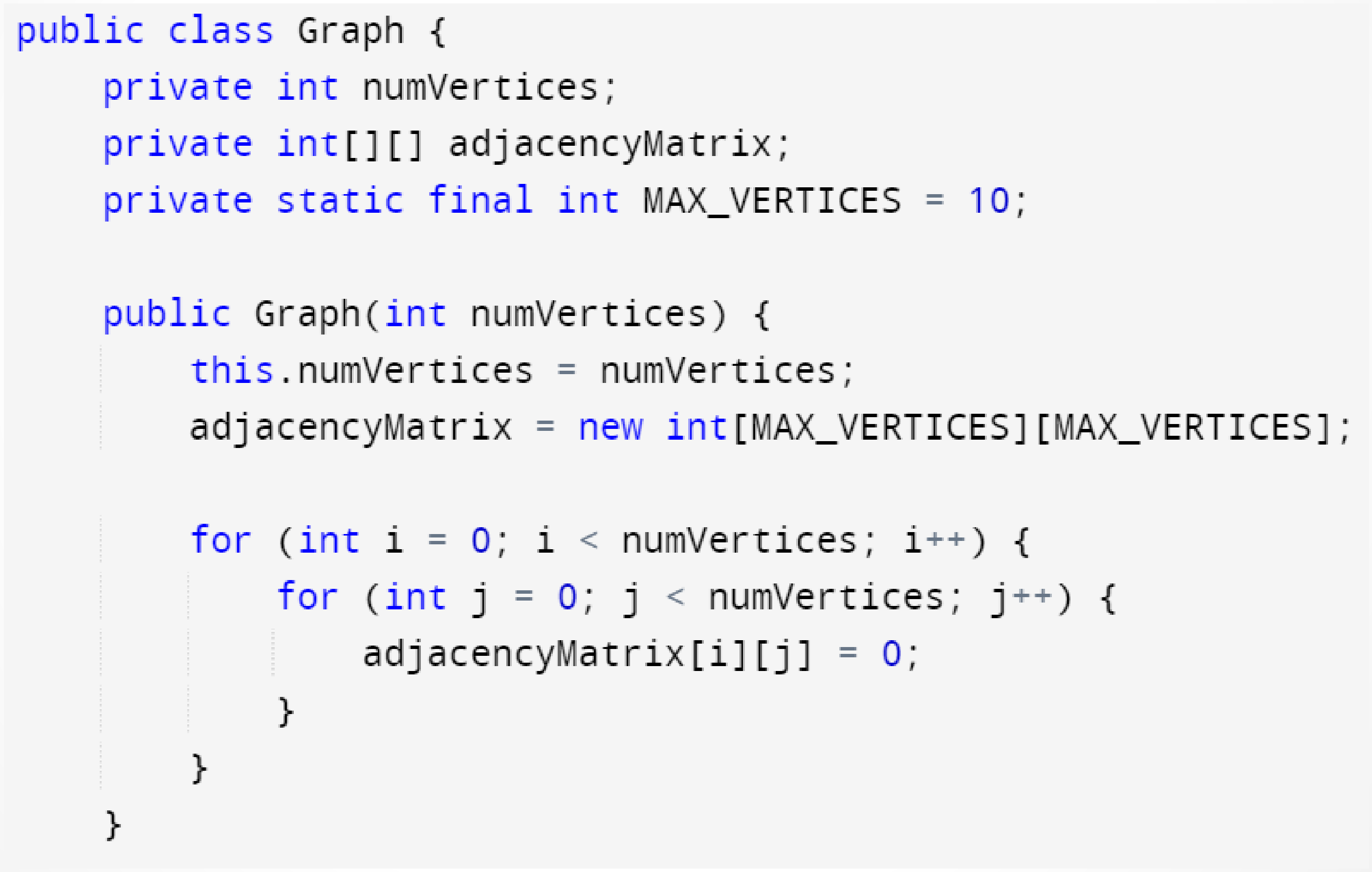
Graphs

```c
// Add an edge to the graph
void addEdge(Graph* graph, int src, int dest) {
    if (src >= 0 && src < graph->numVertices && dest >= 0 && dest < graph->numVertices) {
        graph->adjacencyMatrix[src][dest] = 1;
        graph->adjacencyMatrix[dest][src] = 1;
    } else {
        printf("Invalid vertex indices!\n");
    }
}

// Print the adjacency matrix representation of the graph
void printGraph(Graph* graph) {
    printf("Adjacency Matrix:\n");
    for (int i = 0; i < graph->numVertices; i++) {
        for (int j = 0; j < graph->numVertices; j++) {
            printf("%d ", graph->adjacencyMatrix[i][j]);
        }
        printf("\n");
    }
}
```

Graphs

```c
int main() {
    int numVertices = 5;
    struct Graph* graph = createGraph(numVertices);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    printGraph(graph);

    // Clean up memory
    free(graph);

    return 0;
}
```

Graphs

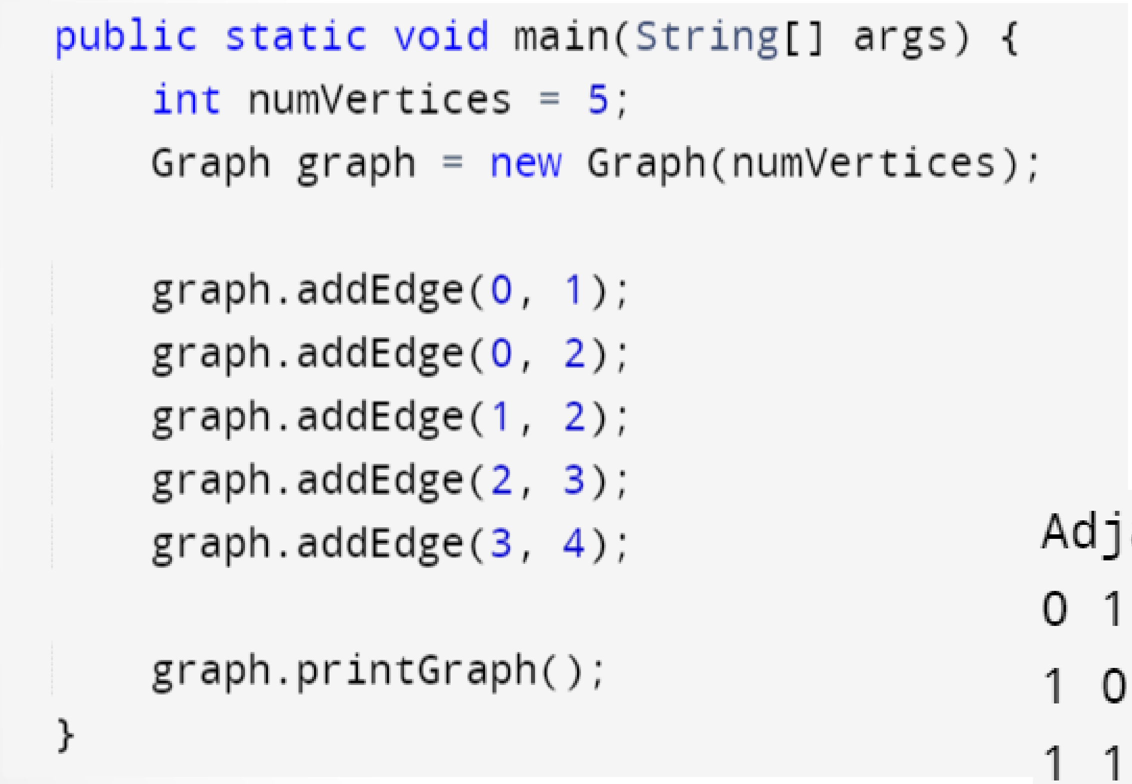


```java
public class Graph {
    private int numVertices;
    private int[][] adjacencyMatrix;
    private static final int MAX_VERTICES = 10;

    public Graph(int numVertices) {
        this.numVertices = numVertices;
        adjacencyMatrix = new int[MAX_VERTICES][MAX_VERTICES];

        for (int i = 0; i < numVertices; i++) {
            for (int j = 0; j < numVertices; j++) {
                adjacencyMatrix[i][j] = 0;
            }
        }
    }
```

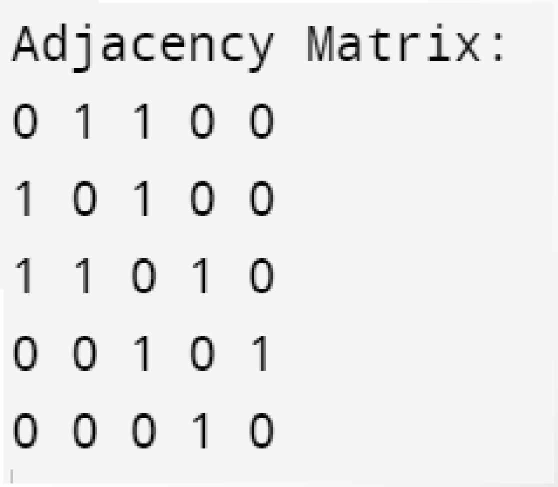
Graphs

```java
public void addEdge(int src, int dest) {
    if (src >= 0 && src < numVertices && dest >= 0 && dest
        < numVertices) {
        adjacencyMatrix[src][dest] = 1;
        adjacencyMatrix[dest][src] = 1;
    } else {
        System.out.println("Invalid vertex indices!");
    }
}

public void printGraph() {
    System.out.println("Adjacency Matrix:");
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            System.out.print(adjacencyMatrix[i][j] + " ");
        }
        System.out.println();
    }
}
```

Graphs



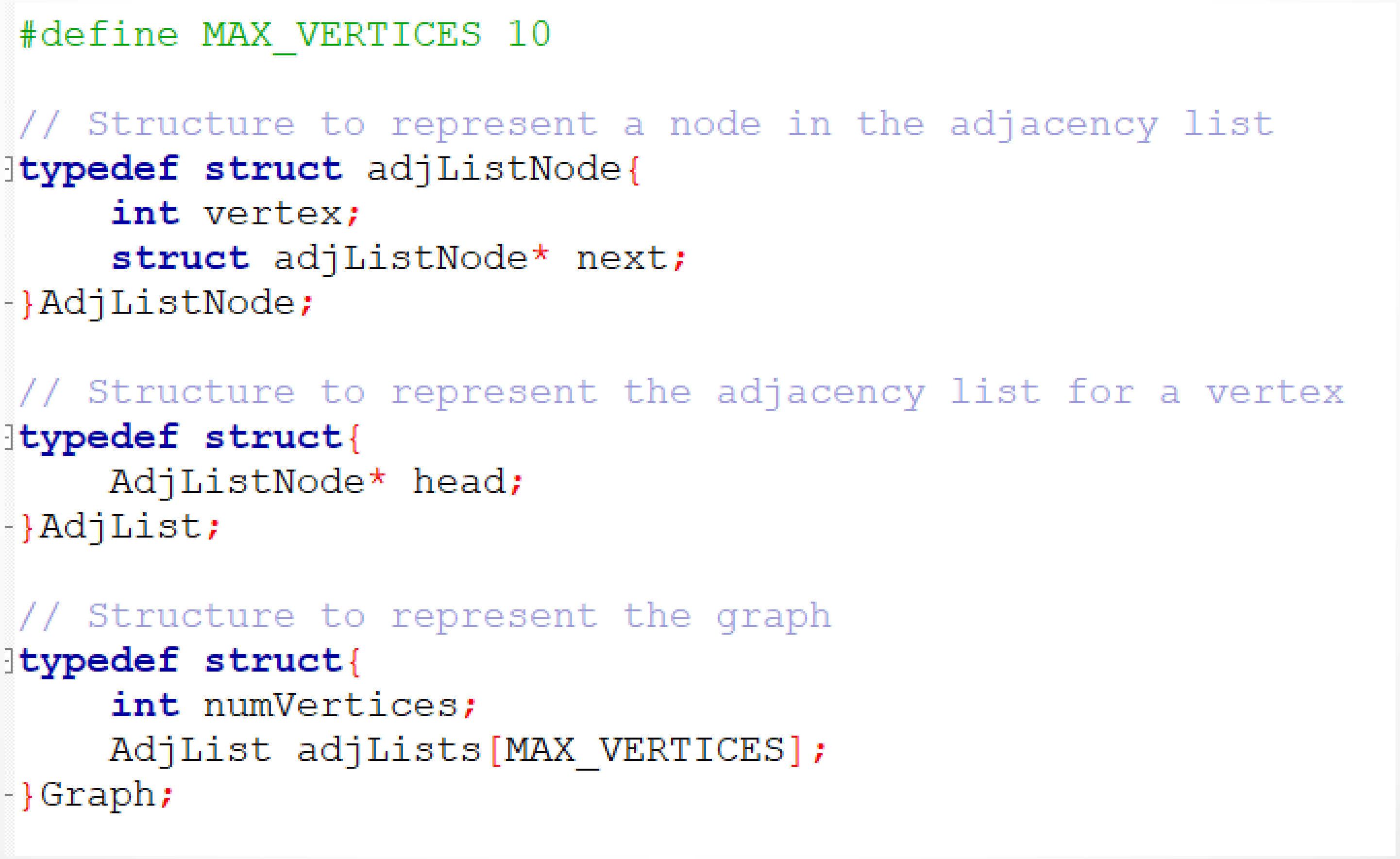
```
public static void main(String[] args) {
    int numVertices = 5;
    Graph graph = new Graph(numVertices);

    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 2);
    graph.addEdge(2, 3);
    graph.addEdge(3, 4);

    graph.printGraph();
}
```



```
Adjacency Matrix:
0 1 1 0 0
1 0 1 0 0
1 1 0 1 0
0 0 1 0 1
0 0 0 1 0
```

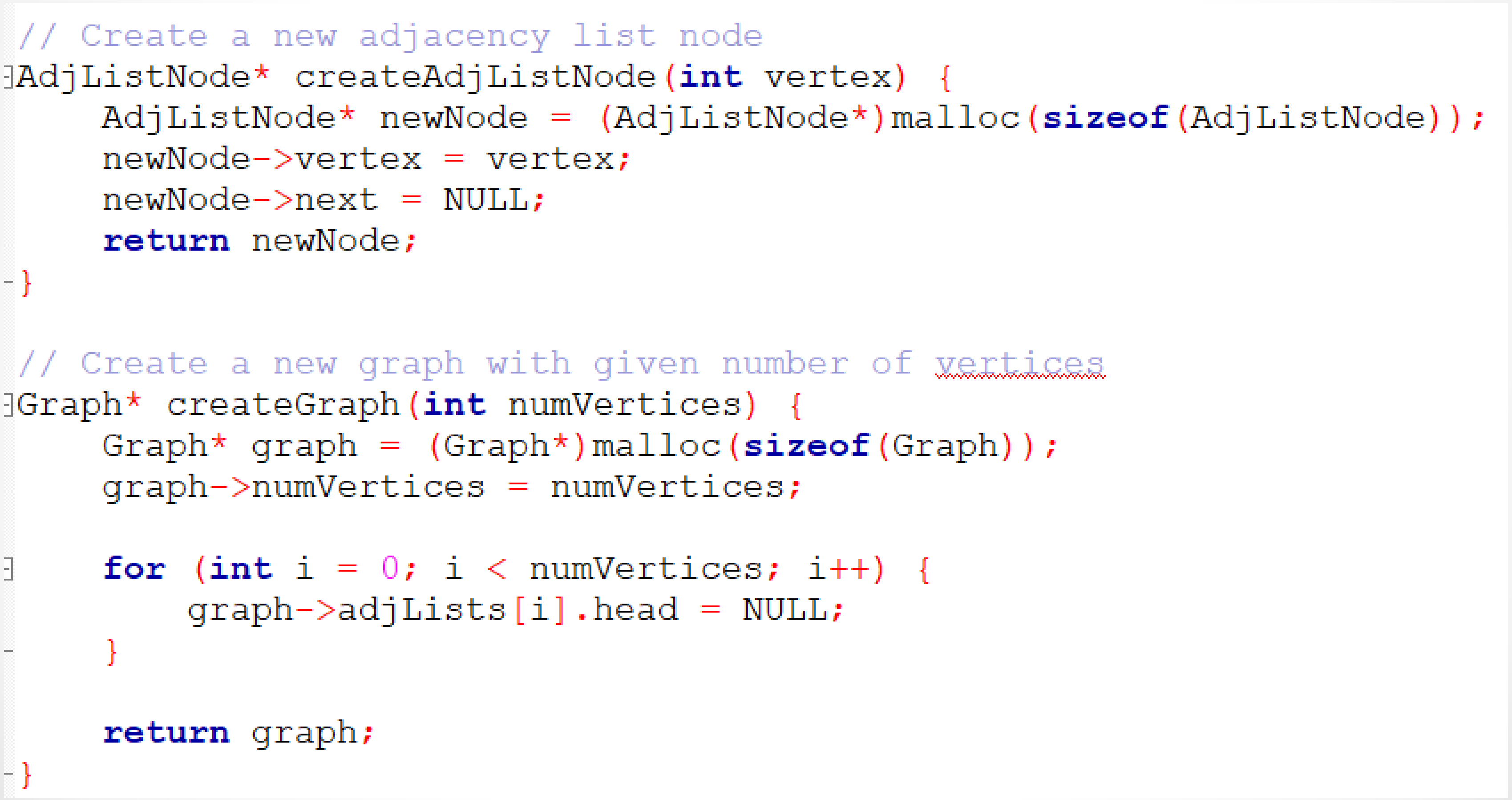
Graphs

- With use of **Adjacency List**



```
#define MAX_VERTICES 10

// Structure to represent a node in the adjacency list
typedef struct adjListNode{
    int vertex;
    struct adjListNode* next;
}AdjListNode;

// Structure to represent the adjacency list for a vertex
typedef struct{
    AdjListNode* head;
}AdjList;

// Structure to represent the graph
typedef struct{
    int numVertices;
    AdjList adjLists[MAX_VERTICES];
}Graph;
```

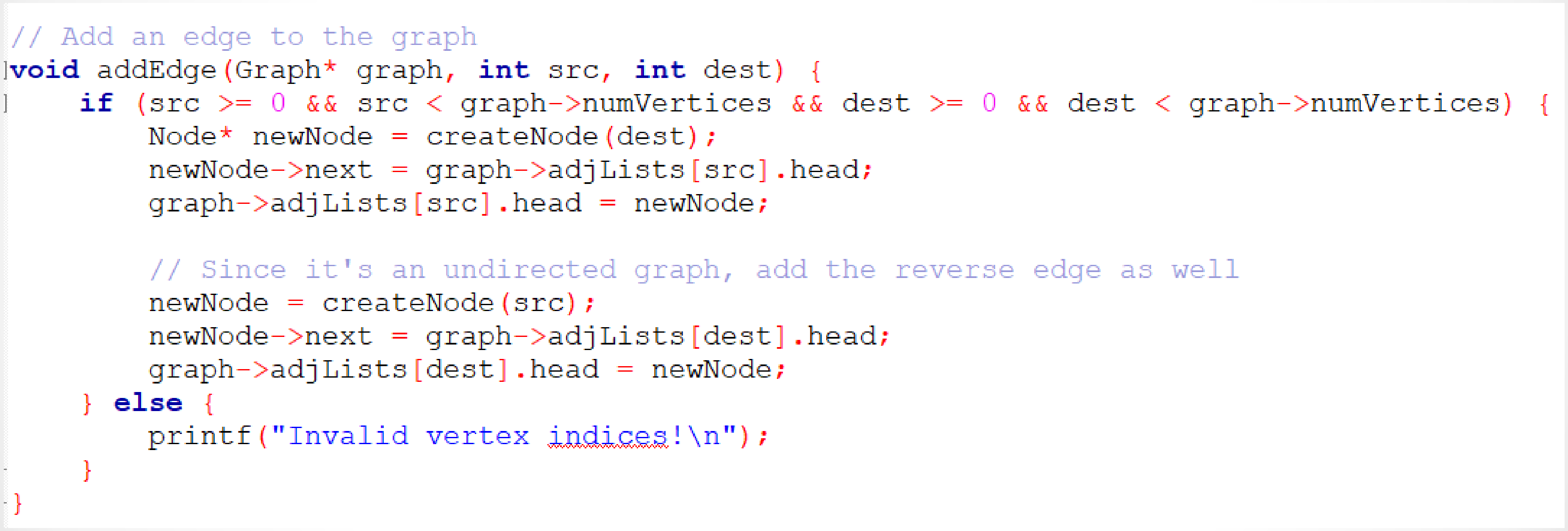
Graphs



```
// Create a new adjacency list node
AdjListNode* createAdjListNode(int vertex) {
    AdjListNode* newNode = (AdjListNode*)malloc(sizeof(AdjListNode));
    newNode->vertex = vertex;
    newNode->next = NULL;
    return newNode;
}

// Create a new graph with given number of vertices
Graph* createGraph(int numVertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = numVertices;

    for (int i = 0; i < numVertices; i++) {
        graph->adjLists[i].head = NULL;
    }

    return graph;
}
```

Graphs

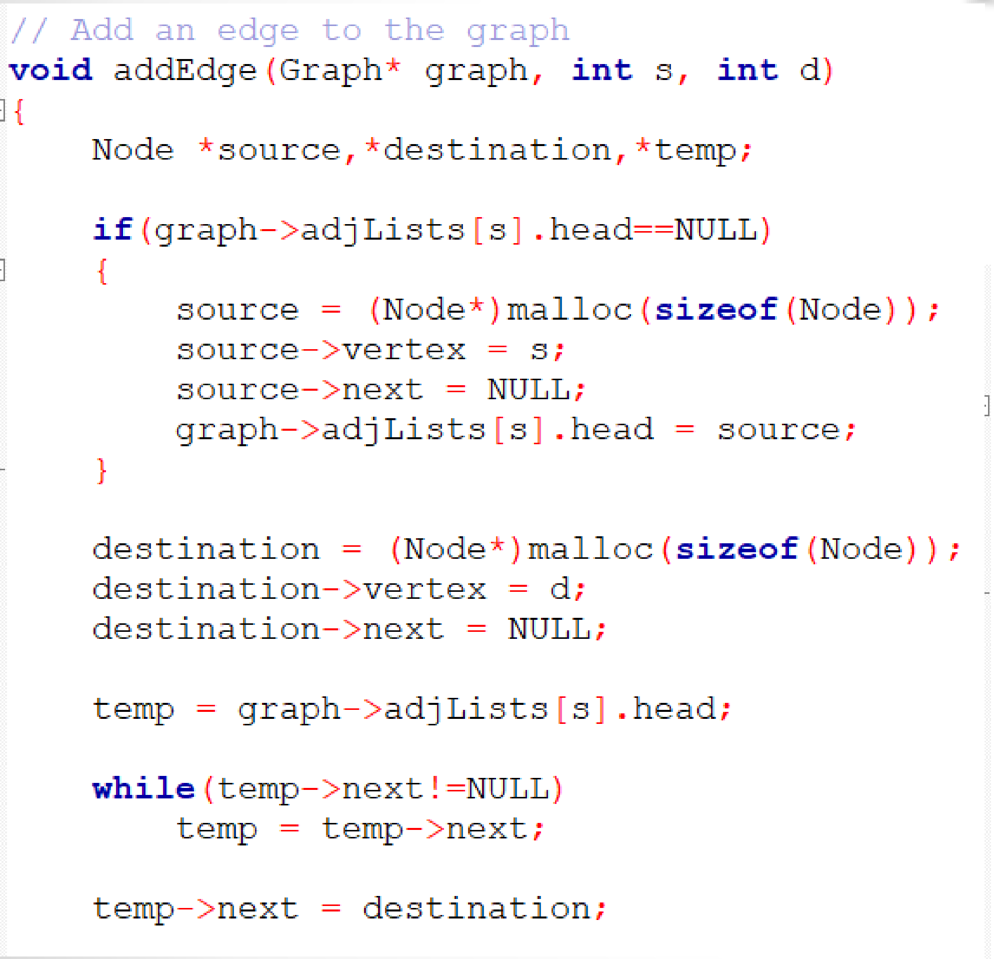
- Here is a one way to add an edge.



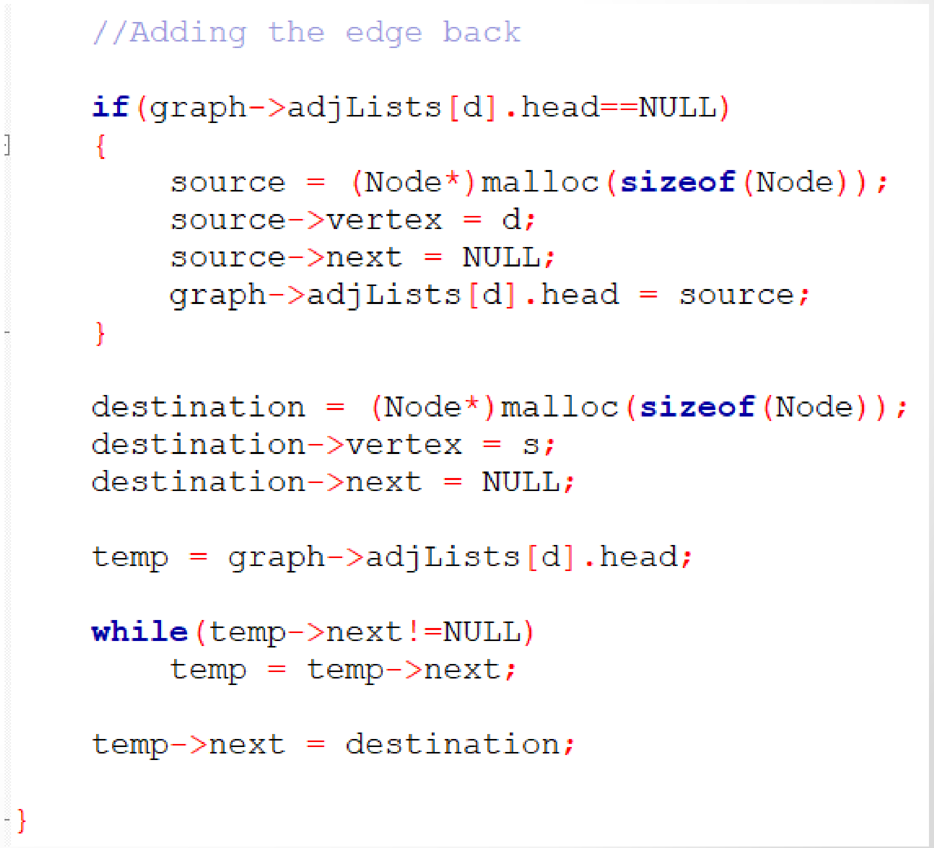
```c
// Add an edge to the graph
void addEdge(Graph* graph, int src, int dest) {
    if (src >= 0 && src < graph->numVertices && dest >= 0 && dest < graph->numVertices) {
        Node* newNode = createNode(dest);
        newNode->next = graph->adjLists[src].head;
        graph->adjLists[src].head = newNode;

        // Since it's an undirected graph, add the reverse edge as well
        newNode = createNode(src);
        newNode->next = graph->adjLists[dest].head;
        graph->adjLists[dest].head = newNode;
    } else {
        printf("Invalid vertex indices!\n");
    }
}
```

- And there is another way too…

Graphs

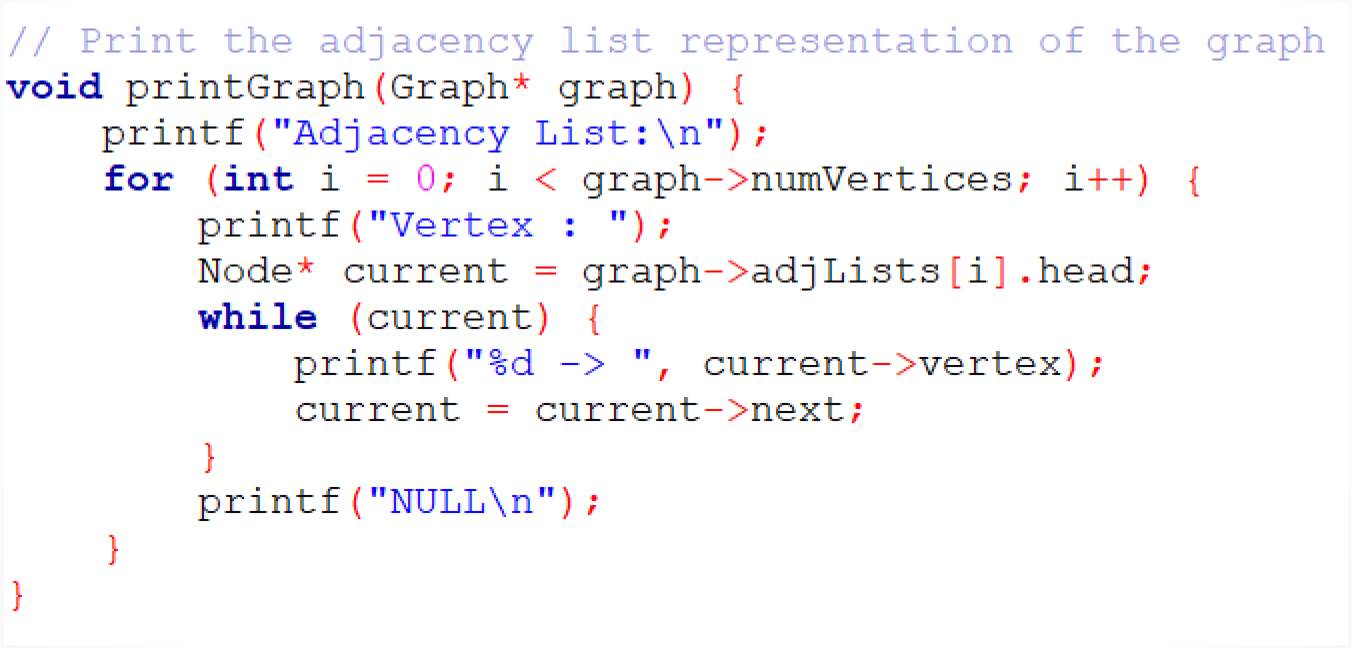


```c
// Add an edge to the graph
void addEdge(Graph* graph, int s, int d)
{
    Node *source,*destination,*temp;

    if(graph->adjLists[s].head==NULL)
    {
        source = (Node*)malloc(sizeof(Node));
        source->vertex = s;
        source->next = NULL;
        graph->adjLists[s].head = source;
    }

    destination = (Node*)malloc(sizeof(Node));
    destination->vertex = d;
    destination->next = NULL;

    temp = graph->adjLists[s].head;

    while(temp->next!=NULL)
        temp = temp->next;

    temp->next = destination;
```



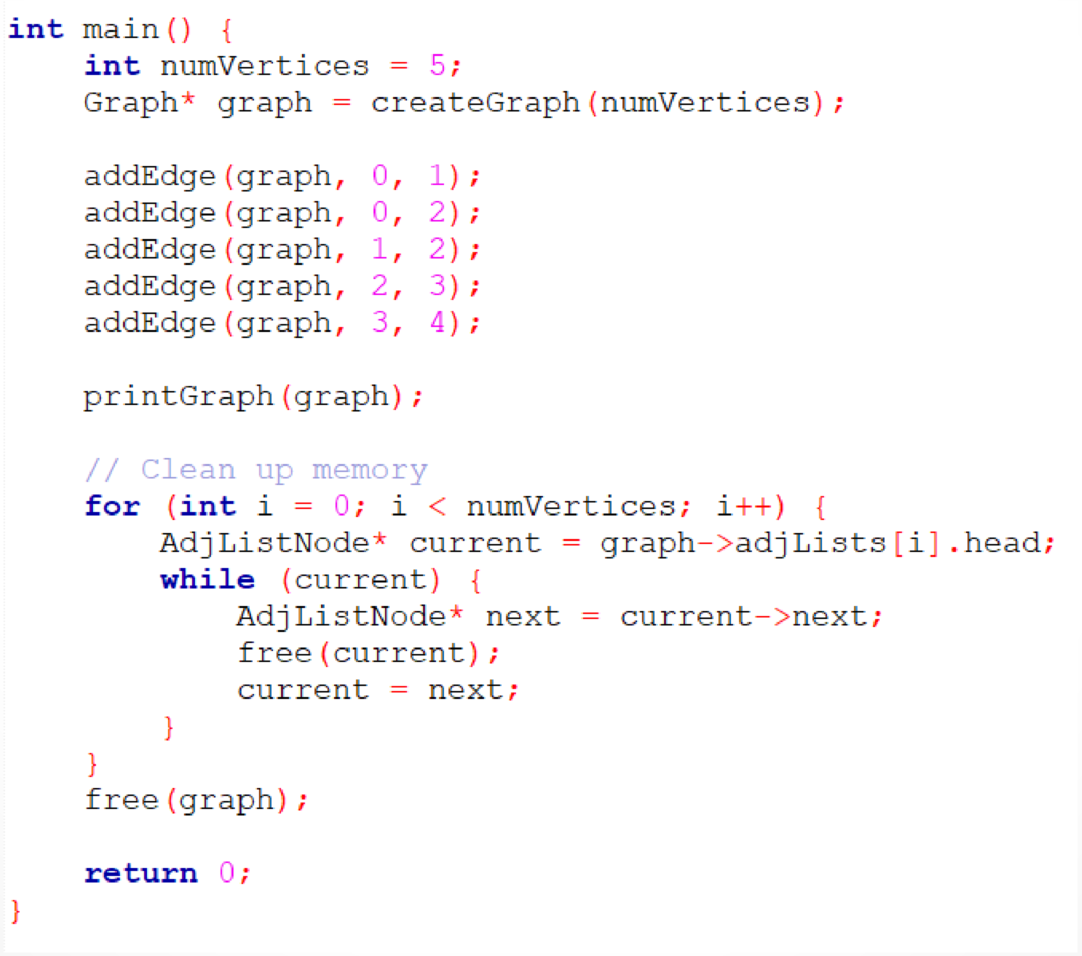
```c
    //Adding the edge back

    if(graph->adjLists[d].head==NULL)
    {
        source = (Node*)malloc(sizeof(Node));
        source->vertex = d;
        source->next = NULL;
        graph->adjLists[d].head = source;
    }

    destination = (Node*)malloc(sizeof(Node));
    destination->vertex = s;
    destination->next = NULL;

    temp = graph->adjLists[d].head;

    while(temp->next!=NULL)
        temp = temp->next;

    temp->next = destination;

}
```

Graphs



```c
// Print the adjacency list representation of the graph
void printGraph(Graph* graph) {
    printf("Adjacency List:\n");
    for (int i = 0; i < graph->numVertices; i++) {
        printf("Vertex : ");
        Node* current = graph->adjLists[i].head;
        while (current) {
            printf("%d -> ", current->vertex);
            current = current->next;
        }
        printf("NULL\n");
    }
}
```

Graphs



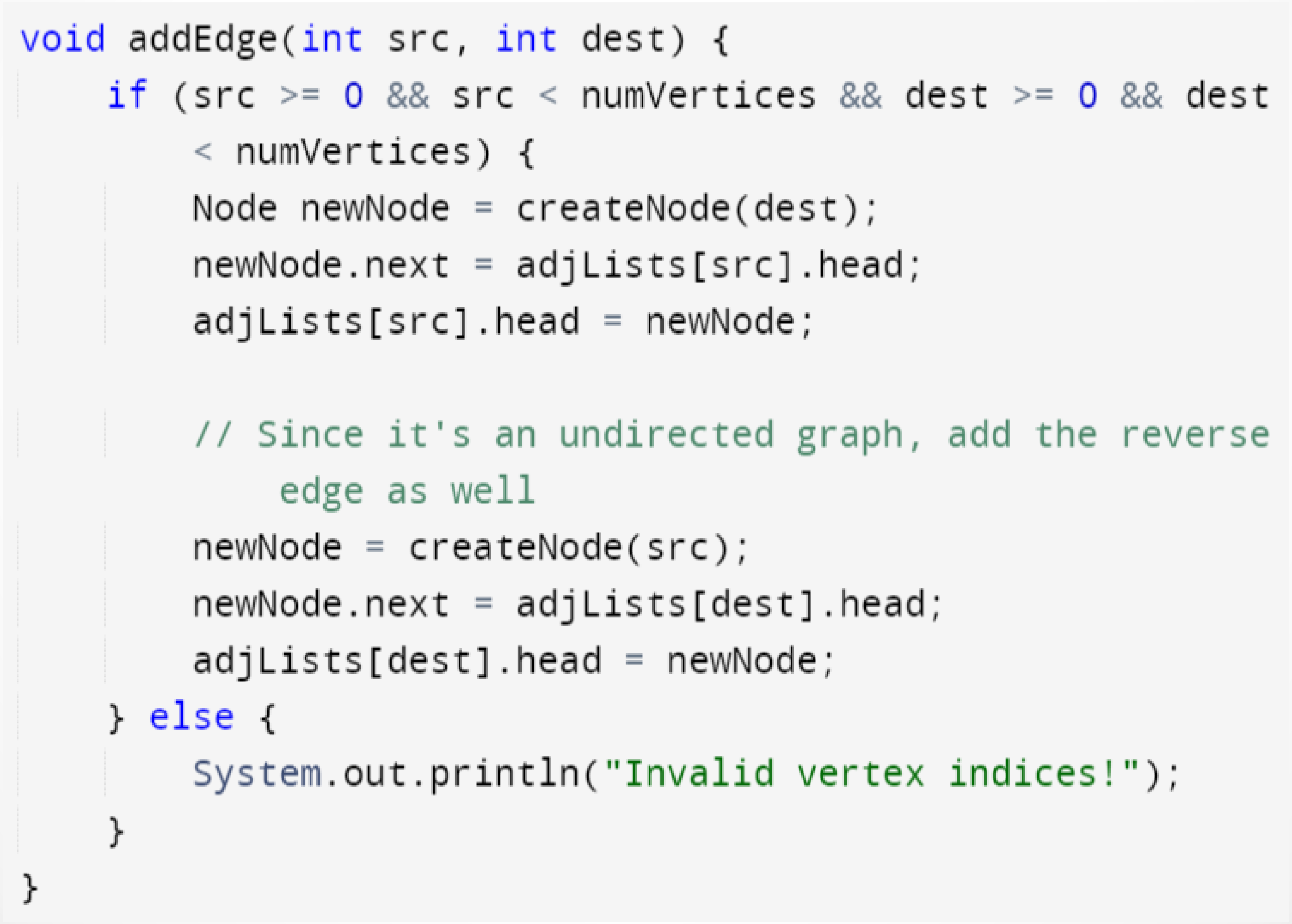
```
int main() {
    int numVertices = 5;
    Graph* graph = createGraph(numVertices);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    printGraph(graph);

    // Clean up memory
    for (int i = 0; i < numVertices; i++) {
        AdjListNode* current = graph->adjLists[i].head;
        while (current) {
            AdjListNode* next = current->next;
            free(current);
            current = next;
        }
    }
    free(graph);

    return 0;
}
```

Graphs

```
class Node {
    int vertex;
    Node next;

    Node(int vertex) {
        this.vertex = vertex;
        this.next = null;
    }
}

class List {
    Node head;

    List() {
        this.head = null;
    }
}
```

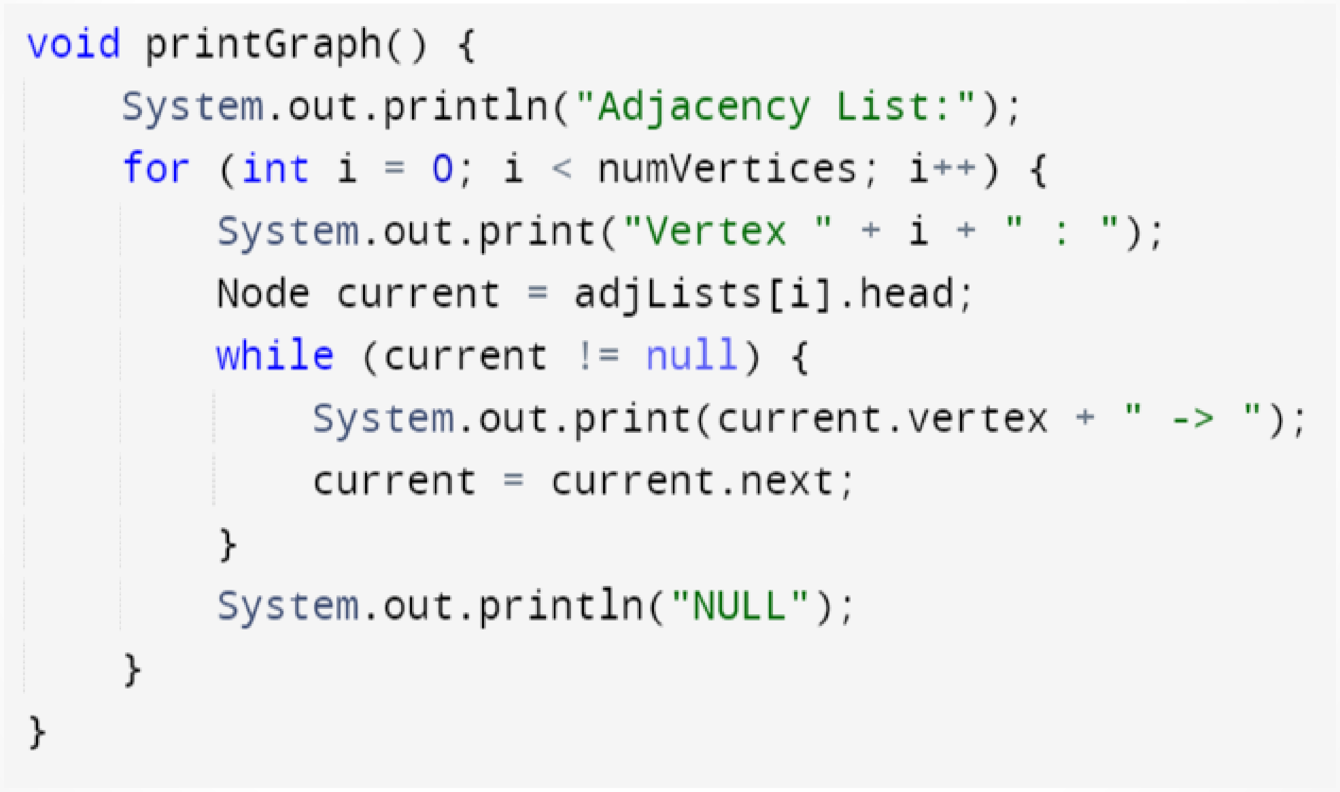
```
class Graph {
    private int numVertices;
    private List[] adjLists;
    private static final int MAX_VERTICES = 10;

    Graph(int numVertices) {
        this.numVertices = numVertices;
        adjLists = new List[MAX_VERTICES];

        for (int i = 0; i < numVertices; i++) {
            adjLists[i] = new List();
        }
    }

    Node createNode(int vertex) {
        return new Node(vertex);
    }
```

Graphs



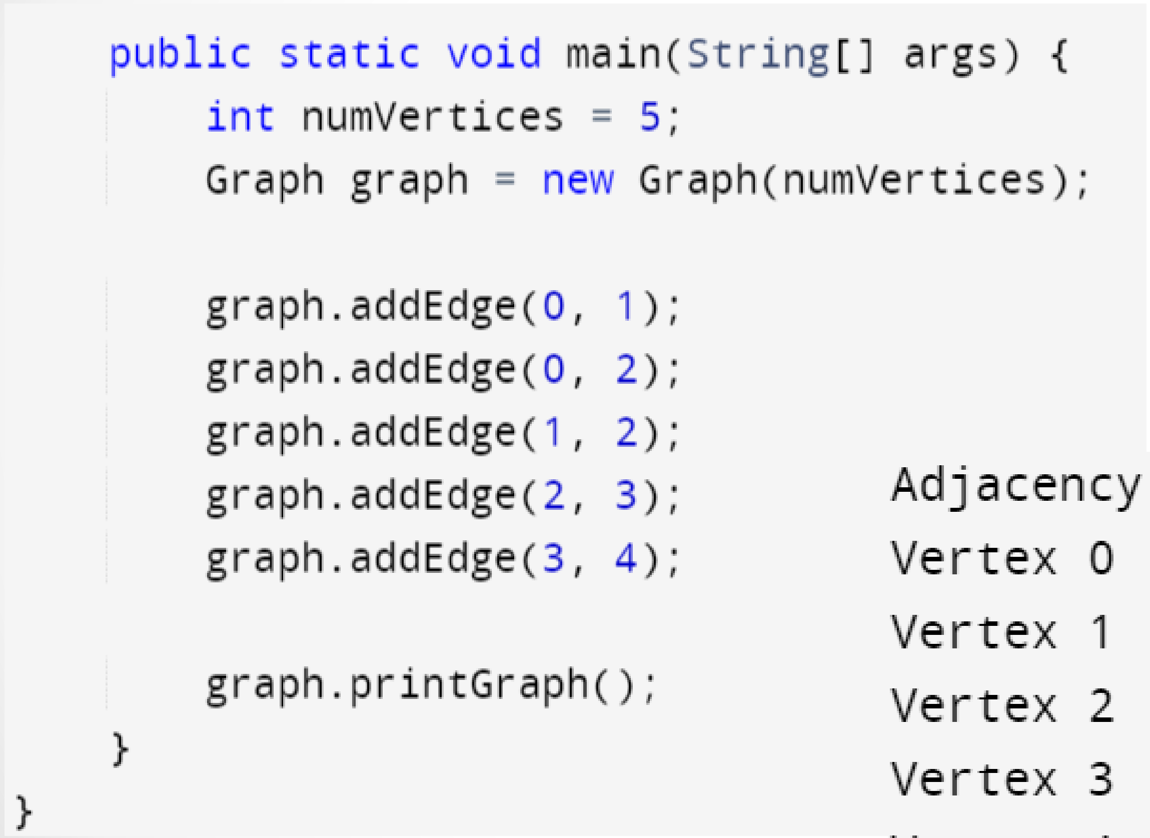
```
void addEdge(int src, int dest) {
    if (src >= 0 && src < numVertices && dest >= 0 && dest
        < numVertices) {
        Node newNode = createNode(dest);
        newNode.next = adjLists[src].head;
        adjLists[src].head = newNode;

        // Since it's an undirected graph, add the reverse
            edge as well
        newNode = createNode(src);
        newNode.next = adjLists[dest].head;
        adjLists[dest].head = newNode;
    } else {
        System.out.println("Invalid vertex indices!");
    }
}
```

Graphs

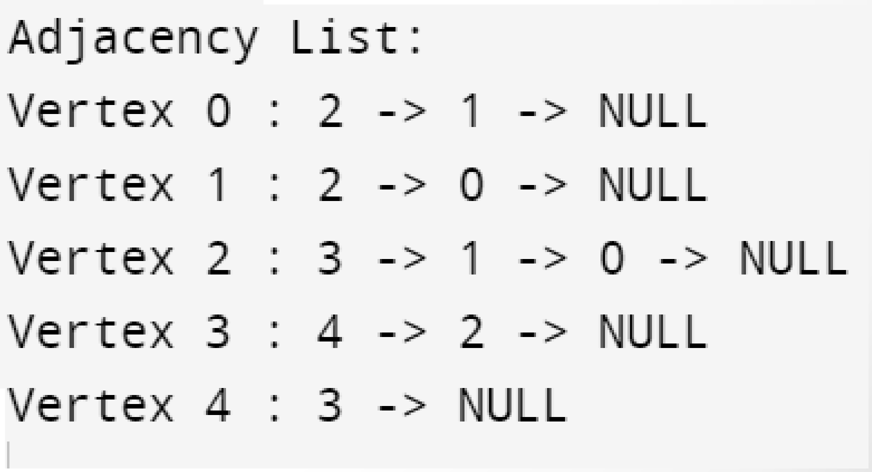


```java
void printGraph() {
    System.out.println("Adjacency List:");
    for (int i = 0; i < numVertices; i++) {
        System.out.print("Vertex " + i + " : ");
        Node current = adjLists[i].head;
        while (current != null) {
            System.out.print(current.vertex + " -> ");
            current = current.next;
        }
        System.out.println("NULL");
    }
}
```

Graphs



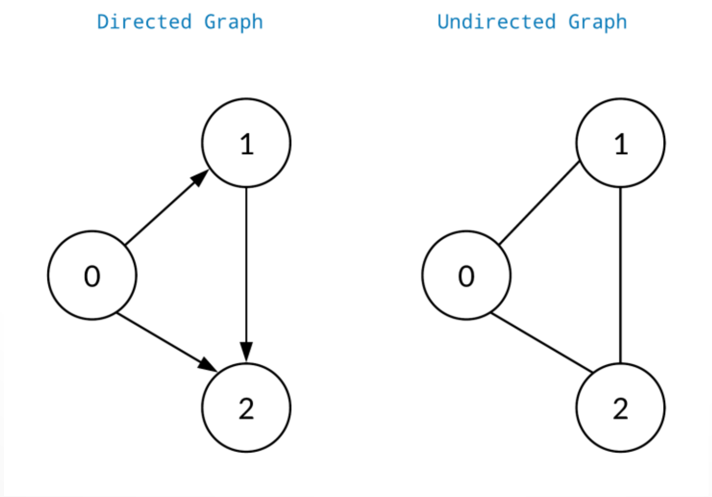
```
    public static void main(String[] args) {
        int numVertices = 5;
        Graph graph = new Graph(numVertices);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 2);
        graph.addEdge(2, 3);
        graph.addEdge(3, 4);

        graph.printGraph();
    }
}
```



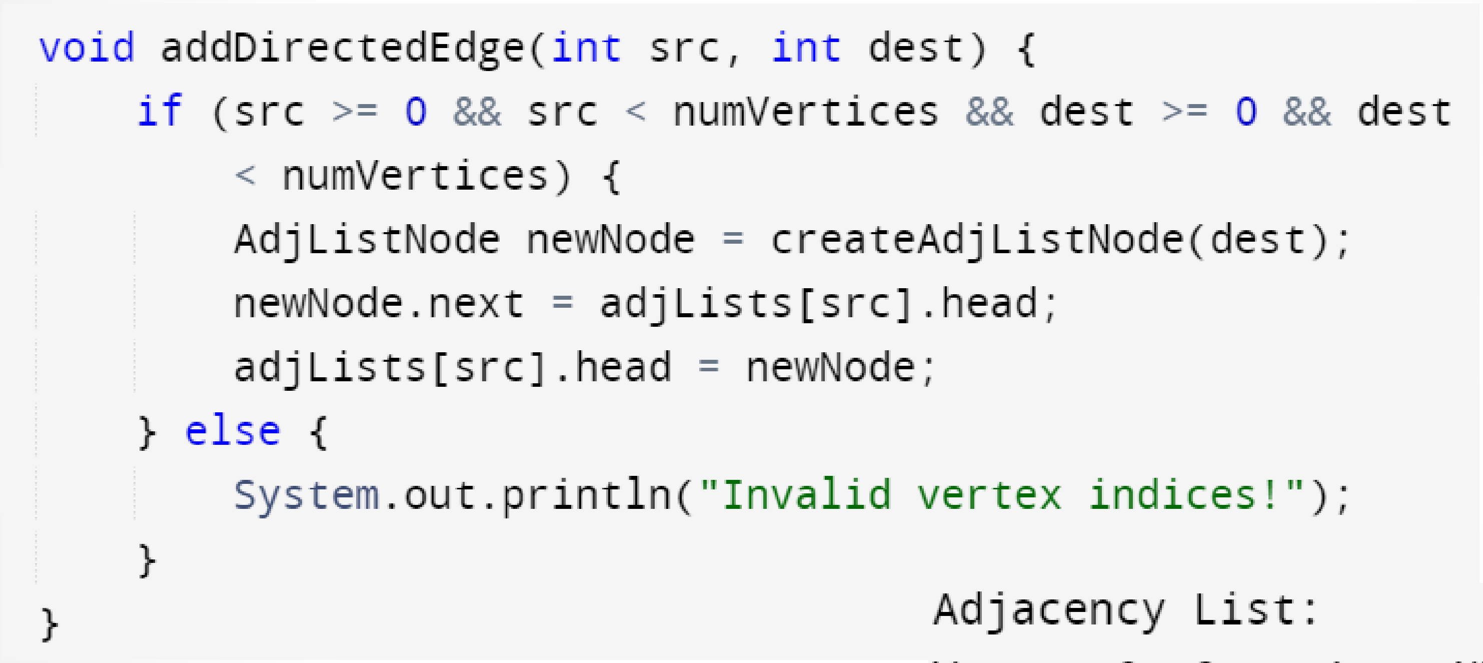
```
Adjacency List:
Vertex 0 : 2 -> 1 -> NULL
Vertex 1 : 2 -> 0 -> NULL
Vertex 2 : 3 -> 1 -> 0 -> NULL
Vertex 3 : 4 -> 2 -> NULL
Vertex 4 : 3 -> NULL
```

Directed vs. Undirected Graphs

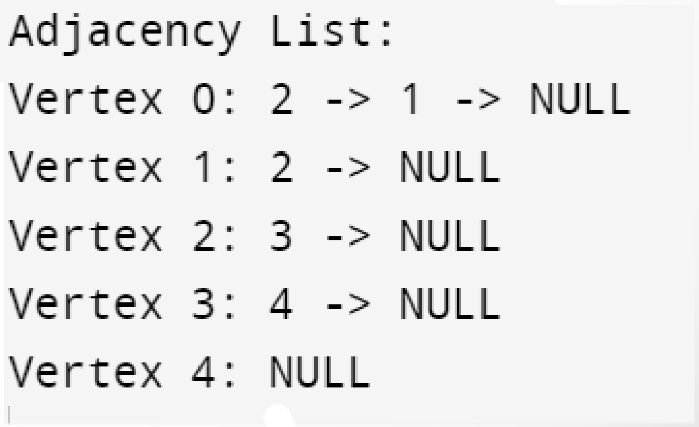
- In an undirected graph, edges have no direction and can be traversed in either direction.
- In a directed graph (also known as a digraph), edges have a direction, pointing from one vertex (the source) to another vertex (the target).



Directed graphs



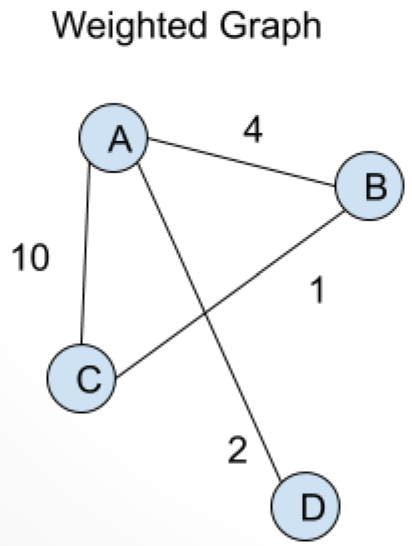
```java
void addDirectedEdge(int src, int dest) {
    if (src >= 0 && src < numVertices && dest >= 0 && dest
        < numVertices) {
        AdjListNode newNode = createAdjListNode(dest);
        newNode.next = adjLists[src].head;
        adjLists[src].head = newNode;
    } else {
        System.out.println("Invalid vertex indices!");
    }
}
```

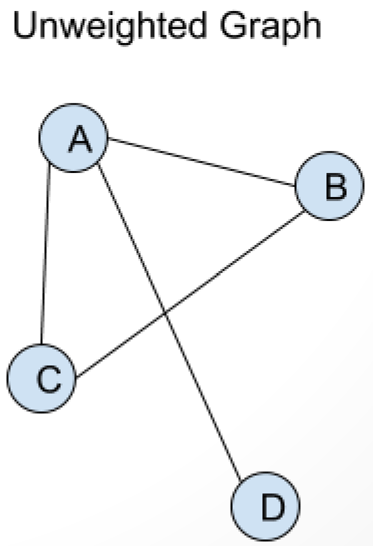


```
Adjacency List:
Vertex 0: 2 -> 1 -> NULL
Vertex 1: 2 -> NULL
Vertex 2: 3 -> NULL
Vertex 3: 4 -> NULL
Vertex 4: NULL
```

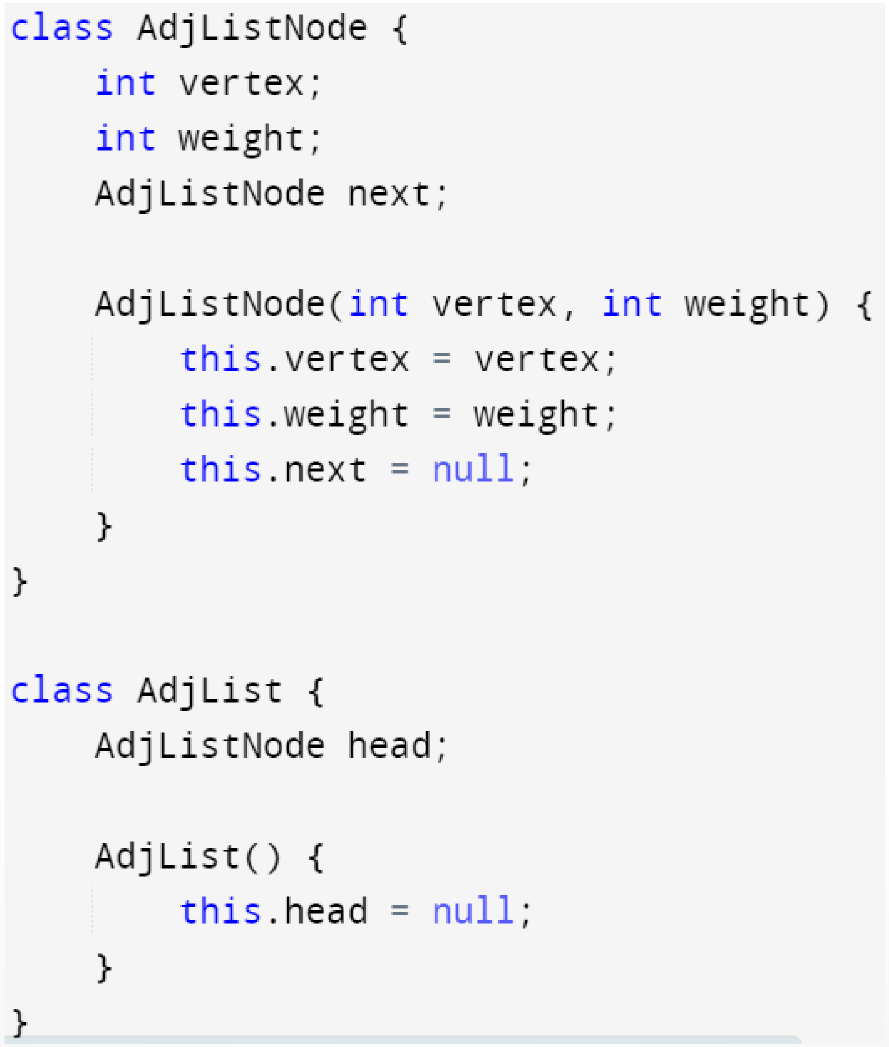
Weighted and Unweighted Graphs

- Graphs can have weights associated with their edges, representing the cost, distance, or some other value.
- Unweighted graphs have edges without weights.





Weighted graphs

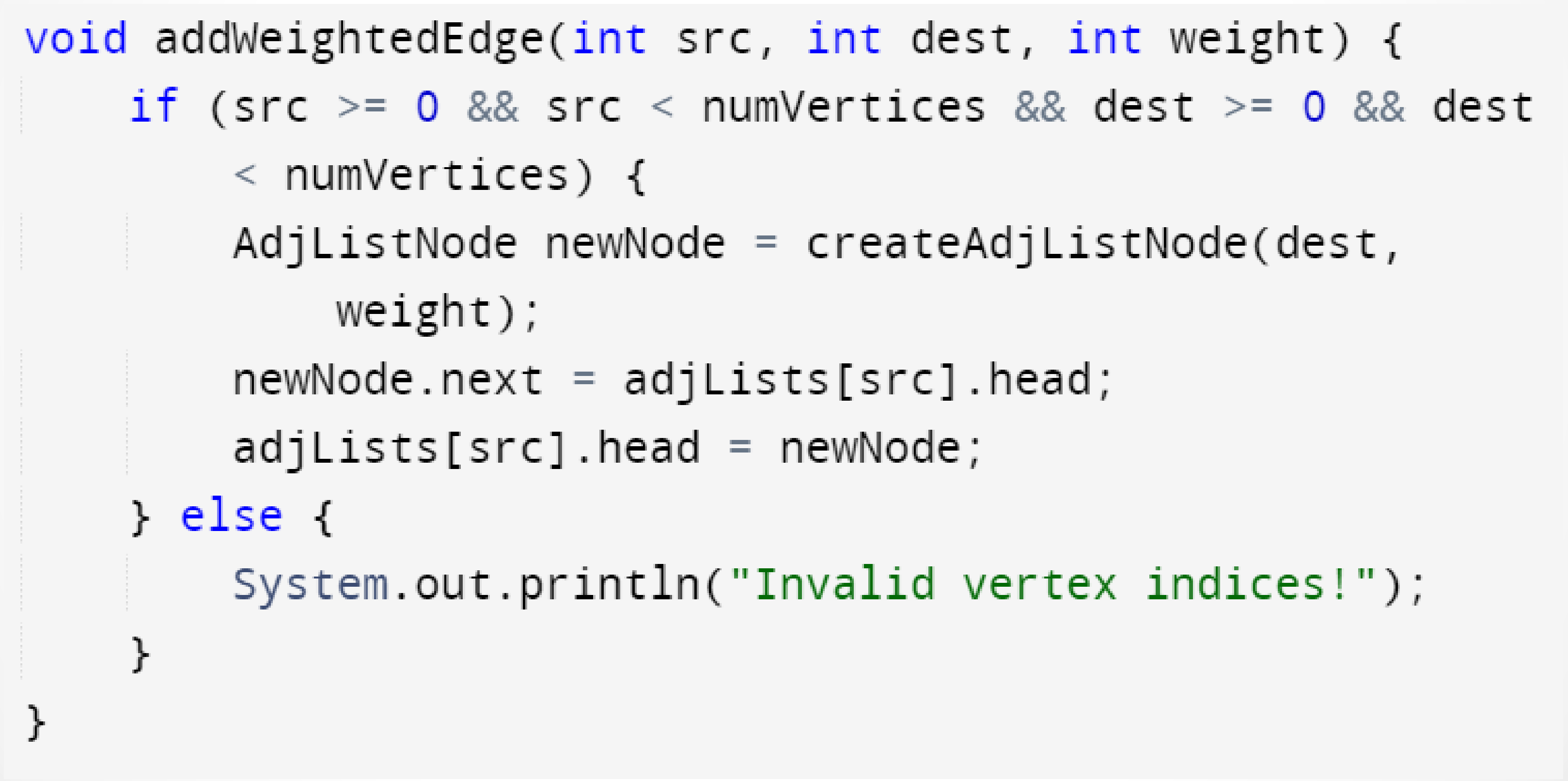


```
class AdjListNode {
    int vertex;
    int weight;
    AdjListNode next;

    AdjListNode(int vertex, int weight) {
        this.vertex = vertex;
        this.weight = weight;
        this.next = null;
    }
}

class AdjList {
    AdjListNode head;

    AdjList() {
        this.head = null;
    }
}
```

Weighted graphs

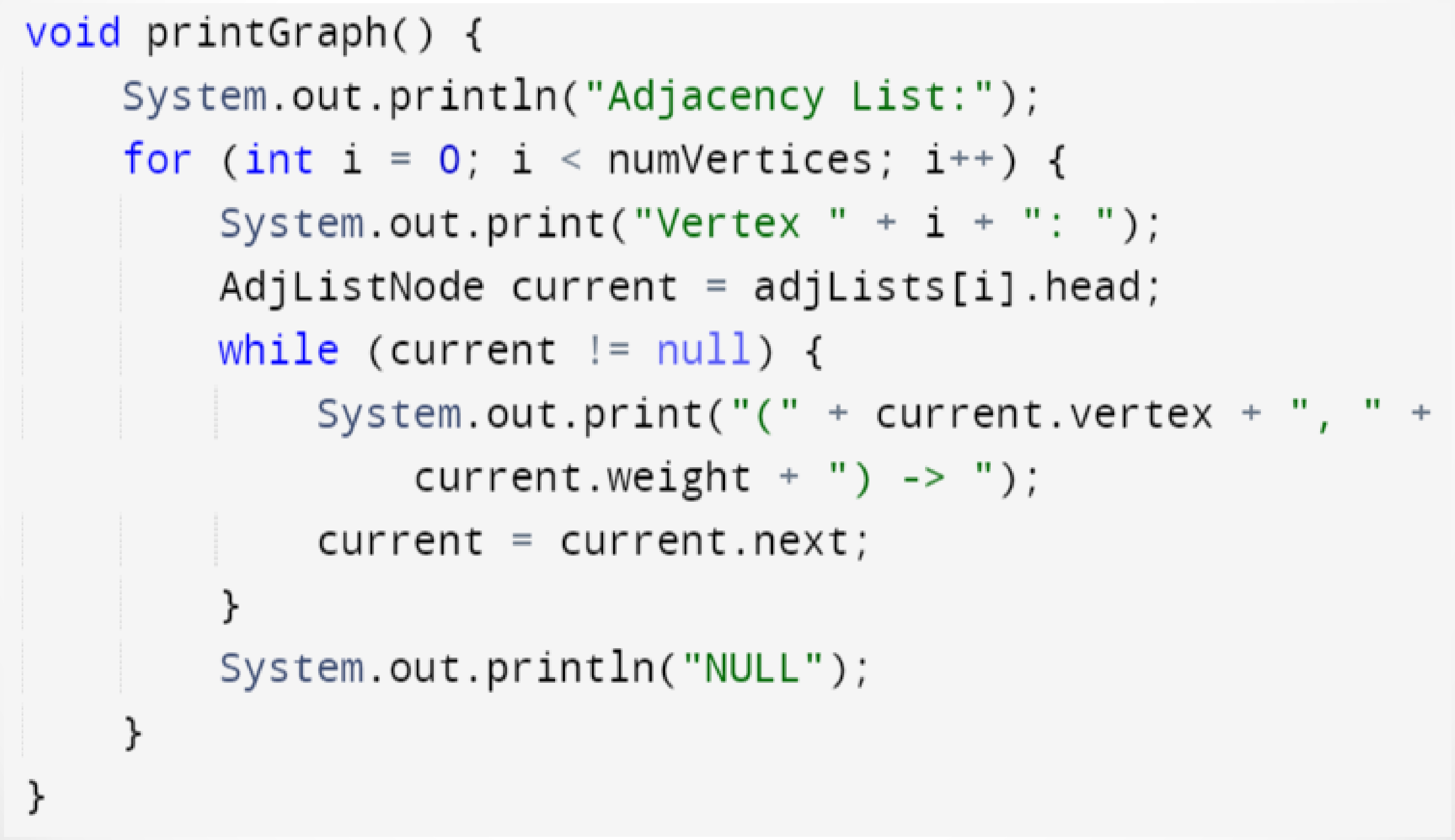
```
class Graph {
    private int numVertices;
    private AdjList[] adjLists;
    private static final int MAX_VERTICES = 10;
    private static final int INF = 99999;

    Graph(int numVertices) {
        this.numVertices = numVertices;
        adjLists = new AdjList[MAX_VERTICES];

        for (int i = 0; i < numVertices; i++) {
            adjLists[i] = new AdjList();
        }
    }

    AdjListNode createAdjListNode(int vertex, int weight) {
        return new AdjListNode(vertex, weight);
    }
```

Weighted graphs



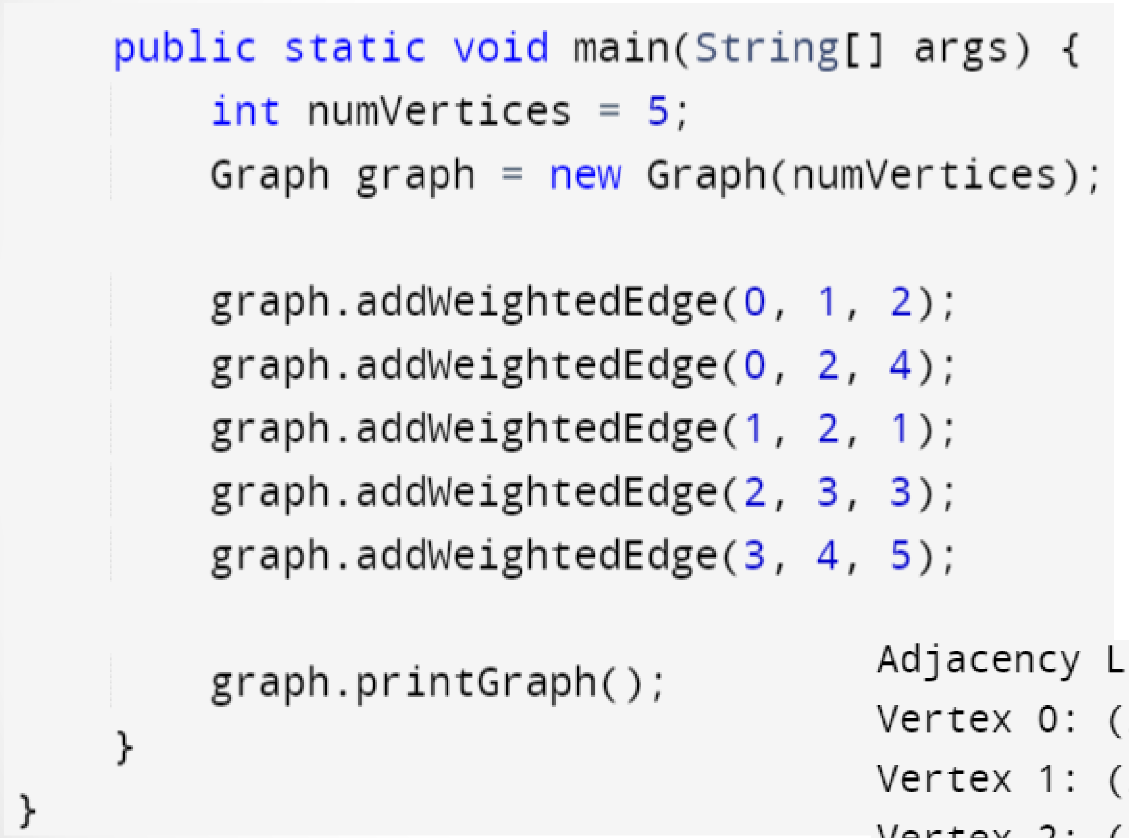
```
void addWeightedEdge(int src, int dest, int weight) {
    if (src >= 0 && src < numVertices && dest >= 0 && dest
        < numVertices) {
        AdjListNode newNode = createAdjListNode(dest,
            weight);
        newNode.next = adjLists[src].head;
        adjLists[src].head = newNode;
    } else {
        System.out.println("Invalid vertex indices!");
    }
}
```

Weighted graphs

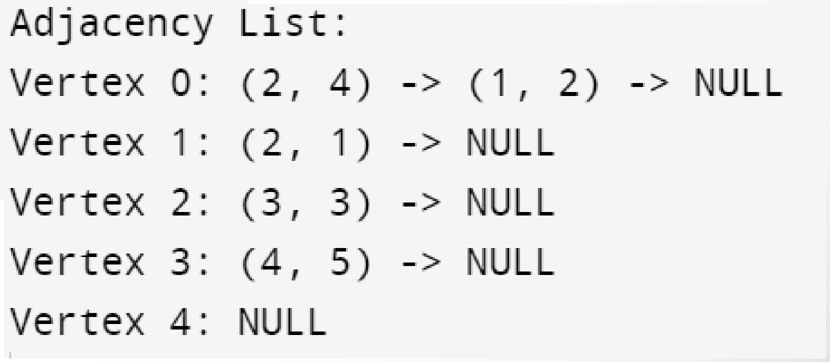


```
void printGraph() {
    System.out.println("Adjacency List:");
    for (int i = 0; i < numVertices; i++) {
        System.out.print("Vertex " + i + ": ");
        AdjListNode current = adjLists[i].head;
        while (current != null) {
            System.out.print("(" + current.vertex + ", " +
                    current.weight + ") -> ");
            current = current.next;
        }
        System.out.println("NULL");
    }
}
```

Weighted graphs



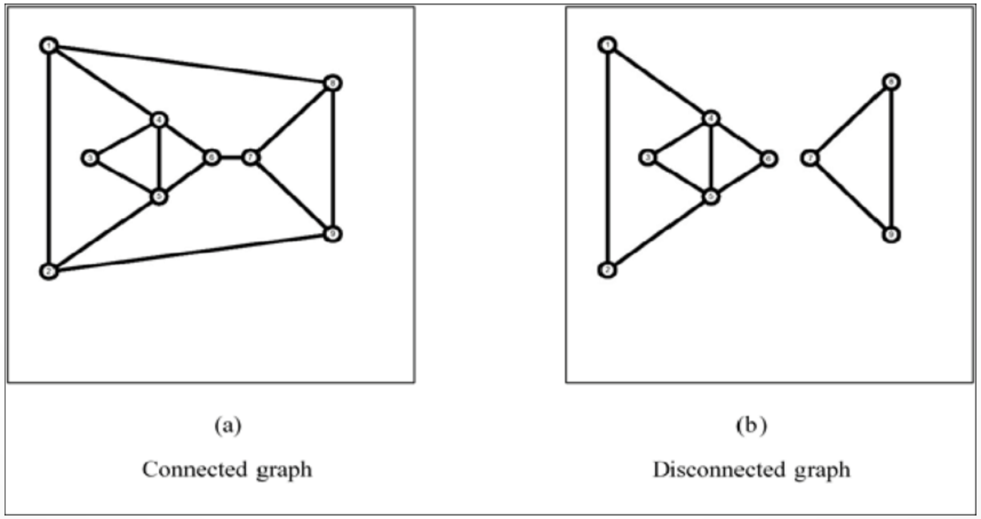
```
    public static void main(String[] args) {
        int numVertices = 5;
        Graph graph = new Graph(numVertices);

        graph.addWeightedEdge(0, 1, 2);
        graph.addWeightedEdge(0, 2, 4);
        graph.addWeightedEdge(1, 2, 1);
        graph.addWeightedEdge(2, 3, 3);
        graph.addWeightedEdge(3, 4, 5);

        graph.printGraph();
    }
}
```



```
Adjacency List:
Vertex 0: (2, 4) -> (1, 2) -> NULL
Vertex 1: (2, 1) -> NULL
Vertex 2: (3, 3) -> NULL
Vertex 3: (4, 5) -> NULL
Vertex 4: NULL
```

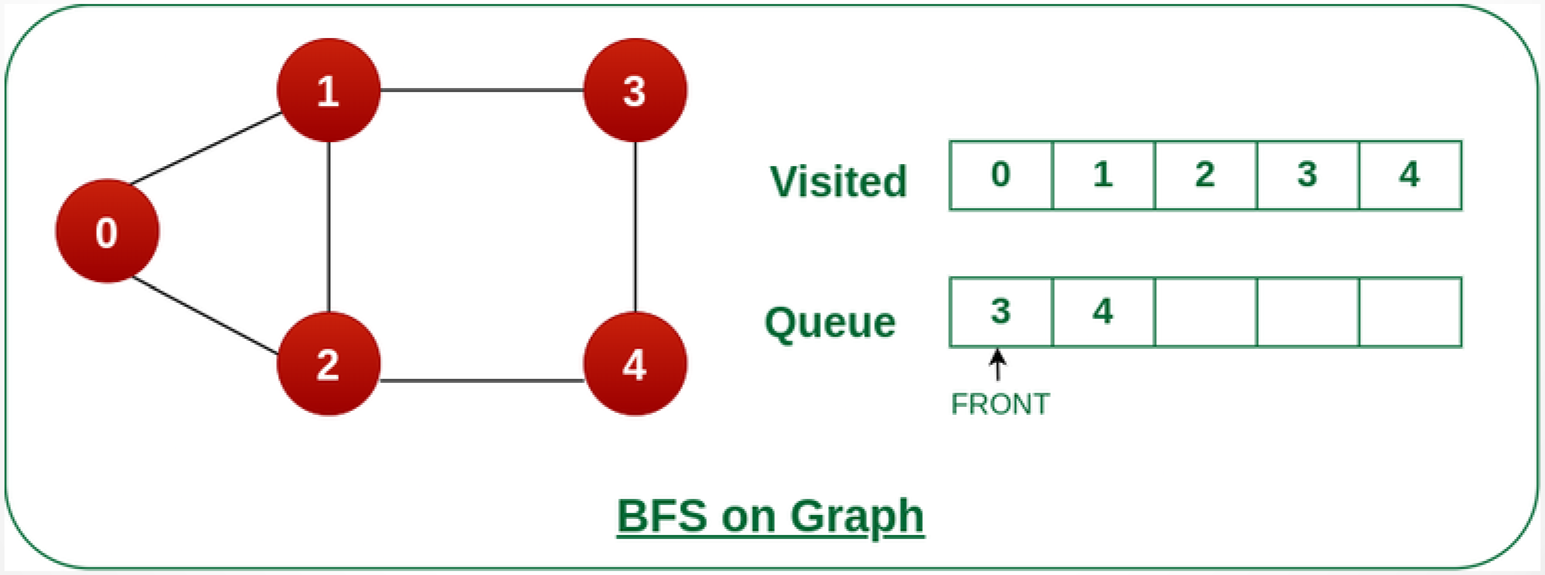
Connected Graphs

- A graph is considered connected if there is a path between every pair of vertices.
- A graph can have connected components, which are sub-graphs where all vertices are reachable from each other



Breadth first search

- BFS is a graph traversal algorithm that visits all the vertices of a graph level by level, starting from a specified source vertex.

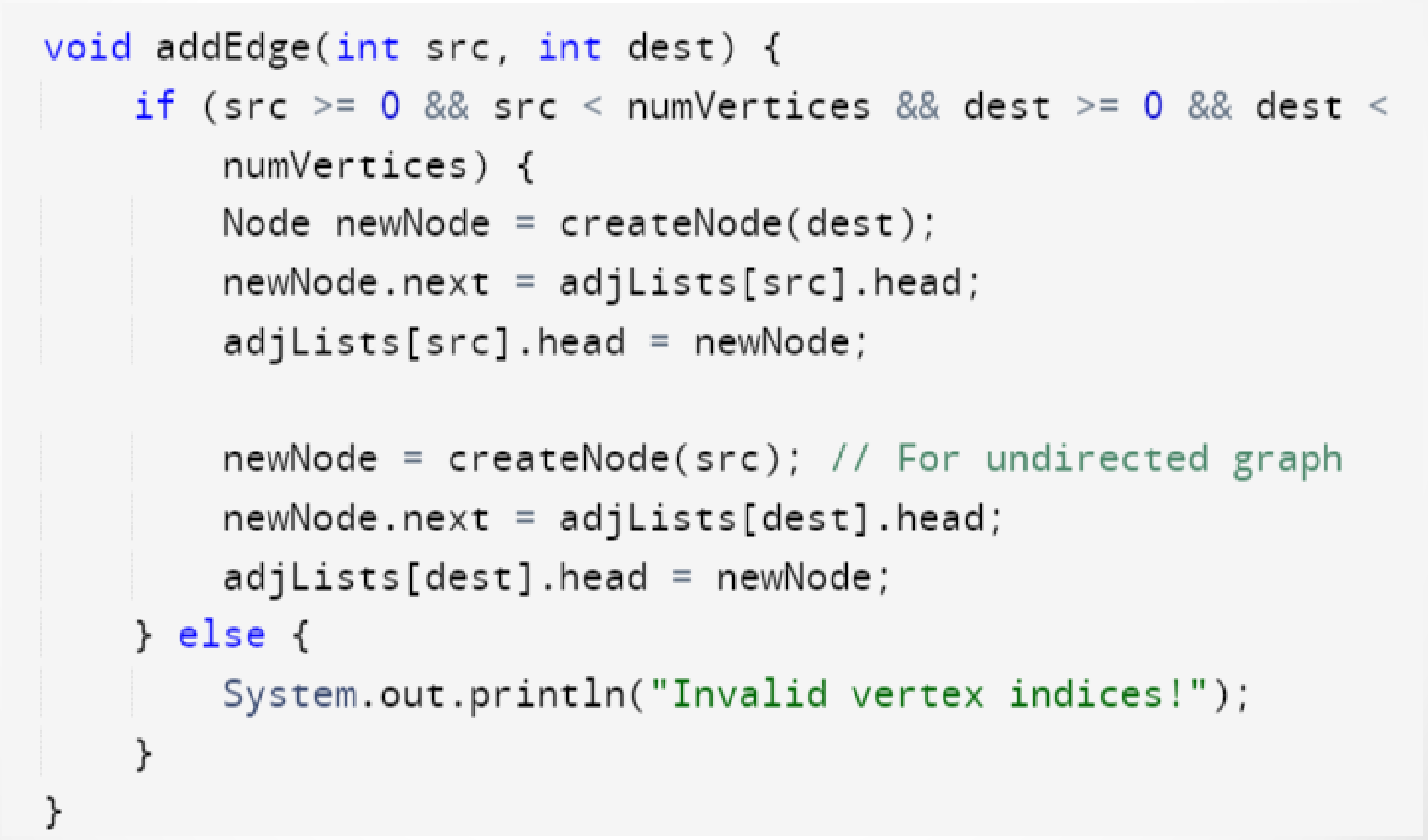


Breadth first search

```java
import java.util.LinkedList;
import java.util.Queue;

class Node {
    int vertex;
    Node next;

    Node(int vertex) {
        this.vertex = vertex;
        this.next = null;
    }
}

class AdjList {
    Node head;

    AdjList() {
        this.head = null;
    }
```

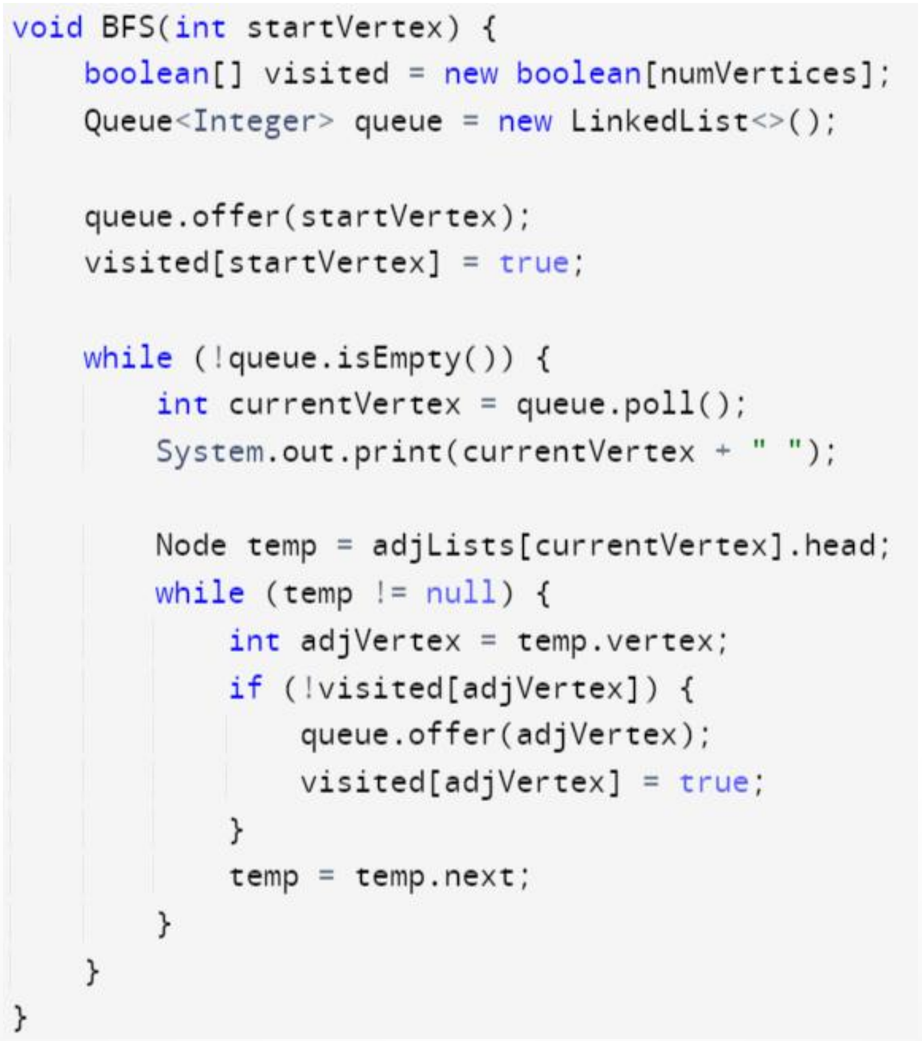
```java
class Graph {
    int numVertices;
    AdjList[] adjLists;

    Graph(int numVertices) {
        this.numVertices = numVertices;
        adjLists = new AdjList[numVertices];


        for (int i = 0; i < numVertices; i++) {
            adjLists[i] = new AdjList();
        }
    }

    Node createNode(int vertex) {
        return new Node(vertex);
    }
```

Breadth first search



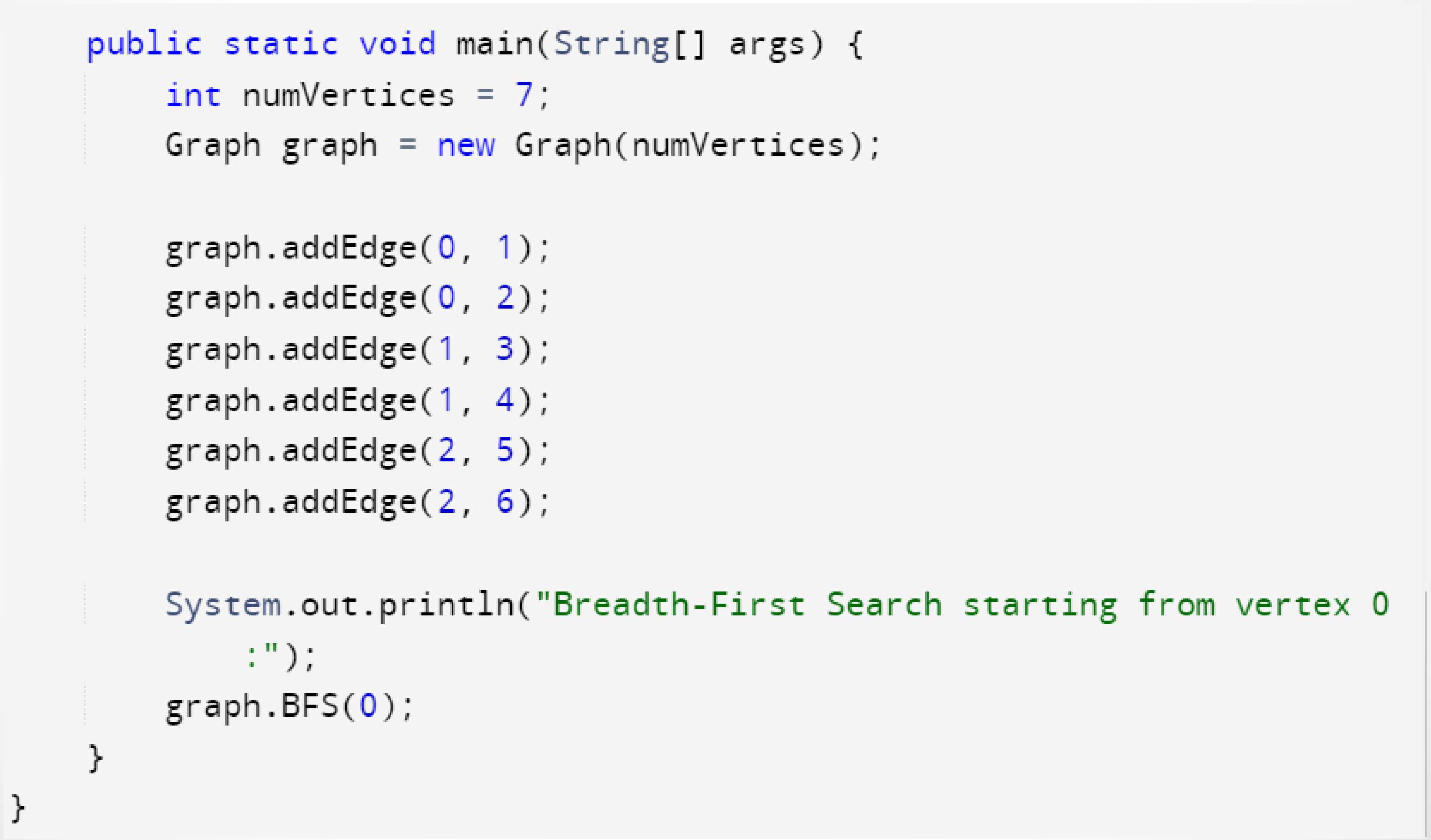
```java
void addEdge(int src, int dest) {
    if (src >= 0 && src < numVertices && dest >= 0 && dest <
        numVertices) {
        Node newNode = createNode(dest);
        newNode.next = adjLists[src].head;
        adjLists[src].head = newNode;

        newNode = createNode(src); // For undirected graph
        newNode.next = adjLists[dest].head;
        adjLists[dest].head = newNode;
    } else {
        System.out.println("Invalid vertex indices!");
    }
}
```

Breadth first search



```
void BFS(int startVertex) {
    boolean[] visited = new boolean[numVertices];
    Queue<Integer> queue = new LinkedList<>();

    queue.offer(startVertex);
    visited[startVertex] = true;

    while (!queue.isEmpty()) {
        int currentVertex = queue.poll();
        System.out.print(currentVertex + " ");

        Node temp = adjLists[currentVertex].head;
        while (temp != null) {
            int adjVertex = temp.vertex;
            if (!visited[adjVertex]) {
                queue.offer(adjVertex);
                visited[adjVertex] = true;
            }
            temp = temp.next;
        }
    }
}
```

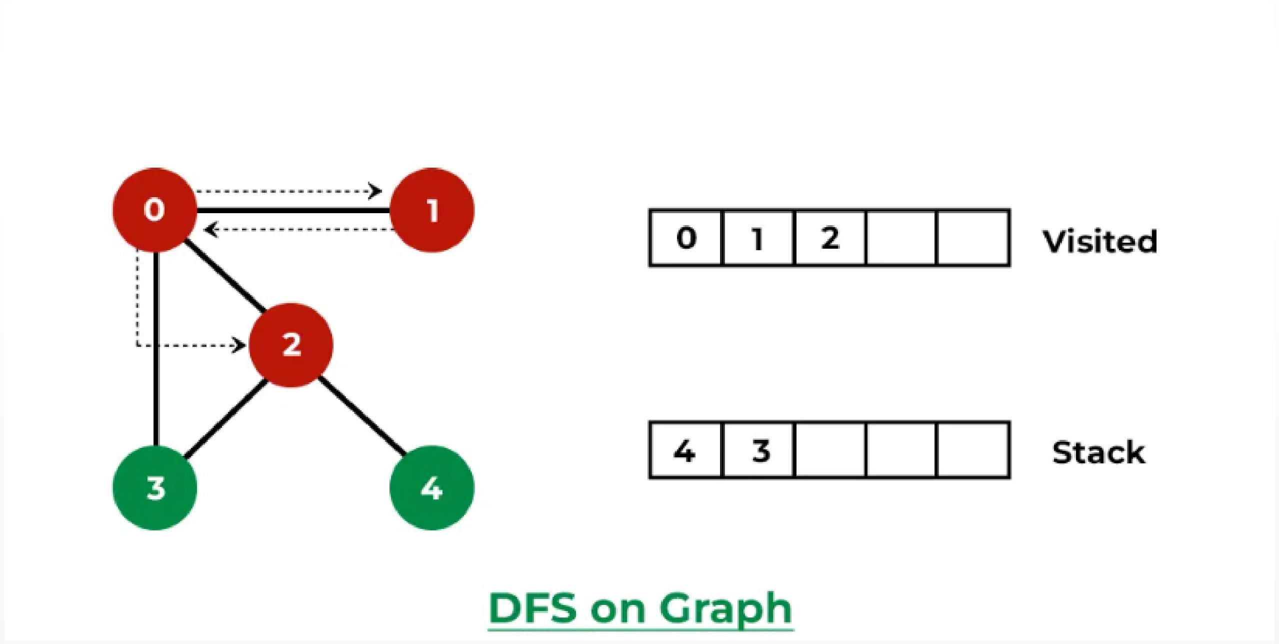
Breadth first search



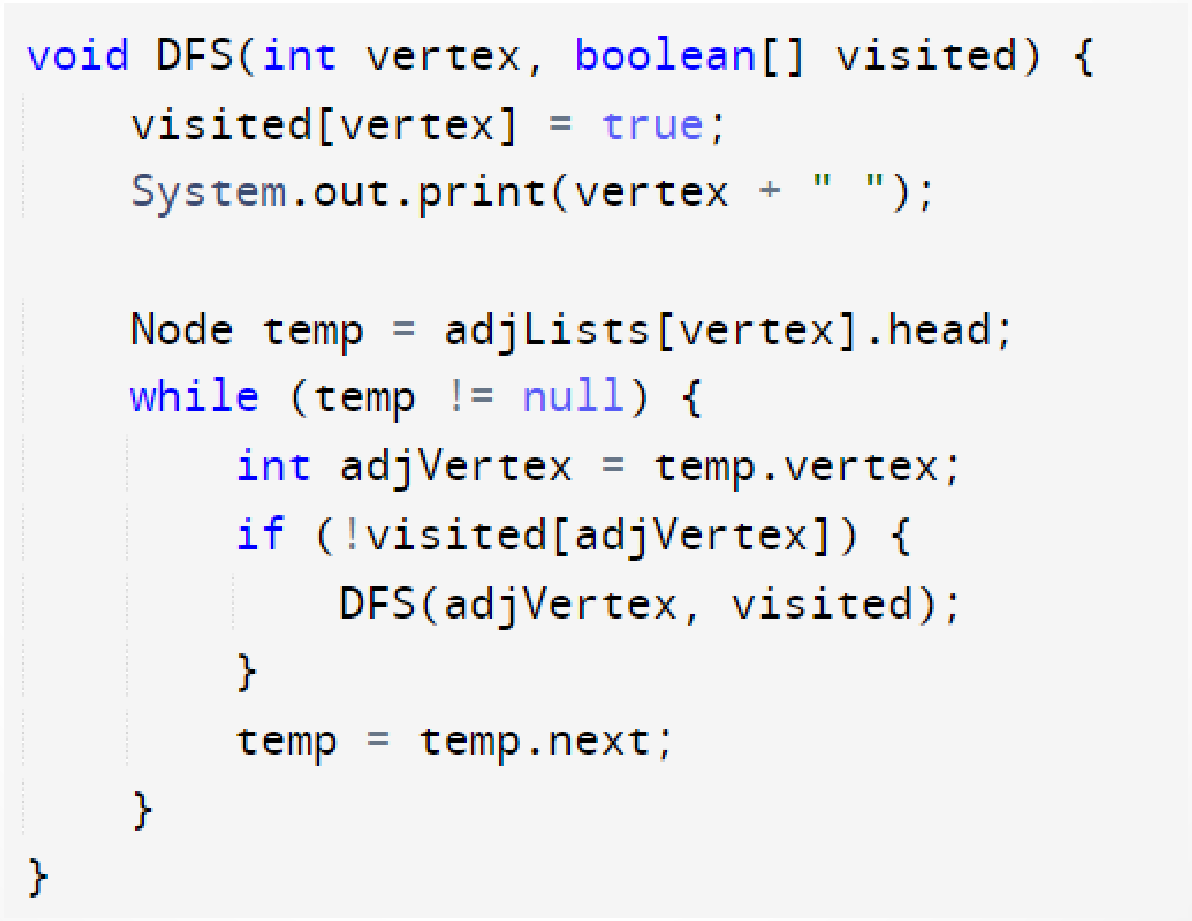
```
    public static void main(String[] args) {
        int numVertices = 7;
        Graph graph = new Graph(numVertices);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 5);
        graph.addEdge(2, 6);

        System.out.println("Breadth-First Search starting from vertex 0
            :");
        graph.BFS(0);
    }
}
```

Depth first search

- DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking.

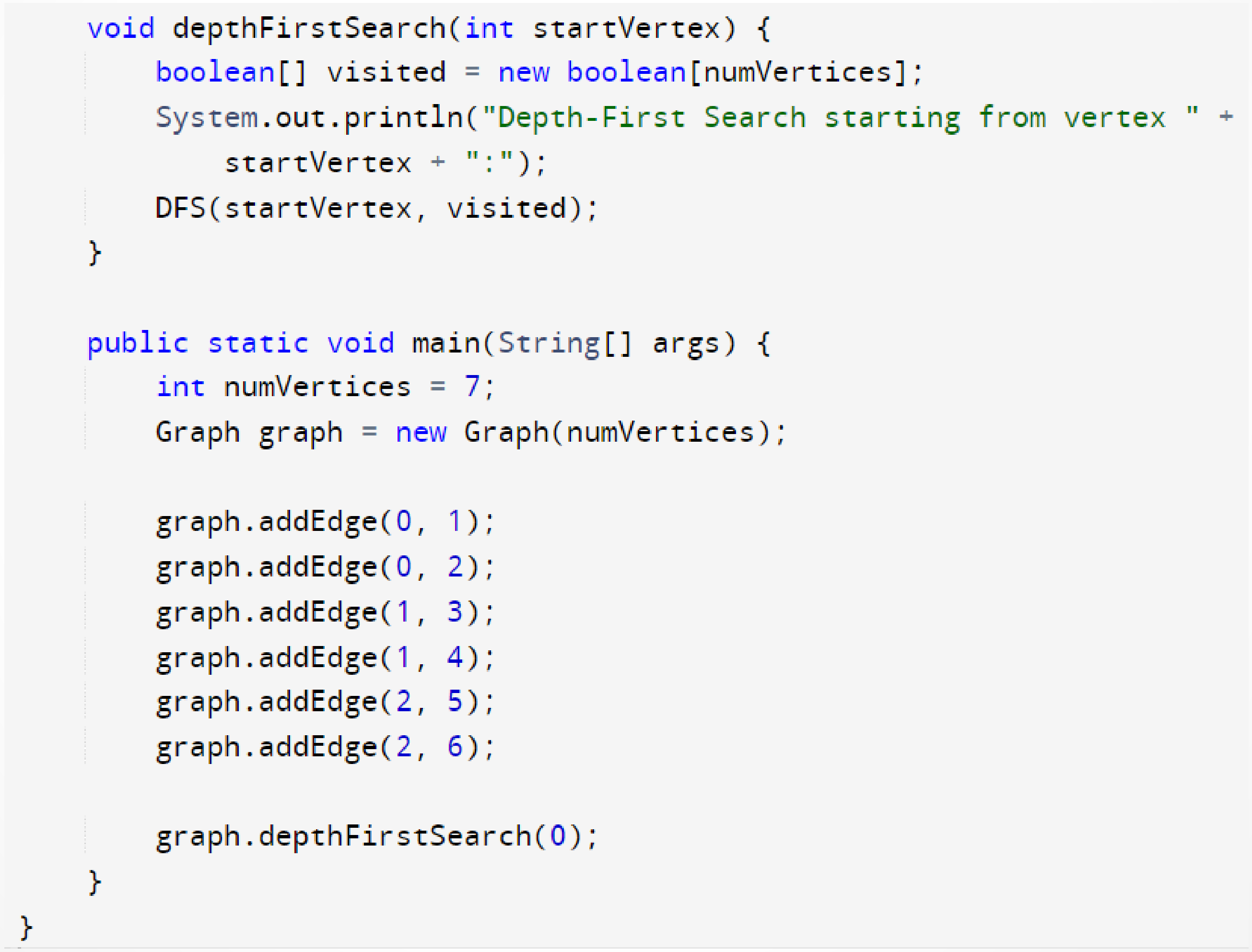


Depth first search



```
void DFS(int vertex, boolean[] visited) {
    visited[vertex] = true;
    System.out.print(vertex + " ");

    Node temp = adjLists[vertex].head;
    while (temp != null) {
        int adjVertex = temp.vertex;
        if (!visited[adjVertex]) {
            DFS(adjVertex, visited);
        }
        temp = temp.next;
    }
}
```

Depth first search



```
    void depthFirstSearch(int startVertex) {
        boolean[] visited = new boolean[numVertices];
        System.out.println("Depth-First Search starting from vertex " +
            startVertex + ":");
        DFS(startVertex, visited);
    }

    public static void main(String[] args) {
        int numVertices = 7;
        Graph graph = new Graph(numVertices);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 5);
        graph.addEdge(2, 6);

        graph.depthFirstSearch(0);
    }
}
```

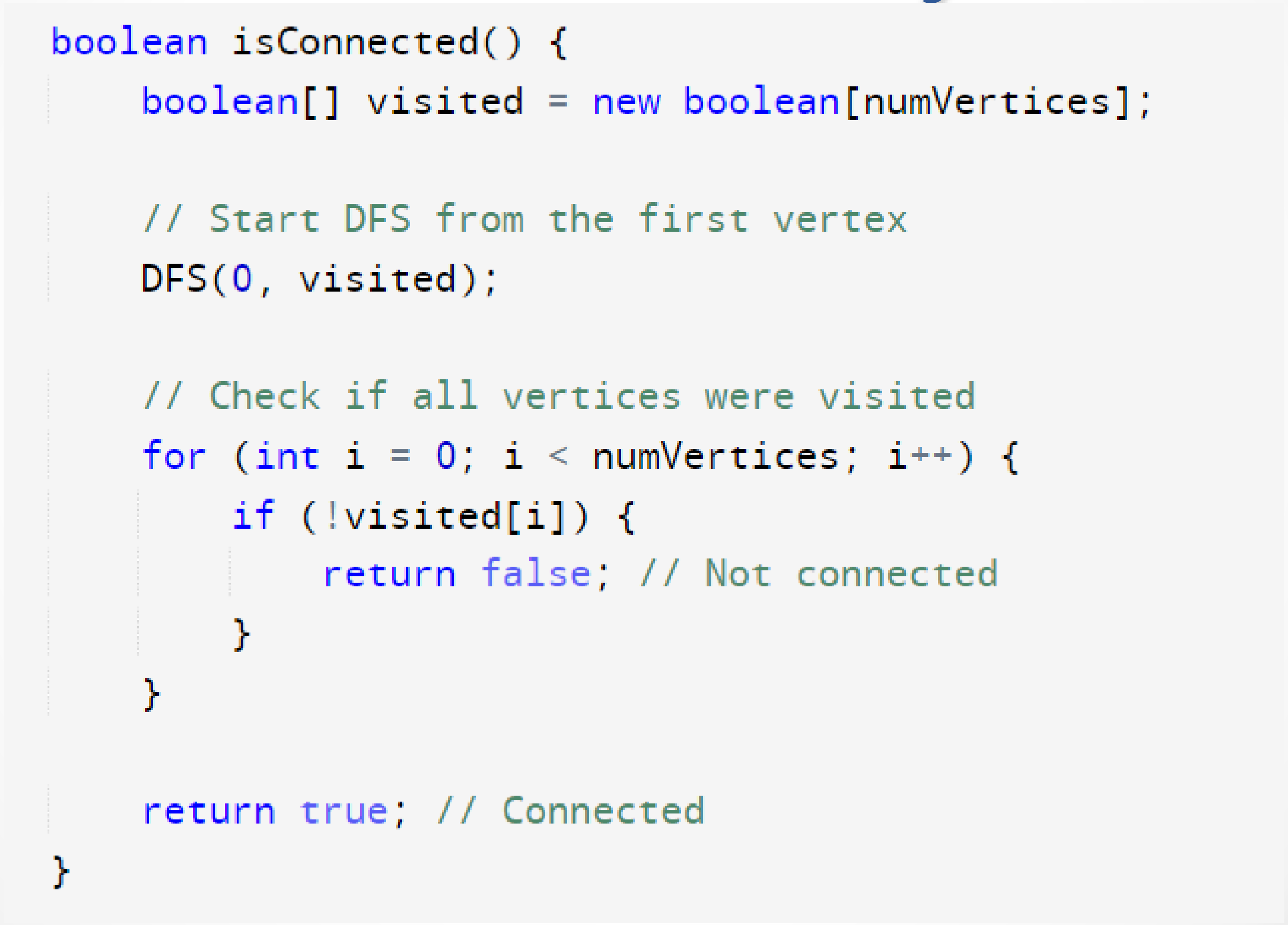
Connectivity

- Connectivity in a graph refers to the degree to which vertices are connected.
- A graph is connected if there is a path between every pair of vertices.
- The concept of connectivity is crucial in networking, where it helps model communication between devices, network reliability, and fault tolerance

Connectivity

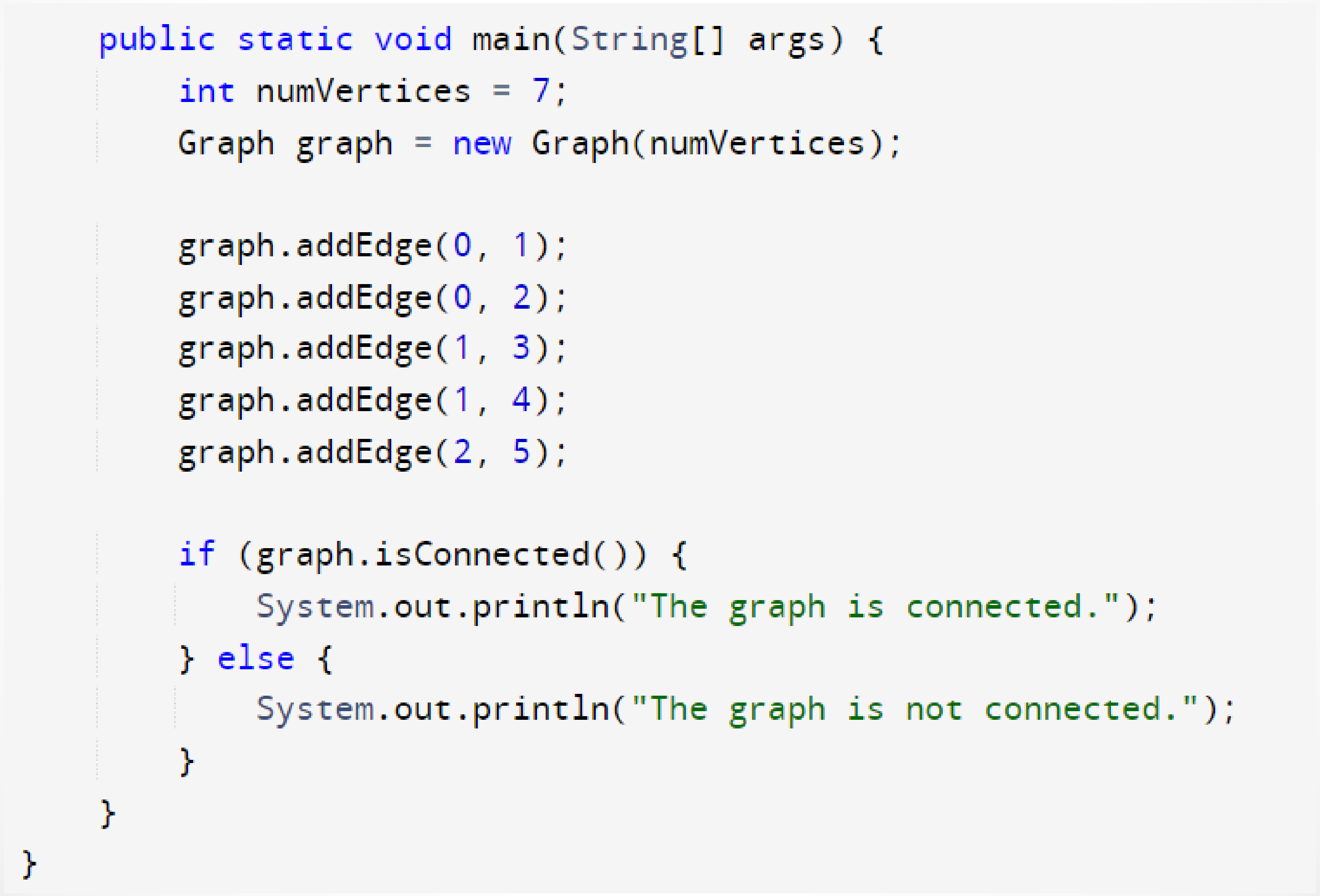
- To check the connectivity of a graph, you can perform a graph traversal (such as Depth-First Search or Breadth-First Search) starting from any vertex.
- If all vertices are visited during the traversal, the graph is considered connected.
- If there are unvisited vertices, the graph is not connected.

Connectivity



```
boolean isConnected() {
    boolean[] visited = new boolean[numVertices];

    // Start DFS from the first vertex
    DFS(0, visited);

    // Check if all vertices were visited
    for (int i = 0; i < numVertices; i++) {
        if (!visited[i]) {
            return false; // Not connected
        }
    }

    return true; // Connected
}
```

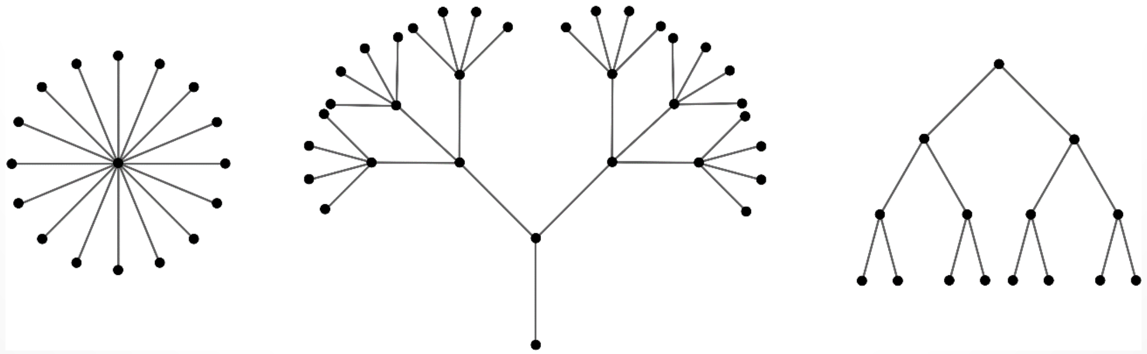
Connectivity



```java
    public static void main(String[] args) {
        int numVertices = 7;
        Graph graph = new Graph(numVertices);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 5);

        if (graph.isConnected()) {
            System.out.println("The graph is connected.");
        } else {
            System.out.println("The graph is not connected.");
        }
    }
}
```

Tree

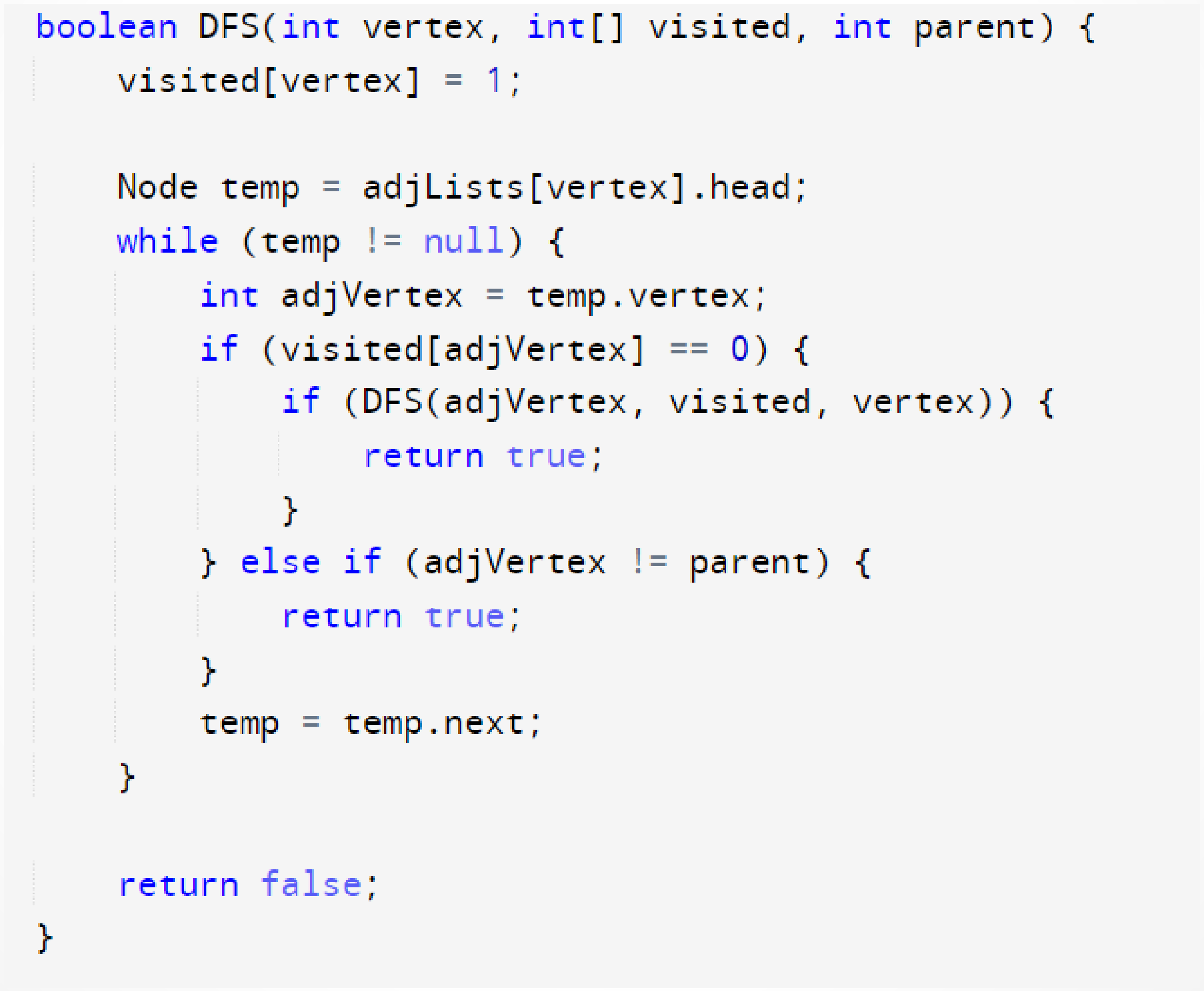
- A tree is a special type of graph with no cycles and a single root vertex.
- Trees are widely used in computer science for data structures (e.g., binary trees) and algorithm design (e.g., depth-first search)



Tree

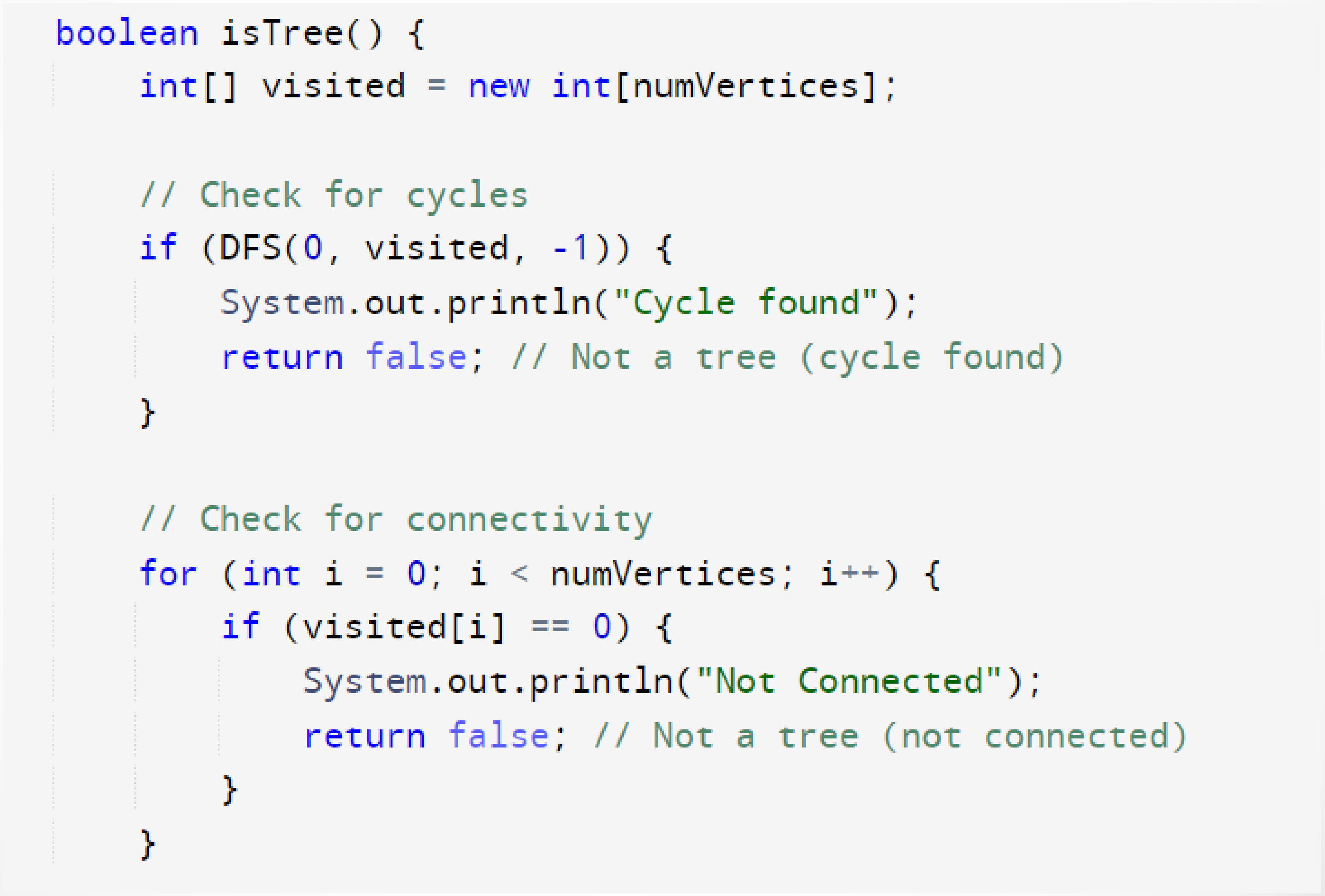
- To check whether a graph is a tree, you can use graph traversal algorithms and some graph properties. A graph is a tree if and only if:
  - It is connected (all vertices are reachable from any vertex).
  - It contains no cycles (it's acyclic).
  - It has exactly one less edge than the number of vertices (|E| = |V| - 1).

Tree



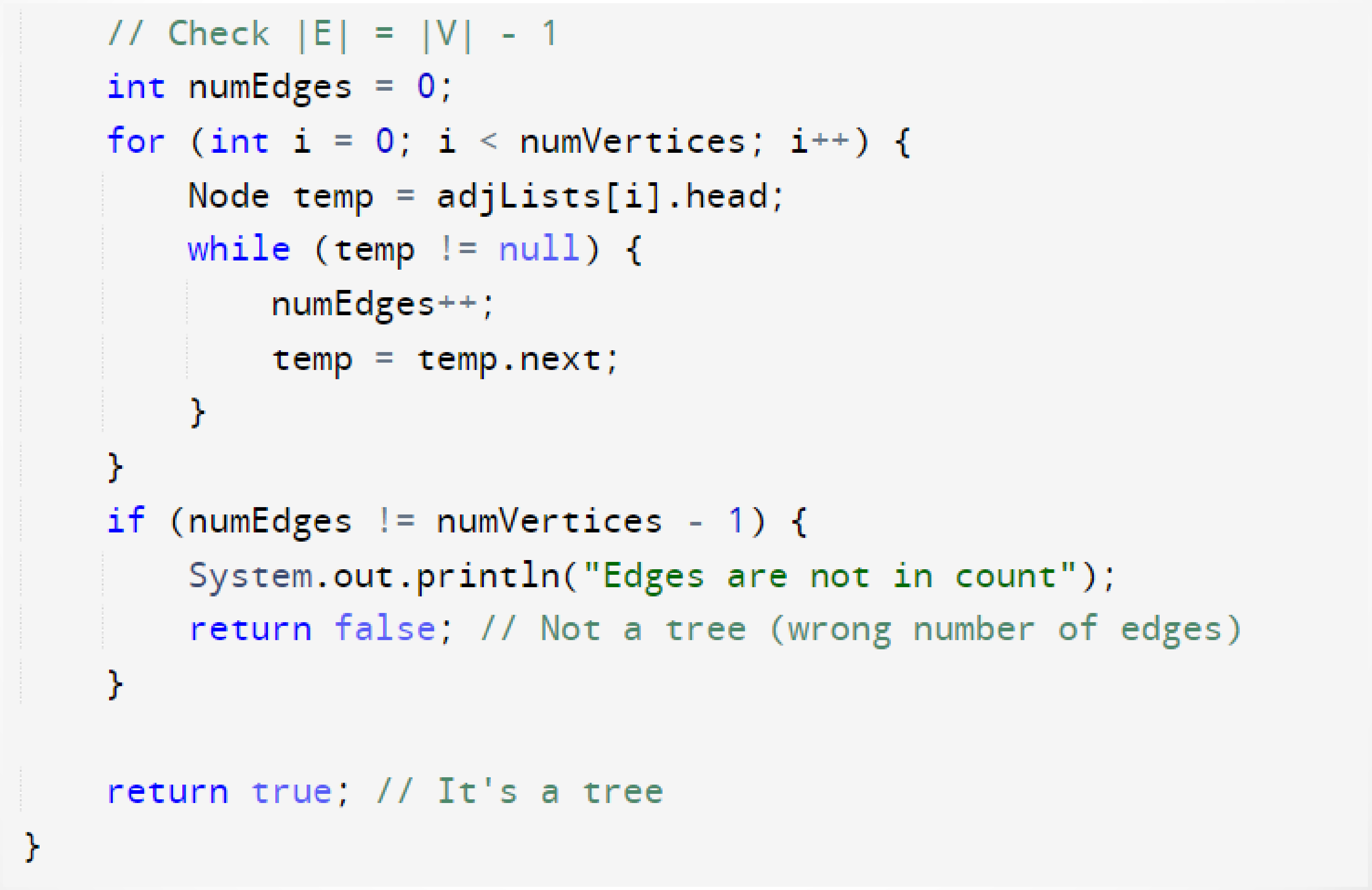
```java
boolean DFS(int vertex, int[] visited, int parent) {
    visited[vertex] = 1;

    Node temp = adjLists[vertex].head;
    while (temp != null) {
        int adjVertex = temp.vertex;
        if (visited[adjVertex] == 0) {
            if (DFS(adjVertex, visited, vertex)) {
                return true;
            }
        } else if (adjVertex != parent) {
            return true;
        }
        temp = temp.next;
    }

    return false;
}
```

Tree



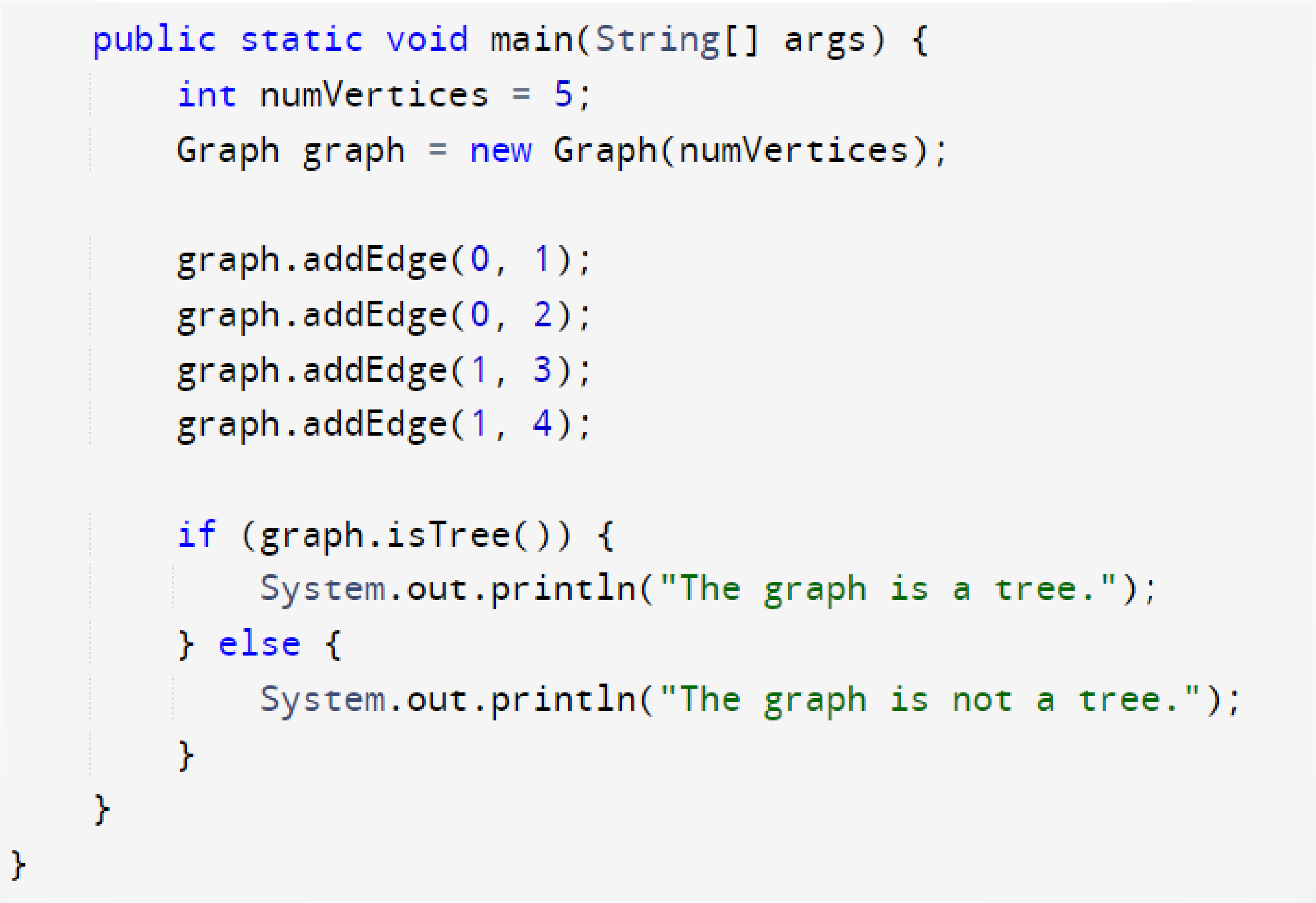
```
boolean isTree() {
    int[] visited = new int[numVertices];

    // Check for cycles
    if (DFS(0, visited, -1)) {
        System.out.println("Cycle found");
        return false; // Not a tree (cycle found)
    }

    // Check for connectivity
    for (int i = 0; i < numVertices; i++) {
        if (visited[i] == 0) {
            System.out.println("Not Connected");
            return false; // Not a tree (not connected)
        }
    }
```

Tree



```
    // Check |E| = |V| - 1
    int numEdges = 0;
    for (int i = 0; i < numVertices; i++) {
        Node temp = adjLists[i].head;
        while (temp != null) {
            numEdges++;
            temp = temp.next;
        }
    }
    if (numEdges != numVertices - 1) {
        System.out.println("Edges are not in count");
        return false; // Not a tree (wrong number of edges)
    }

    return true; // It's a tree
}
```

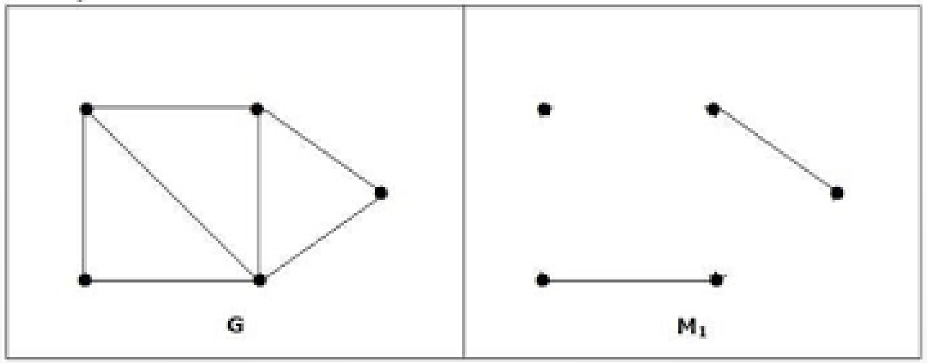
Tree



```
    public static void main(String[] args) {
        int numVertices = 5;
        Graph graph = new Graph(numVertices);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);

        if (graph.isTree()) {
            System.out.println("The graph is a tree.");
        } else {
            System.out.println("The graph is not a tree.");
        }
    }
}
```

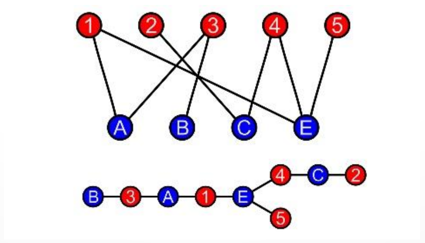
Matching

- In graph theory, a "matching" refers to a set of edges in an undirected graph where no two edges share a common vertex.
- In other words, a matching is a subset of edges in which no two edges have a common endpoint.
- Matchings are often used in various applications such as optimization problems, bipartite graphs, and network flow algorithms.



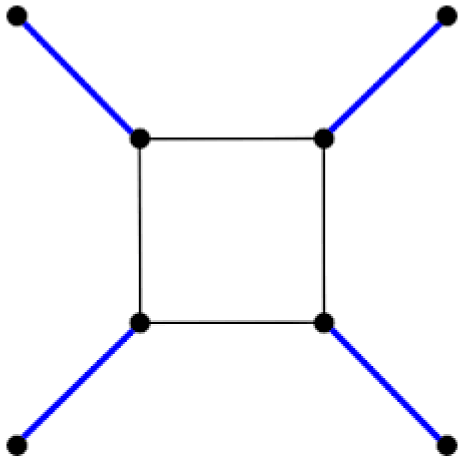
Bipartite graph

- A bipartite graph is a type of graph whose vertices can be divided into two disjoint sets such that no two vertices within the same set are connected by an edge.
- In other words, there are no edges between vertices within the same set, only between vertices in different sets.
- This property makes bipartite graphs particularly useful for modeling relationships where there is a clear distinction between two distinct groups.

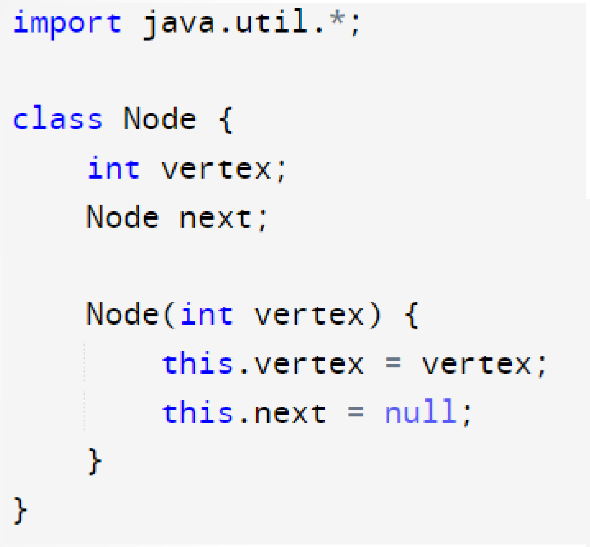


Maximum matching

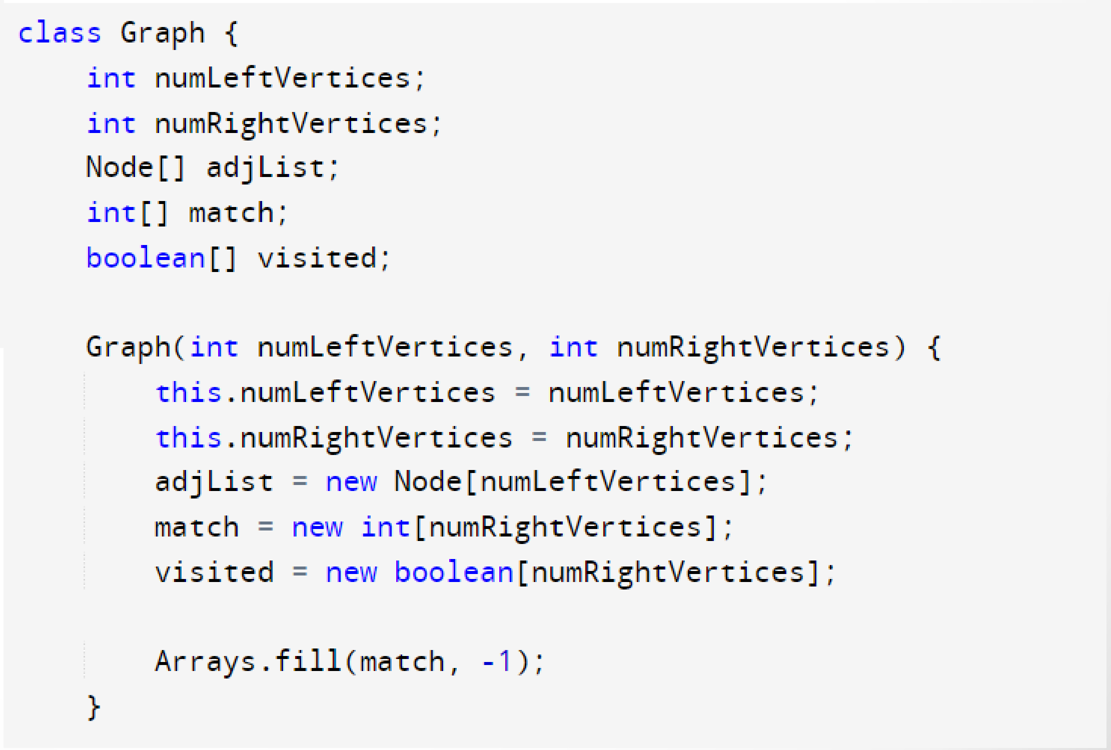
Maximum Matching: A maximum matching is a matching that contains the maximum possible number of edges. It's not necessarily a perfect matching. Finding a maximum matching is a common optimization problem in graph theory.



Maximum matching



```java
import java.util.*;

class Node {
    int vertex;
    Node next;

    Node(int vertex) {
        this.vertex = vertex;
        this.next = null;
    }
}
```



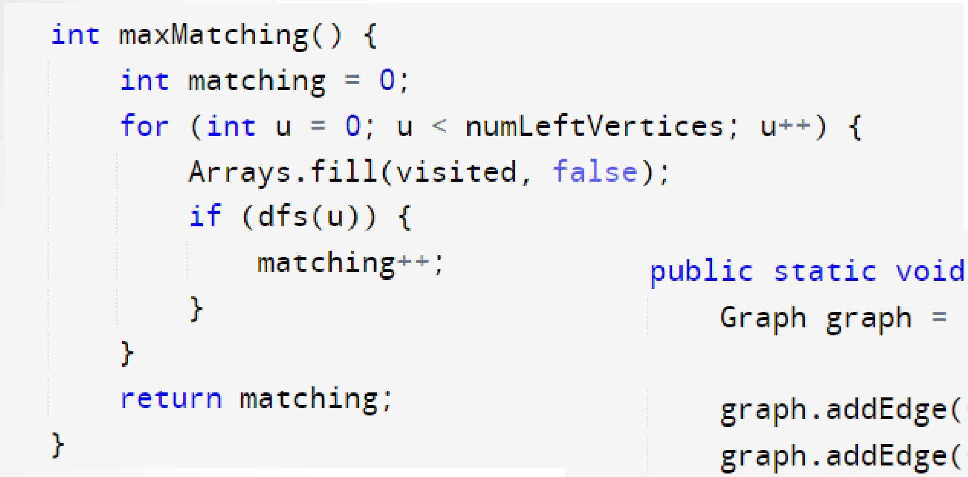
```java
class Graph {
    int numLeftVertices;
    int numRightVertices;
    Node[] adjList;
    int[] match;
    boolean[] visited;

    Graph(int numLeftVertices, int numRightVertices) {
        this.numLeftVertices = numLeftVertices;
        this.numRightVertices = numRightVertices;
        adjList = new Node[numLeftVertices];
        match = new int[numRightVertices];
        visited = new boolean[numRightVertices];

        Arrays.fill(match, -1);
    }
```

Maximum matching

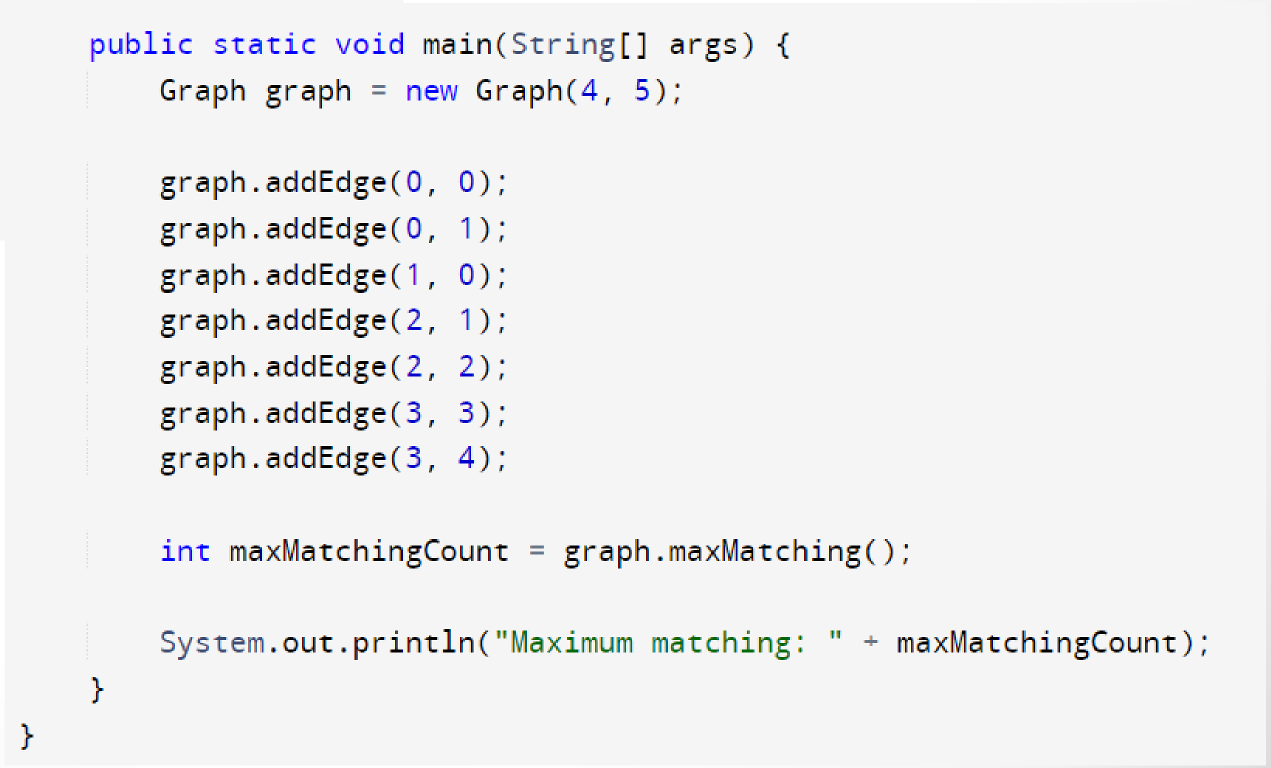
```
void addEdge(int u, int v) {
    Node newNode = new Node(v);
    newNode.next = adjList[u];
    adjList[u] = newNode;
}
```

```
boolean dfs(int u) {
    if (u != -1) {
        Node temp = adjList[u];
        while (temp != null) {
            int v = temp.vertex;
            if (!visited[v]) {
                visited[v] = true;
                if (match[v] == -1 || dfs(match[v])) {
                    match[u] = v;
                    match[v] = u;
                    return true;
                }
            }
            temp = temp.next;
        }
        return false;
    }
    return true;
}
```

Maximum matching



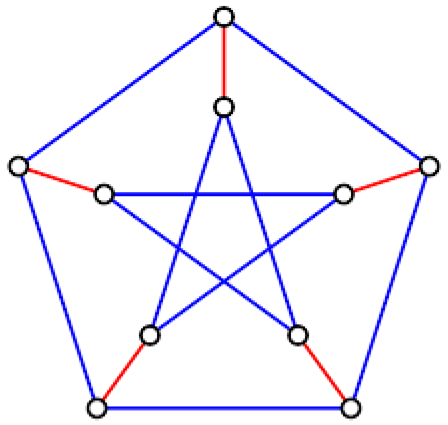
```
int maxMatching() {
    int matching = 0;
    for (int u = 0; u < numLeftVertices; u++) {
        Arrays.fill(visited, false);
        if (dfs(u)) {
            matching++;
        }
    }
    return matching;
}
```



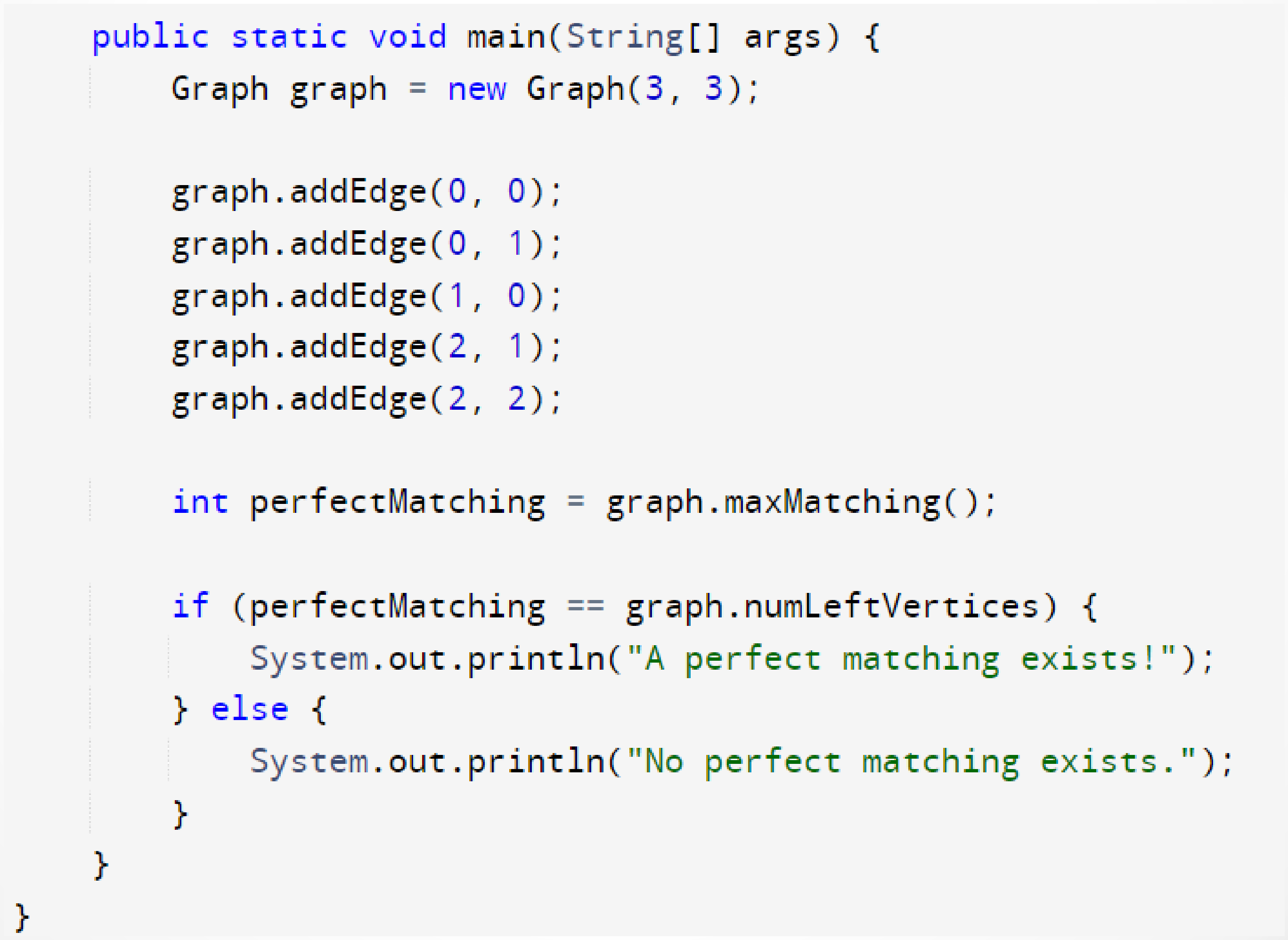
```
public static void main(String[] args) {
    Graph graph = new Graph(4, 5);

    graph.addEdge(0, 0);
    graph.addEdge(0, 1);
    graph.addEdge(1, 0);
    graph.addEdge(2, 1);
    graph.addEdge(2, 2);
    graph.addEdge(3, 3);
    graph.addEdge(3, 4);

    int maxMatchingCount = graph.maxMatching();

    System.out.println("Maximum matching: " + maxMatchingCount);
  }
}
```

Perfect matching

- **Perfect Matching**: A perfect matching is a matching in which every vertex in the graph is incident to exactly one edge from the matching. In other words, every vertex is matched.



Perfect matching



```
    public static void main(String[] args) {
        Graph graph = new Graph(3, 3);

        graph.addEdge(0, 0);
        graph.addEdge(0, 1);
        graph.addEdge(1, 0);
        graph.addEdge(2, 1);
        graph.addEdge(2, 2);

        int perfectMatching = graph.maxMatching();

        if (perfectMatching == graph.numLeftVertices) {
            System.out.println("A perfect matching exists!");
        } else {
            System.out.println("No perfect matching exists.");
        }
    }
}
```

Network

- In graph theory, a network refers to a collection of interconnected nodes (also called vertices) and edges (also called links) that represent relationships between those nodes.
- In some contexts, a network refers to a graph that represents the flow of something between nodes.
- Networks are used to model a wide range of complex systems, from social interactions to computer networks, transportation systems, and more.

Network

- **Flow Networks**: These are graphs where edges have capacities (maximum flow) and can represent things like data flow in computer networks, fluid flow in pipelines, etc.
- **Transportation Networks**: These models transportation systems, where vertices represent locations, and edges represent connections between them (e.g., road networks, flight routes).
- **Social Networks**: These represent relationships between individuals, where nodes are people, and edges represent interactions (e.g., friendships, collaborations).
- **Communication Networks**: These are used to model communication systems, where nodes are devices, and edges represent communication links (e.g., computer networks, the internet).
- **Electrical Networks**: Used in electrical engineering to represent circuits, with nodes as components and edges as connections.

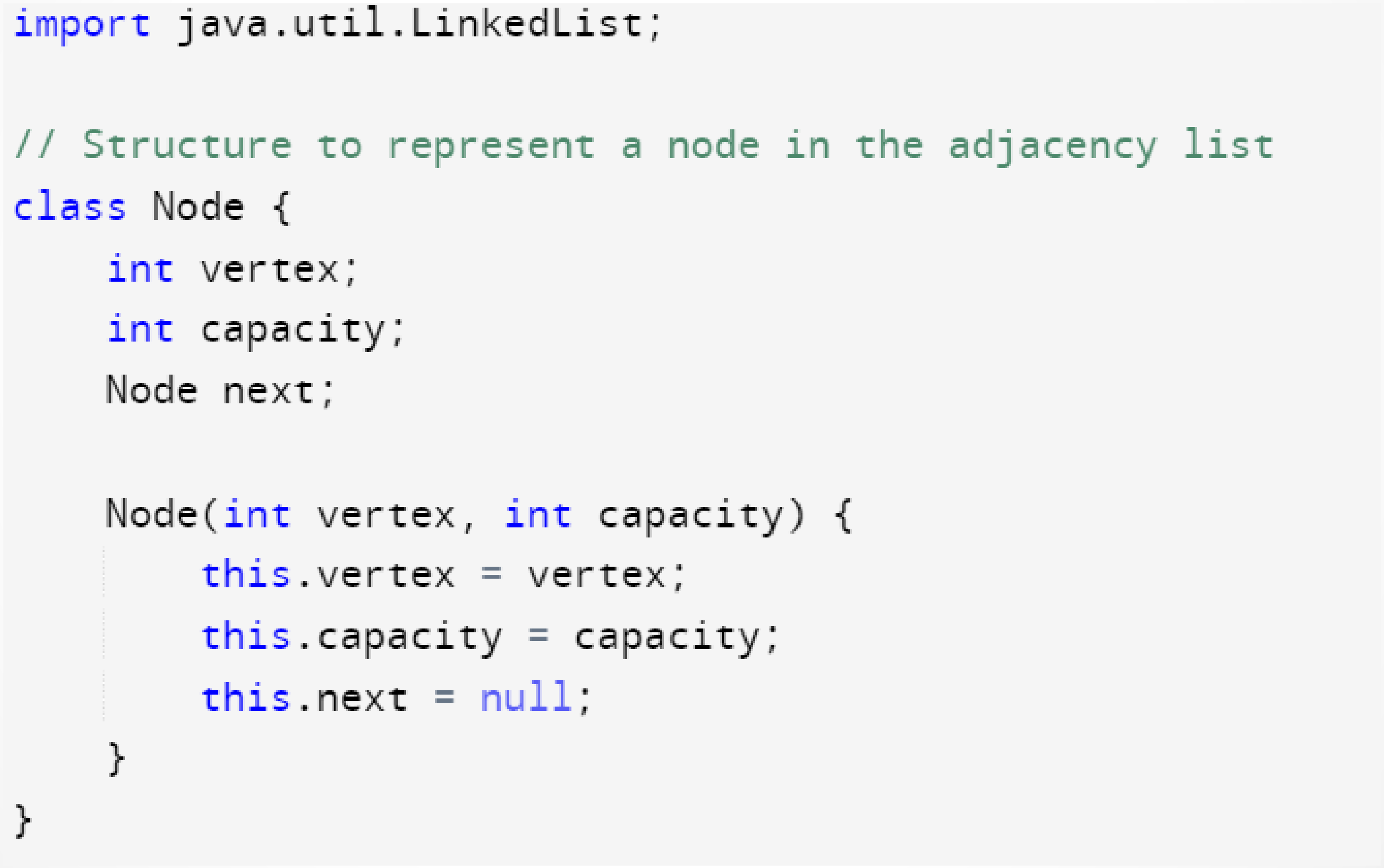
Flow networks

- Key Concepts in Flow Networks:
- **Nodes (Vertices)**: Nodes represent points or locations in the network where the flow can be sent, received, or passed through. In a flow network, you typically have a source node (where the flow originates) and a sink node (where the flow terminates).
- **Edges**: Edges represent connections between nodes in the network. Each edge has a capacity, which indicates the maximum amount of flow it can carry.
- **Flow**: The flow represents the quantity of a resource (e.g., goods, data) moving through the edges of the network. The flow on an edge must not exceed its capacity.
- **Source and Sink**: The source node is where the flow originates, and the sink node is where the flow terminates. The goal is often to maximize the flow from the source to the sink while adhering to capacity constraints.
- **Residual Capacity**: The difference between the capacity of an edge and the flow currently passing through it. If the flow is less than the capacity, there is residual capacity available for additional flow.
- **Residual Graph**: The residual graph is a representation of the remaining capacity on edges after some flow has been passed through. It's used in algorithms like the Ford-Fulkerson algorithm.

Flow networks

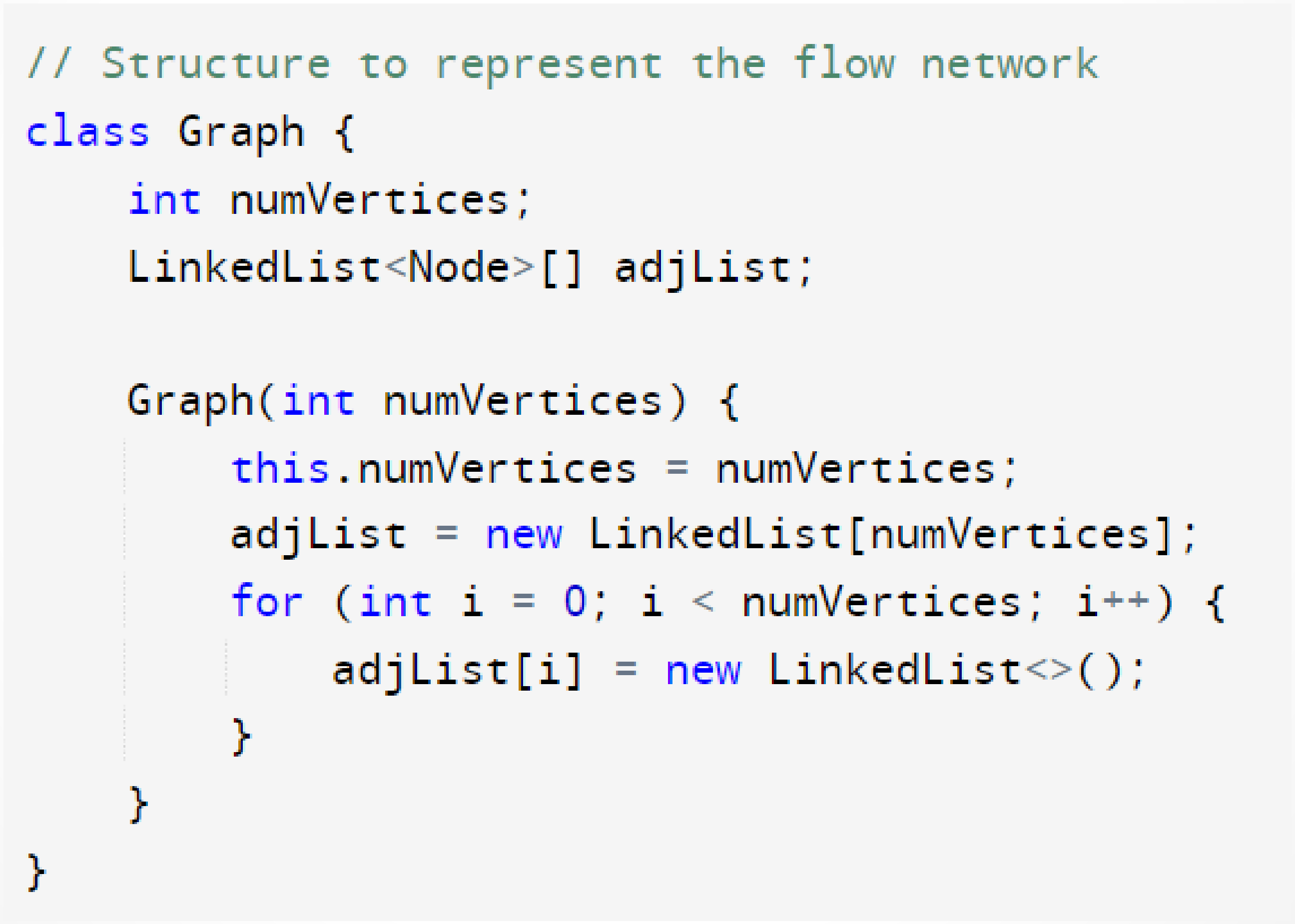
- Flow Networks Applications:
- **Transportation and Distribution Networks**: Flow networks can model the distribution of goods, materials, or products through a network of warehouses, factories, and distribution centers.
- **Telecommunication Networks**: In data networks, flow networks can represent the transmission of data packets through routers and switches with limited bandwidth.
- **Max Flow-Min Cut Theorem**: Flow networks are used to solve problems related to maximum flow and minimum cut. The Max Flow-Min Cut Theorem states that the maximum flow in a network is equal to the capacity of the minimum cut.
- **Network Flows in Operations Research**: Flow networks are used in operations research to model transportation, supply chain, and resource allocation problems.
- **Image Segmentation**: Flow networks can be used in image processing for segmenting images into distinct regions based on certain criteria.

Ford-Fulkerson algorithm



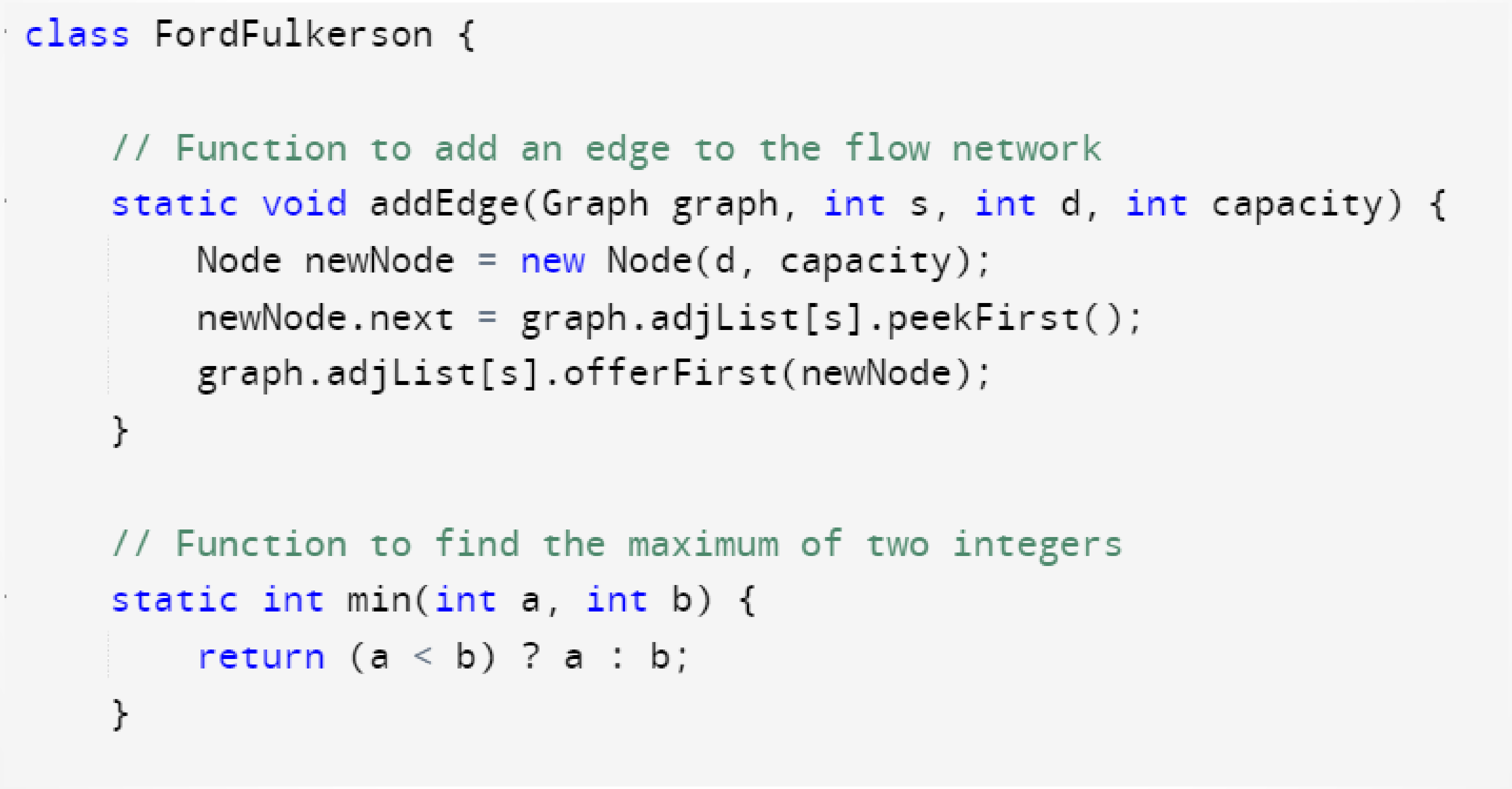
```java
import java.util.LinkedList;

// Structure to represent a node in the adjacency list
class Node {
    int vertex;
    int capacity;
    Node next;

    Node(int vertex, int capacity) {
        this.vertex = vertex;
        this.capacity = capacity;
        this.next = null;
    }
}
```

Ford-Fulkerson algorithm



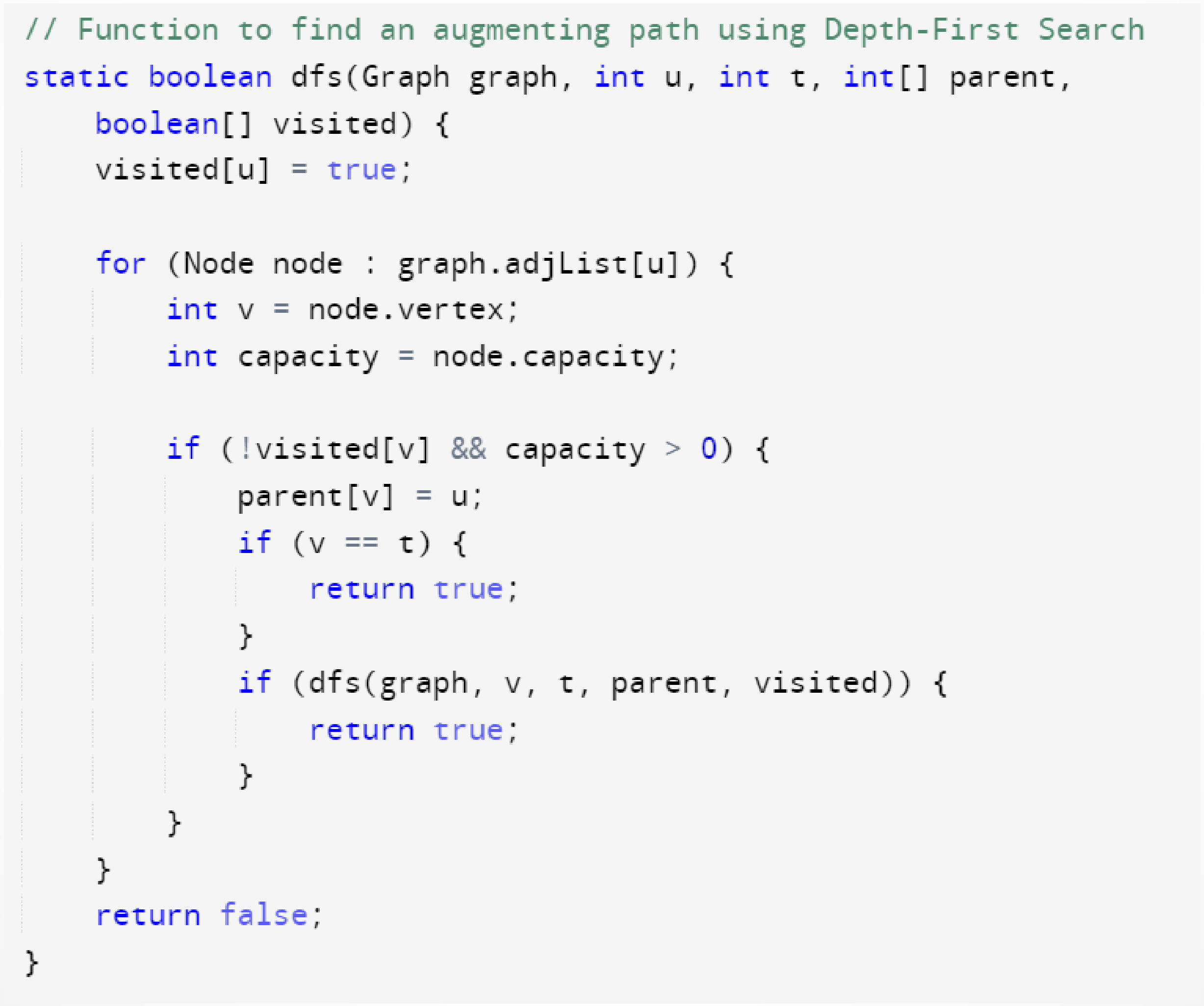
```
// Structure to represent the flow network
class Graph {
    int numVertices;
    LinkedList<Node>[] adjList;

    Graph(int numVertices) {
        this.numVertices = numVertices;
        adjList = new LinkedList[numVertices];
        for (int i = 0; i < numVertices; i++) {
            adjList[i] = new LinkedList<>();
        }
    }
}
```

Ford-Fulkerson algorithm



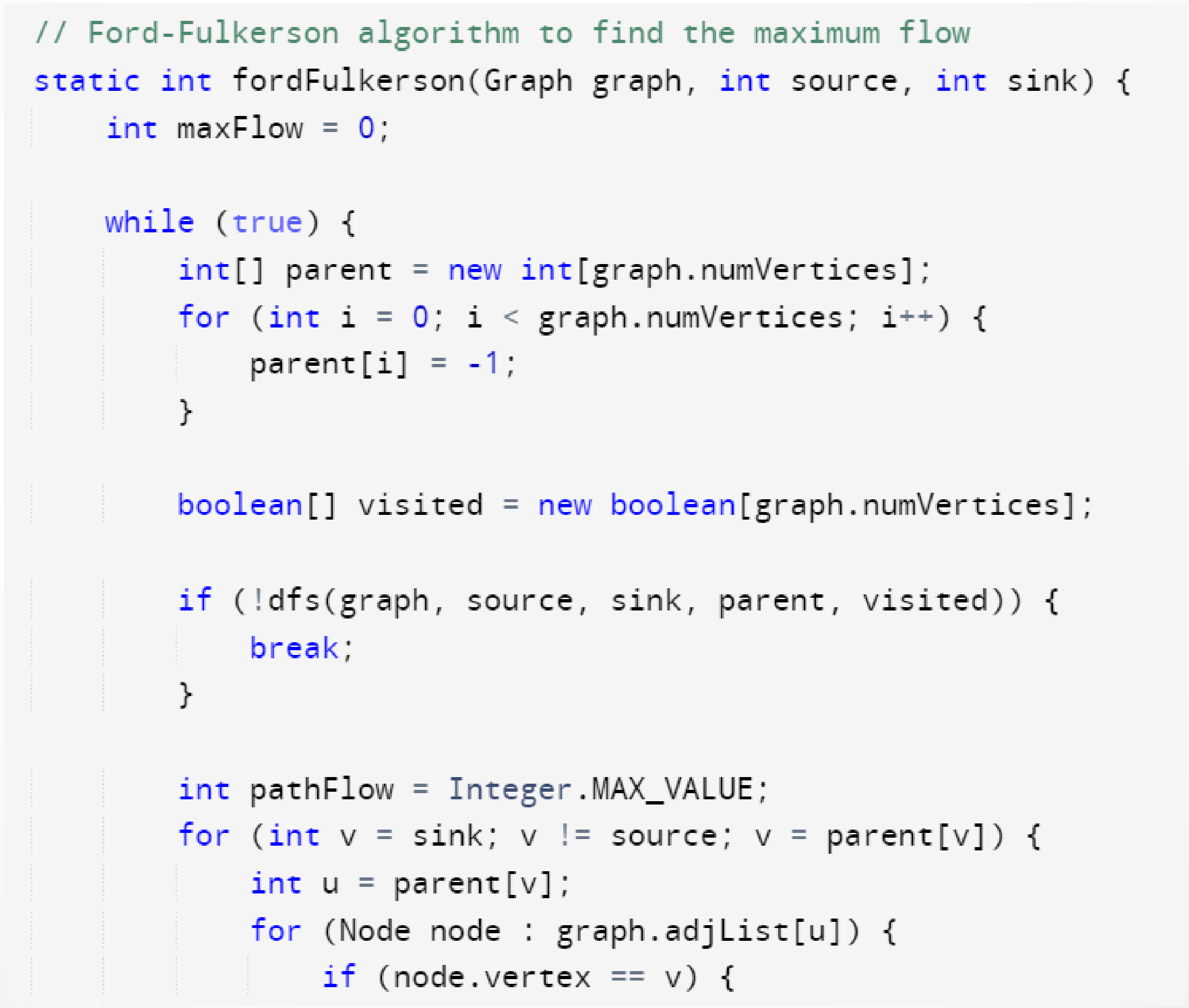
```java
class FordFulkerson {

    // Function to add an edge to the flow network
    static void addEdge(Graph graph, int s, int d, int capacity) {
        Node newNode = new Node(d, capacity);
        newNode.next = graph.adjList[s].peekFirst();
        graph.adjList[s].offerFirst(newNode);
    }

    // Function to find the maximum of two integers
    static int min(int a, int b) {
        return (a < b) ? a : b;
    }
```

Ford-Fulkerson algorithm



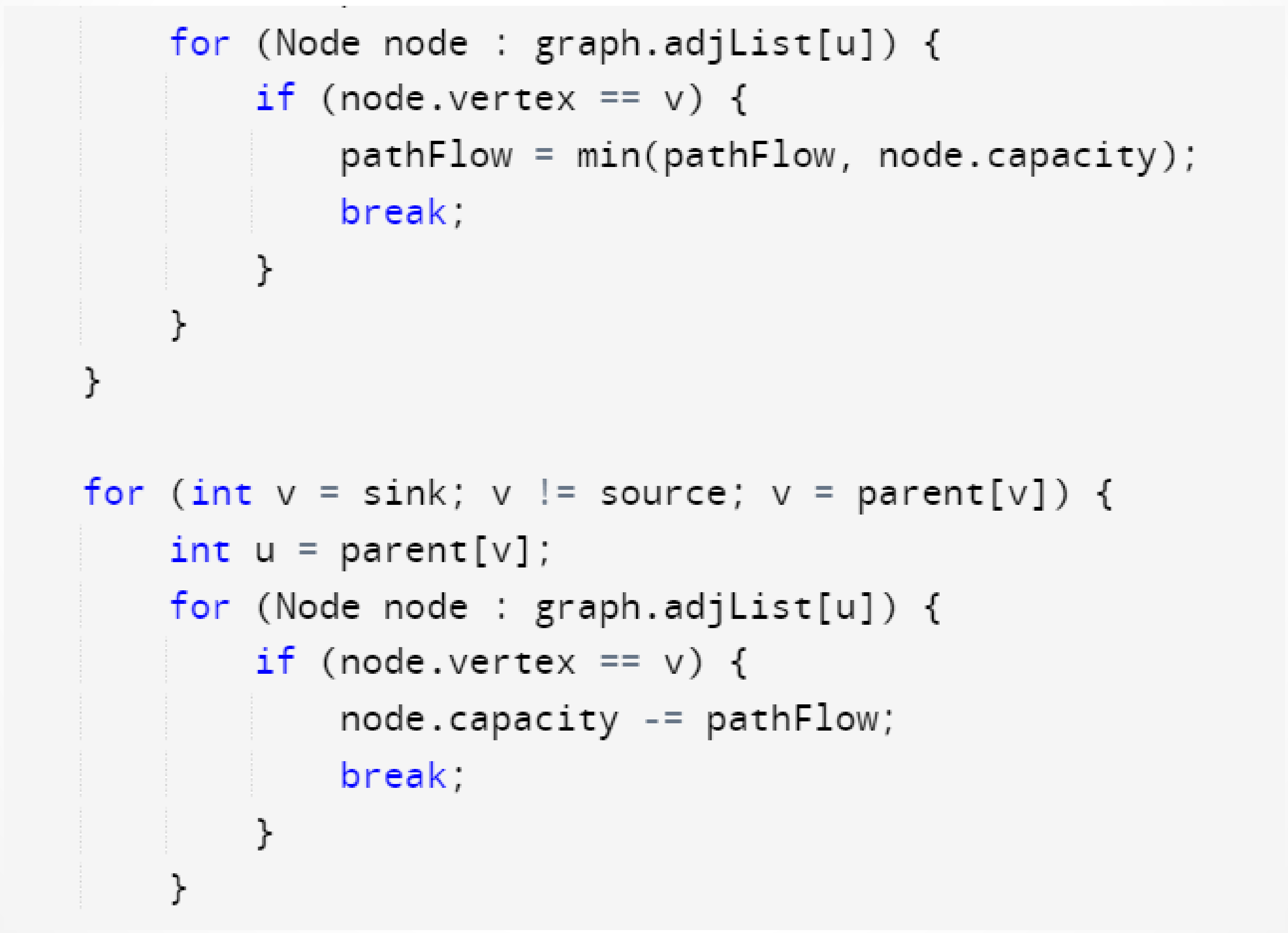
```
// Function to find an augmenting path using Depth-First Search
static boolean dfs(Graph graph, int u, int t, int[] parent,
    boolean[] visited) {
    visited[u] = true;

    for (Node node : graph.adjList[u]) {
        int v = node.vertex;
        int capacity = node.capacity;

        if (!visited[v] && capacity > 0) {
            parent[v] = u;
            if (v == t) {
                return true;
            }
            if (dfs(graph, v, t, parent, visited)) {
                return true;
            }
        }
    }
    return false;
}
```

Ford-Fulkerson algorithm



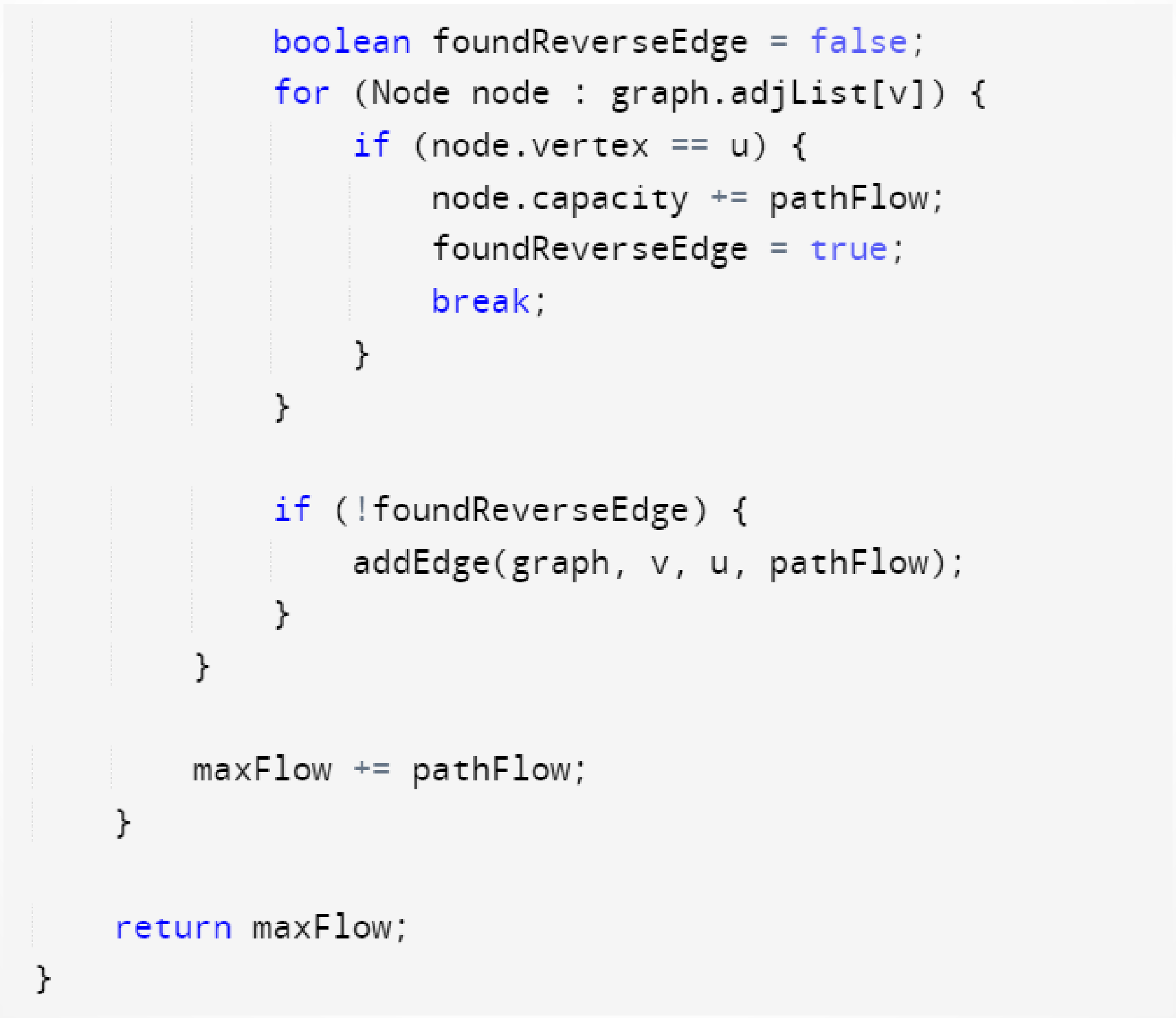
```java
// Ford-Fulkerson algorithm to find the maximum flow
static int fordFulkerson(Graph graph, int source, int sink) {
    int maxFlow = 0;

    while (true) {
        int[] parent = new int[graph.numVertices];
        for (int i = 0; i < graph.numVertices; i++) {
            parent[i] = -1;
        }

        boolean[] visited = new boolean[graph.numVertices];

        if (!dfs(graph, source, sink, parent, visited)) {
            break;
        }

        int pathFlow = Integer.MAX_VALUE;
        for (int v = sink; v != source; v = parent[v]) {
            int u = parent[v];
            for (Node node : graph.adjList[u]) {
                if (node.vertex == v) {
```

Ford-Fulkerson algorithm



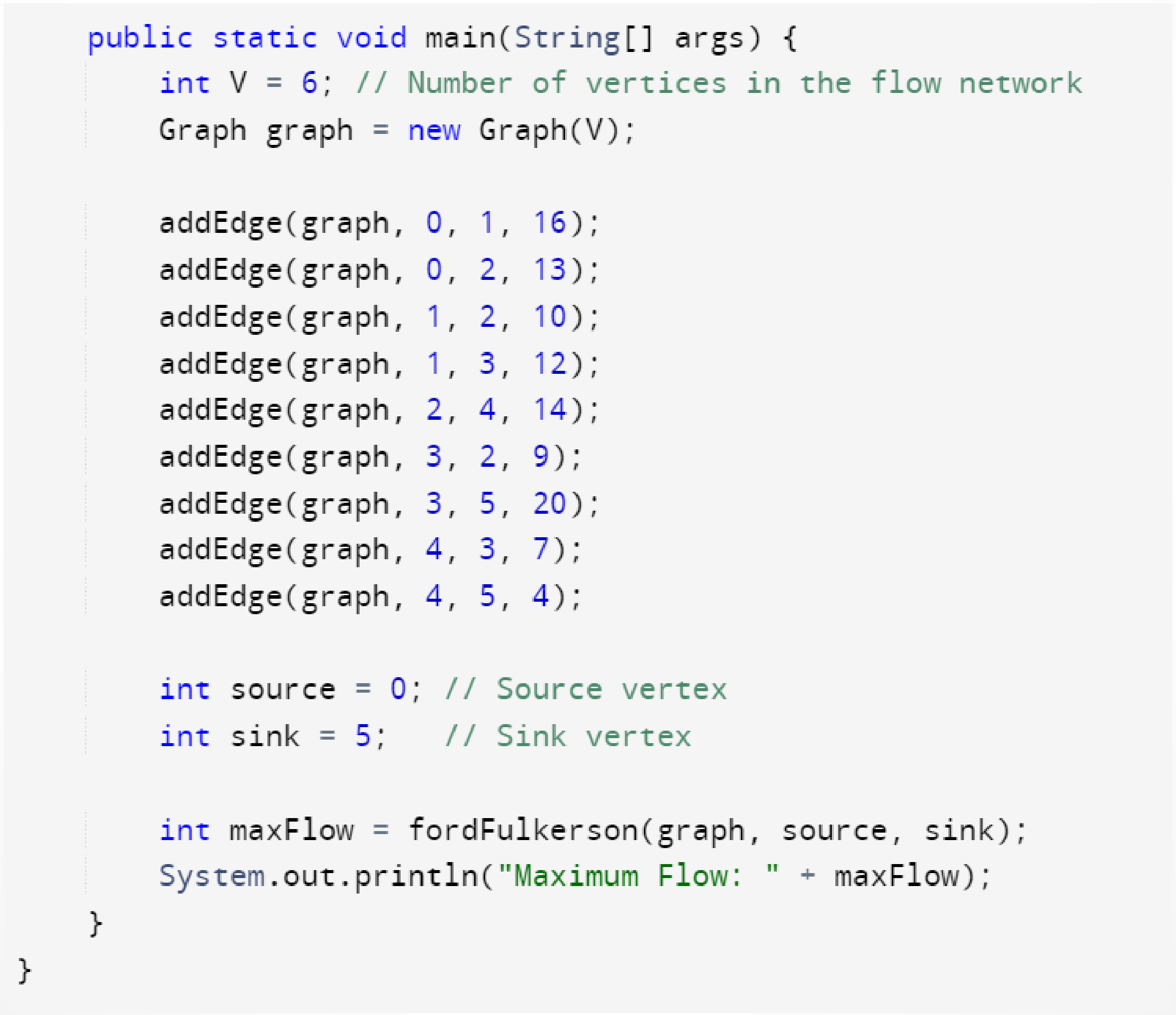
```
        for (Node node : graph.adjList[u]) {
            if (node.vertex == v) {
                pathFlow = min(pathFlow, node.capacity);
                break;
            }
        }
    }

    for (int v = sink; v != source; v = parent[v]) {
        int u = parent[v];
        for (Node node : graph.adjList[u]) {
            if (node.vertex == v) {
                node.capacity -= pathFlow;
                break;
            }
        }
```

Ford-Fulkerson algorithm



```
            boolean foundReverseEdge = false;
            for (Node node : graph.adjList[v]) {
                if (node.vertex == u) {
                    node.capacity += pathFlow;
                    foundReverseEdge = true;
                    break;
                }
            }

            if (!foundReverseEdge) {
                addEdge(graph, v, u, pathFlow);
            }
        }

        maxFlow += pathFlow;
    }

    return maxFlow;
}
```

Ford-Fulkerson algorithm



```java
    public static void main(String[] args) {
        int V = 6; // Number of vertices in the flow network
        Graph graph = new Graph(V);

        addEdge(graph, 0, 1, 16);
        addEdge(graph, 0, 2, 13);
        addEdge(graph, 1, 2, 10);
        addEdge(graph, 1, 3, 12);
        addEdge(graph, 2, 4, 14);
        addEdge(graph, 3, 2, 9);
        addEdge(graph, 3, 5, 20);
        addEdge(graph, 4, 3, 7);
        addEdge(graph, 4, 5, 4);

        int source = 0; // Source vertex
        int sink = 5;   // Sink vertex

        int maxFlow = fordFulkerson(graph, source, sink);
        System.out.println("Maximum Flow: " + maxFlow);
    }
}
```

Summary

- Representation of a graph with use of the adjacency matrix and adjacency list.
- Representation of weighted and directed graphs.
- Observe whether a graph is connected or represent a tree.
- Find the matching of a graph.
- Factorize a graph.
- Use of graphs in networks.

Brief of the Next Lecture

- We are going to explore more about trees and distance in next lecture.

Q & A

References

- Clark J. and Holton, D.A. "A first look at graph theory", Allied publishers, 1995.
- Douglas B. West, Introduction to Graph Theory, 2nd edition.

Thank You.