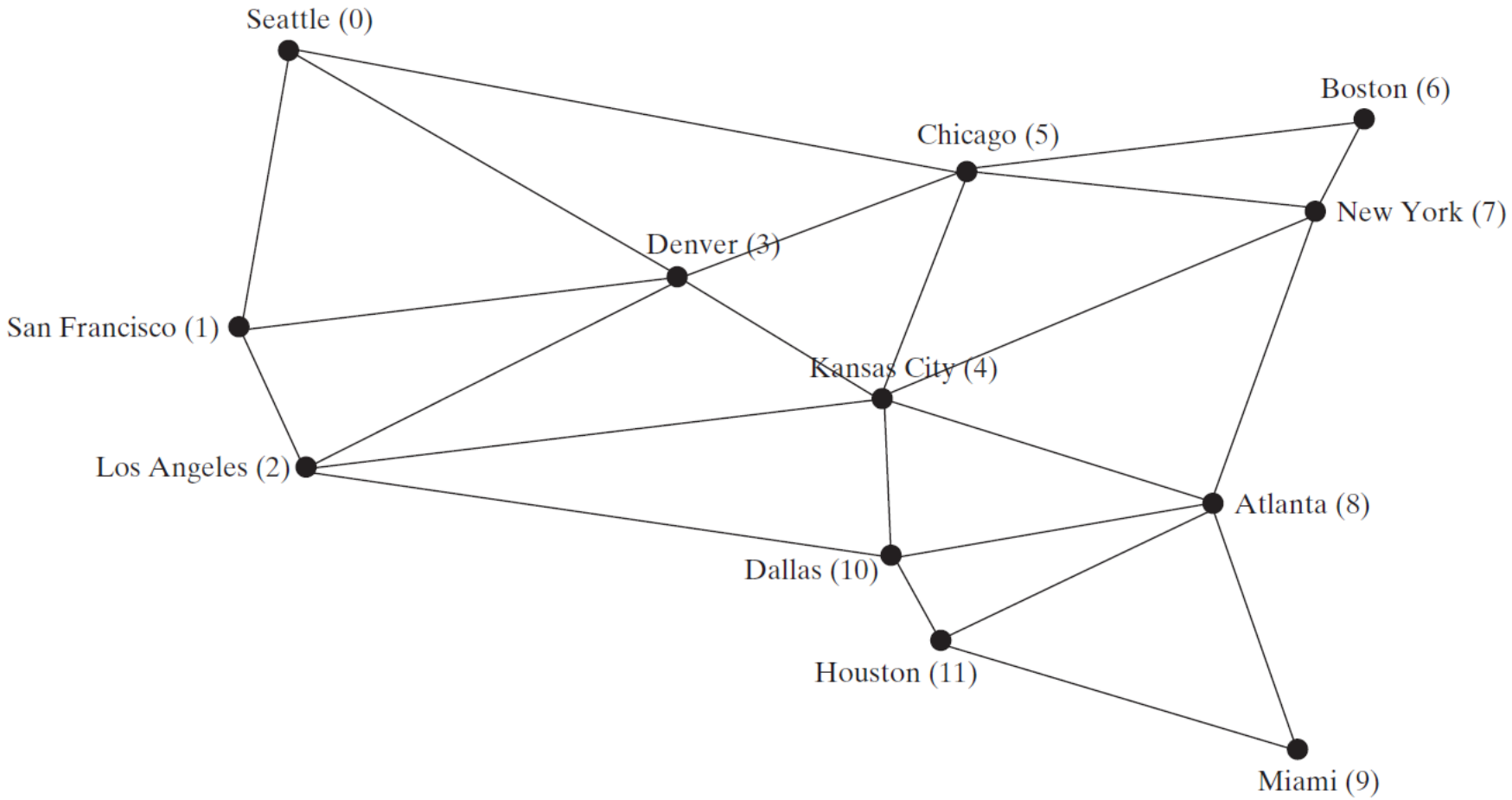
# Graphs


Seattle (0)
Boston (6)
Chicago (5)
New York (7)
Denver (3)
San Francisco (1)
Kansas City (4)
Los Angeles (2)
Atlanta (8)
Dallas (10)
Houston (11)
Miami (9)

# Intro

- The study of graph problems is known as **graph theory**.
- Graph theory was founded by Leonhard Euler in 1736, when he introduced graph terminology to solve the famous *Seven Bridges of Königsberg* problem.
  - The city of *Königsberg*, Prussia (now Kaliningrad, Russia), was divided by the Pregel River.

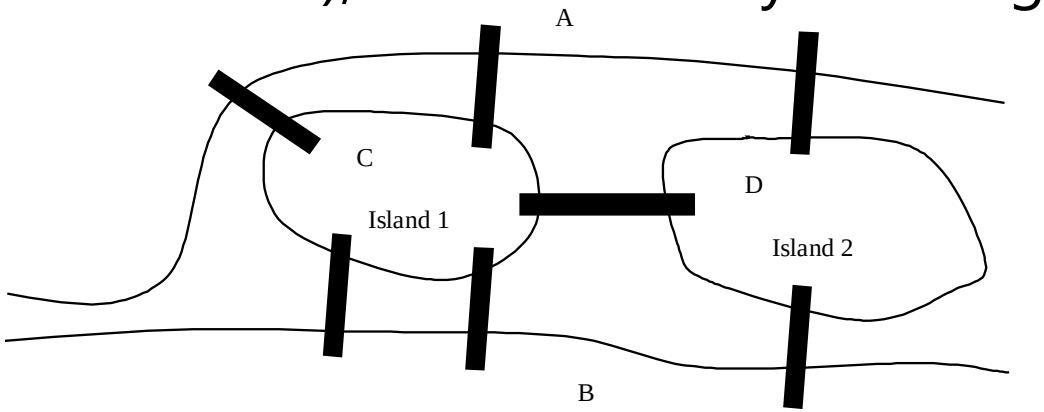
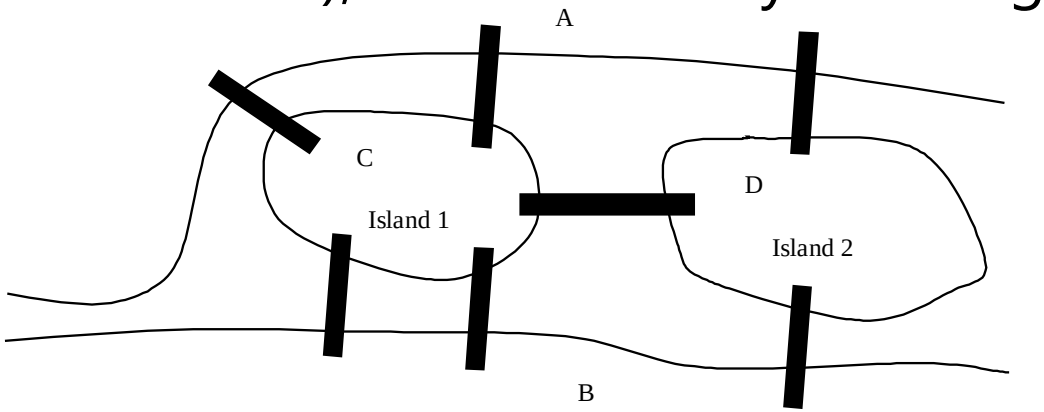




Image Source: https://en.wikipedia.org/wiki/Leonhard_Euler

# Intro

- The study of graph problems is known as **graph theory**.
- Graph theory was founded by Leonhard Euler in 1736, when he introduced graph terminology to solve the famous *Seven Bridges of Königsberg* problem.
  - The city of *Königsberg*, Prussia (now Kaliningrad, Russia), was divided by the Pregel River.



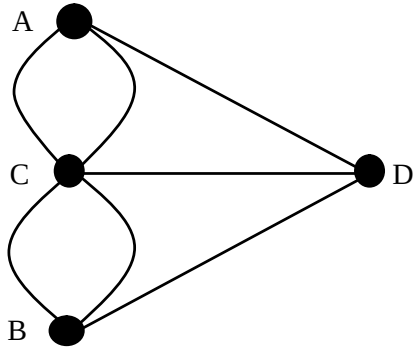





Image Source: https://en.wikipedia.org/wiki/Leonhard_Euler

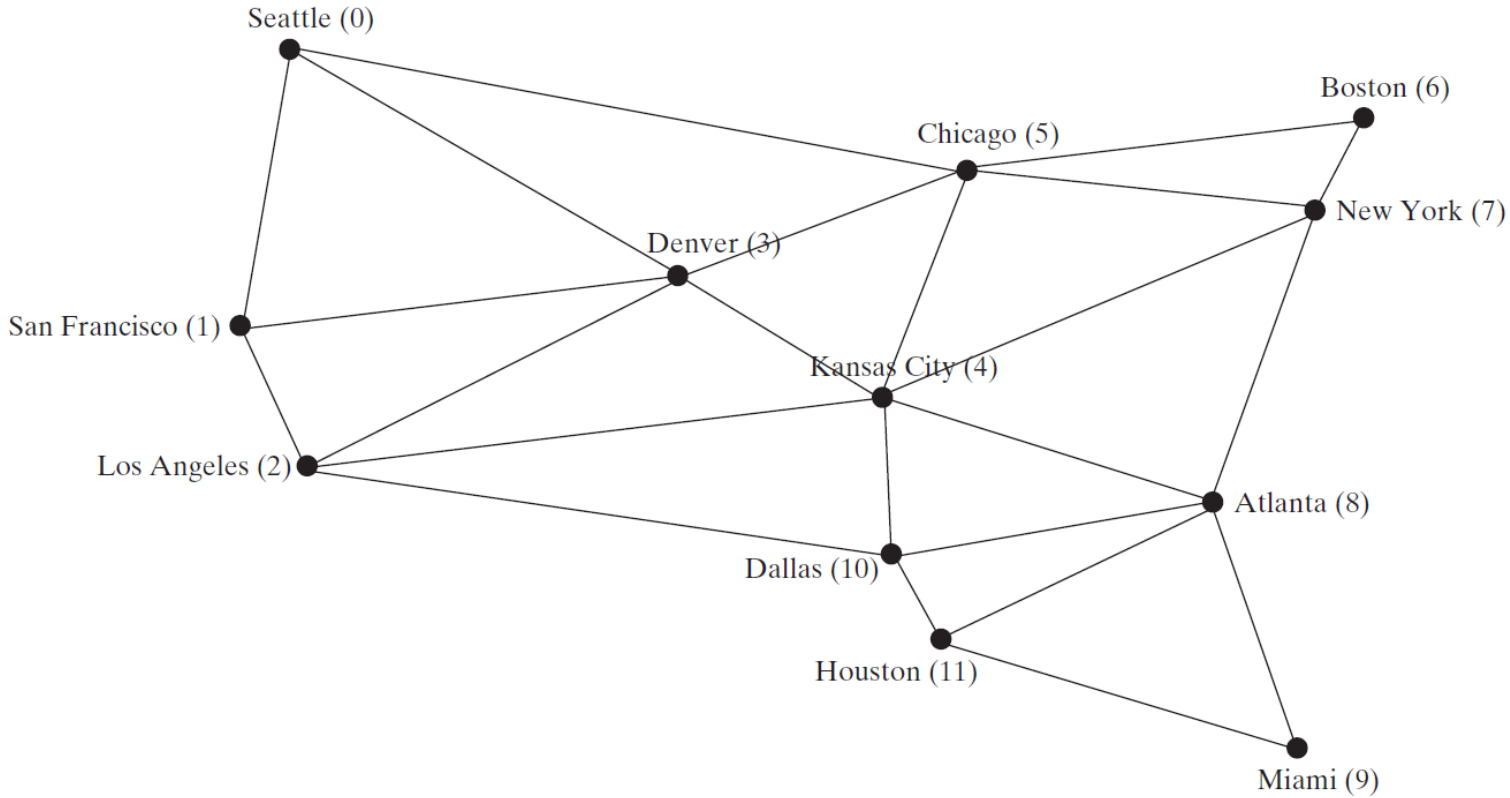
https://projecteuler.net/ ?

# Basic Graph terminology

- What is a graph? A **Graph** is a mathematical structure that represents relationships among entities in the real world.

- A graph consists of a *nonempty* set of **Vertices** (a.k.a. as nodes or points), and a set of **Edges** that connect the vertices.

- For convenience, we define a graph as G = (V, E), where V represents a set of vertices and E represents a set of edges.
  - |V| = number of vertices in G
  - |E| = number of edges in G

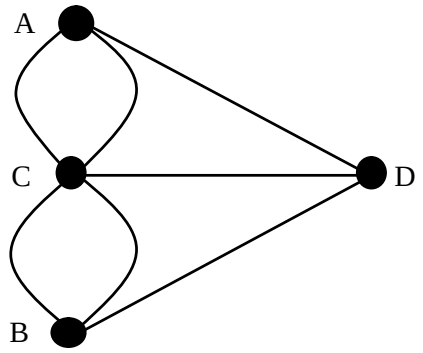


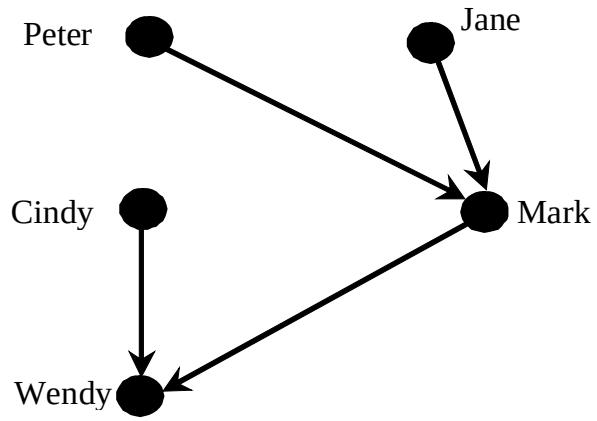

```
V = {"Seattle", "San Francisco", "Los Angeles",
     "Denver", "Kansas City", "Chicago", "Boston", "New York",
     "Atlanta", "Miami", "Dallas", "Houston"};

E = {{"Seattle", "San Francisco"}, {"Seattle", "Chicago"},
     {"Seattle", "Denver"}, {"San Francisco", "Denver"},
     ...
    };
```

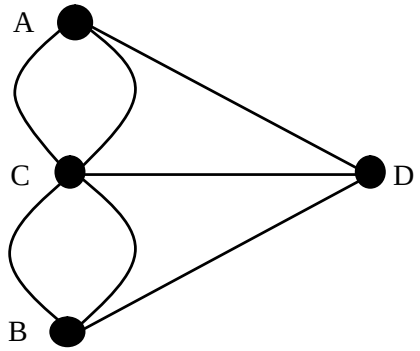
# Directed vs Undirected graph

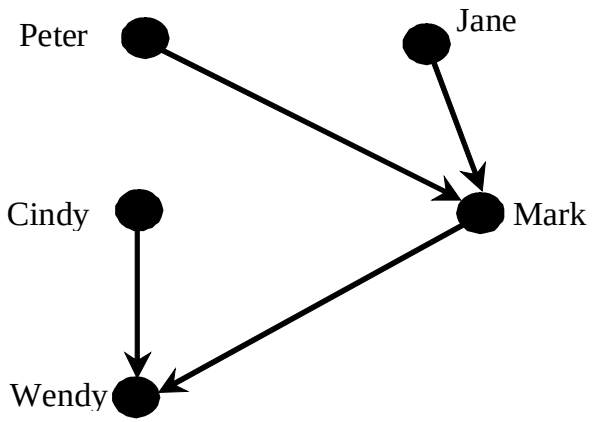
- A graph may be **directed** (digraph) or **undirected**.
- In a directed graph, each edge has a direction, which indicates you can move from one vertex to the other through the edge.

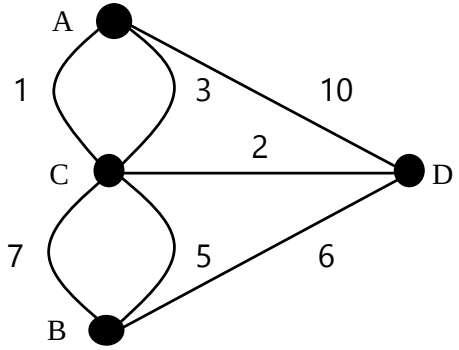


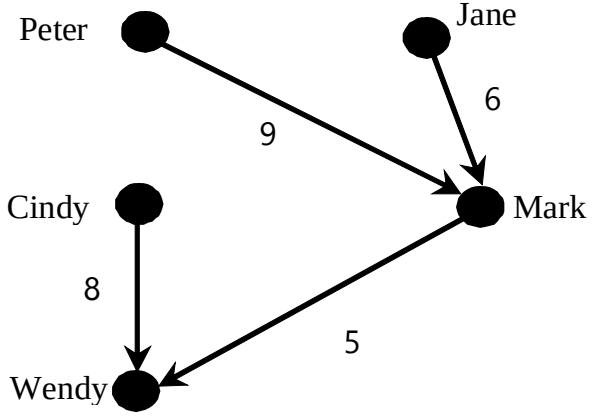



# Weighted vs Unweighted edges



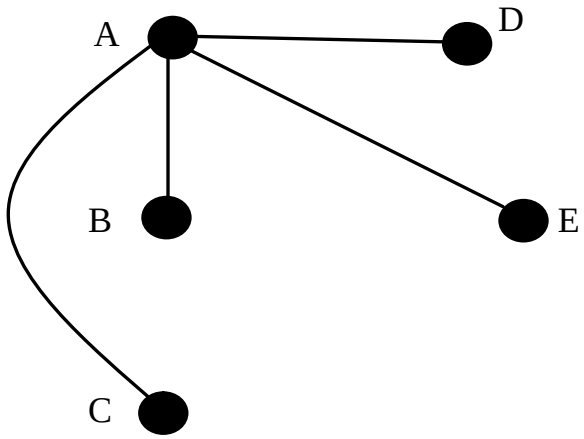



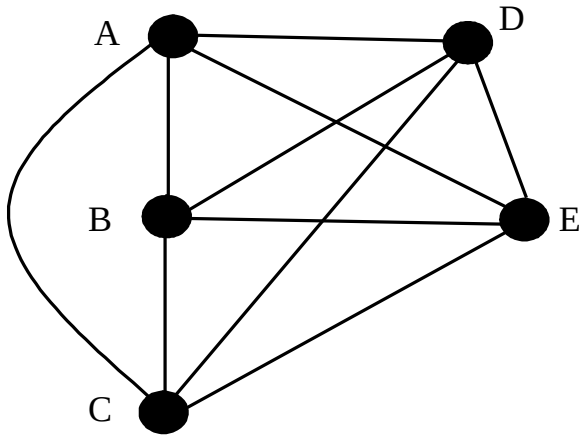
# Basic Graph terminology (cont.)

- Two vertices in a graph are said to be **adjacent** if they are connected by the same edge. Similarly, two edges are said to be **adjacent** if they are connected to the same vertex.
- An edge in a graph that joins two vertices is said to be **incident** to both vertices.
- The **degree** of a vertex is the number of edges incident to it.
- Two vertices are called **neighbors** if they are adjacent. Similarly, two edges are called **neighbors** if they are adjacent.

# Basic Graph terminology (cont.)

- A **loop** is an edge that links a vertex to itself.
- If two vertices are connected by two or more edges, these edges are called **parallel edges**.
- A **simple graph** is one that does not have any loops or parallel edges.
- In a **complete graph**, every two pairs of vertices are connected.

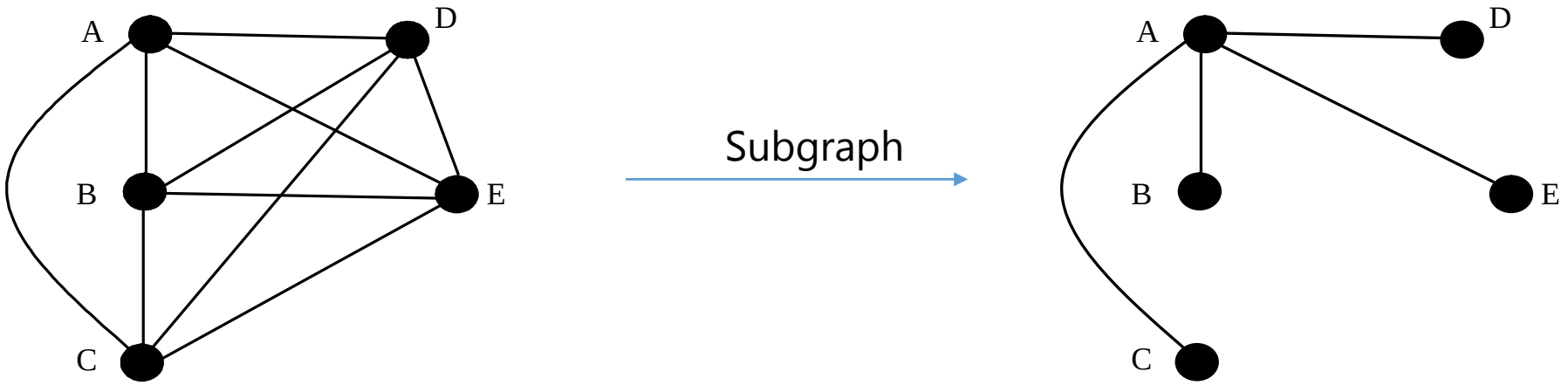


# Basic Graph terminology (cont.)

- A graph is **connected** if there exists a path between any two vertices in the graph.

- A **subgraph** of a graph G is a graph whose vertex set if subset of that of G and whose edge set is a subset of that of G.



# Basic Graph terminology (cont.)

- Assume the graph is connected and undirected.
  - A **cycle** is a closed path that starts from a vertex and ends at the same vertex.
  - A **connected graph** is a **tree** if it does not have cycles.
  - A **spanning tree** of a graph G is a **connected subgraph** of G, and the subgraph is a **tree** that **contains all vertices** in G.

# Representing Graphs: Representing Vertices

```
String[] vertices = {"Seattle", "Des Moines", "Chicago", "Denver", ...};

City[] vertices = {city0, city1, ...};

public class City
{
  ...
}

List<String> vertices;
```

# Representing Graphs: Representing Edges Edge Array

```
int[][] edges = {{0,1},{0,3},{0,5},{1,0},{1,2},...};
```

# Representing Graphs: Representing Edges Edge Object

```
List<Edge> list = new ArrayList<>();
list.add(new Edge(0,1));
list.add(new Edge(0,3));
...
```

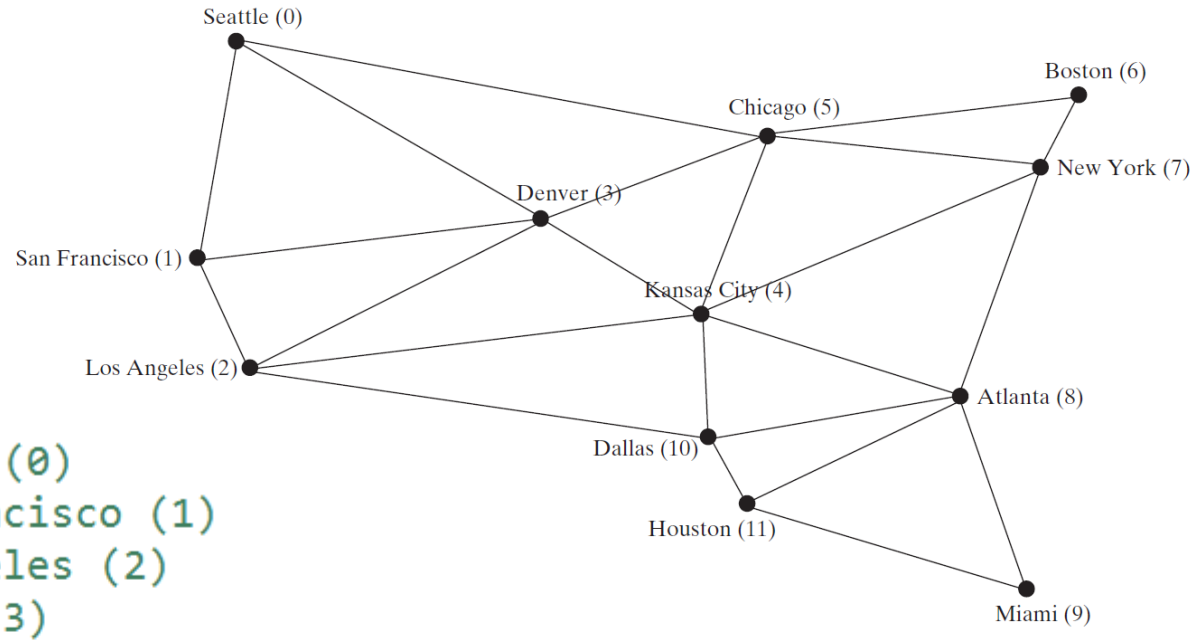
```
public class Edge
{
 int u;
 int v;

 public Edge(int u, int v)
 {
  this.u = u;
  this.v = v;
 }

 public boolean equals(Object o)
 {
  if(o==null || !(o instanceof Edge)) return false;
  if(o==this) return true;

  Edge e = (Edge)o;
  return this.u==e.u && this.v==e.v;
 }
}
```

# Representing Graphs: Representing Edges Adjacency Matrix



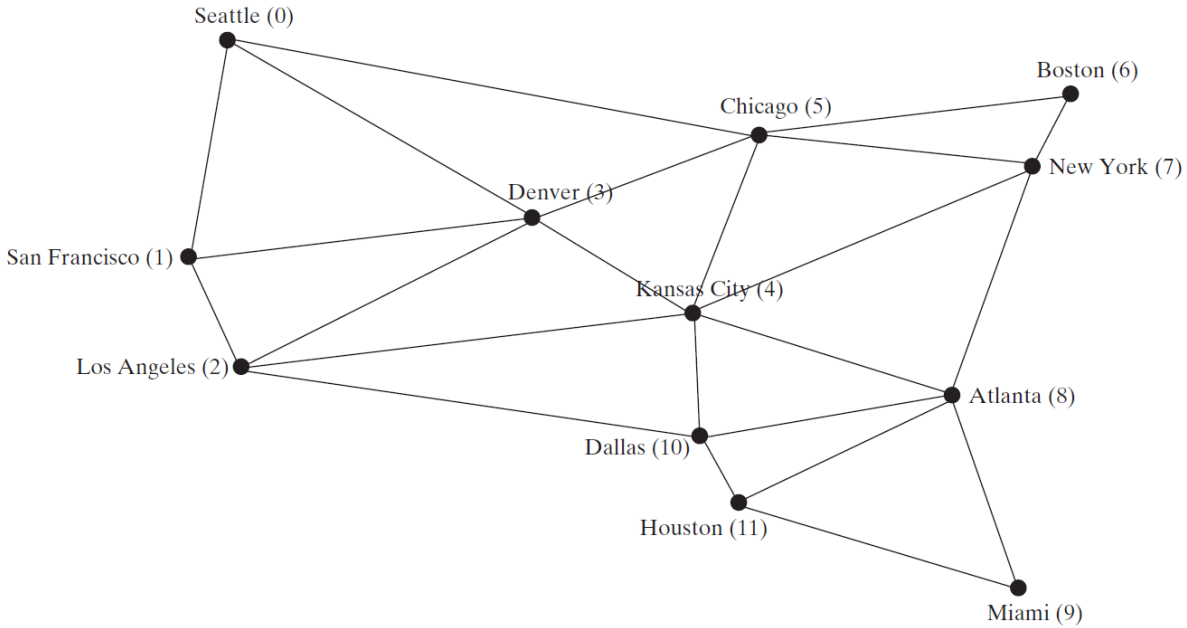

```
int[][] adjacencyMatrix =
 {
  //0  1  2  3  4  5  6  7  8  9  10  11
    {0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0,  0},// Seattle (0)
    {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0,  0},// San Francisco (1)
    {0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1,  0},// Los Angeles (2)
    {1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0,  0},// Denver (3)
    {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1,  0},// Kansas City (4)
    {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0,  0},// Chicago (5)
    {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,  0},// Boston (6)
    {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0,  0},// New York (7)
    {0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1,  1},// Atlanta (8)
    {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,  1},// Miami (9)
    {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0,  1},// Dallas (10)
    {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,  0} // Houston (11)
 };
```

# Representing Graphs: Representing Edges Adjacency Lists

```
List<Integer>[] neighbors = new List[12];
```

Adjacency vertex list

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Seattle | neighbors[0] | 1 | 3 | 5 | | | |
| San Francisco | neighbors[1] | 0 | 2 | 3 | | | |
| Los Angeles | neighbors[2] | 1 | 3 | 4 | 10 | | |
| Denver | neighbors[3] | 0 | 1 | 2 | 4 | 5 | |
| Kansas City | neighbors[4] | 2 | 3 | 5 | 7 | 8 | 10 |
| Chicago | neighbors[5] | 0 | 3 | 4 | 6 | 7 | |
| Boston | neighbors[6] | 5 | 7 | | | | |
| New York | neighbors[7] | 4 | 5 | 6 | 8 | | |
| Atlanta | neighbors[8] | 4 | 7 | 9 | 10 | 11 | |
| Miami | neighbors[9] | 8 | 11 | | | | |
| Dallas | neighbors[10] | 2 | 4 | 8 | 11 | | |
| Houston | neighbors[11] | 8 | 9 | 10 | | | |

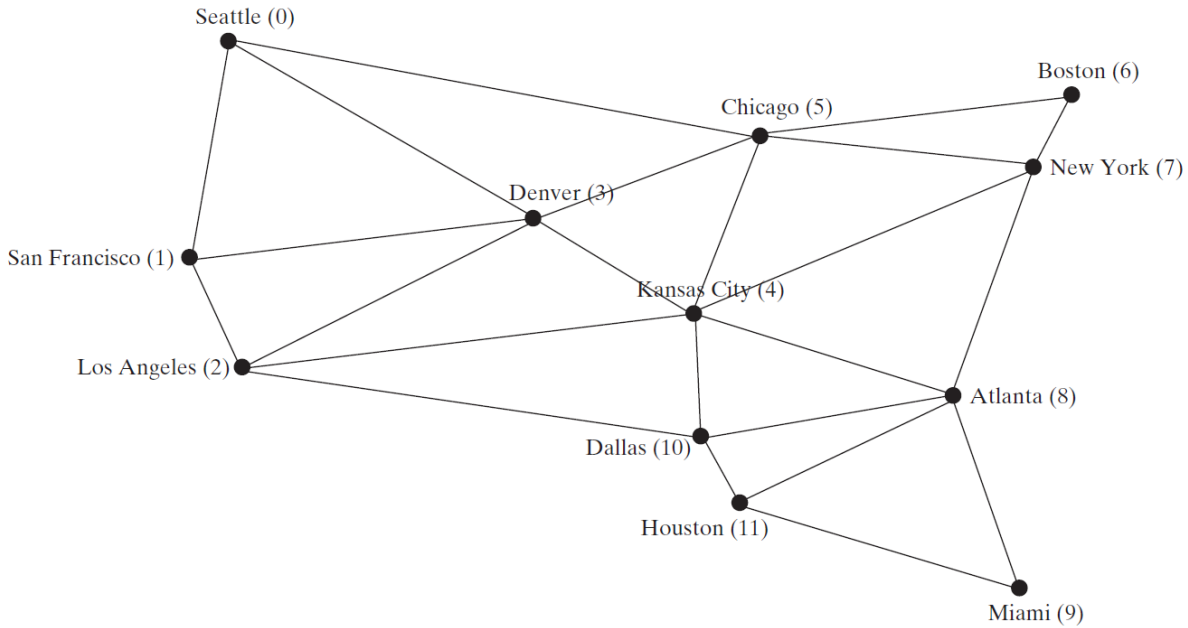



```
java.util.List<Edge>[] neighbors = new java.util.List[12];
```

Adjacency edge list

# Representing Graphs: Representing Adjacency Edge List Using ArrayList

```
List<ArrayList<Edge>> neighbors = new ArrayList<>();
neighbors.add(new ArrayList<Edge>());
neighbors.get(0).add(new Edge(0, 1));
neighbors.get(0).add(new Edge(0, 3));
neighbors.get(0).add(new Edge(0, 5));
neighbors.add(new ArrayList<Edge>());
neighbors.get(1).add(new Edge(1, 0));
neighbors.get(1).add(new Edge(1, 2));
neighbors.get(1).add(new Edge(1, 3));
...
...
neighbors.get(11).add(new Edge(11, 8));
neighbors.get(11).add(new Edge(11, 9));
neighbors.get(11).add(new Edge(11, 10));
```



# Traversals

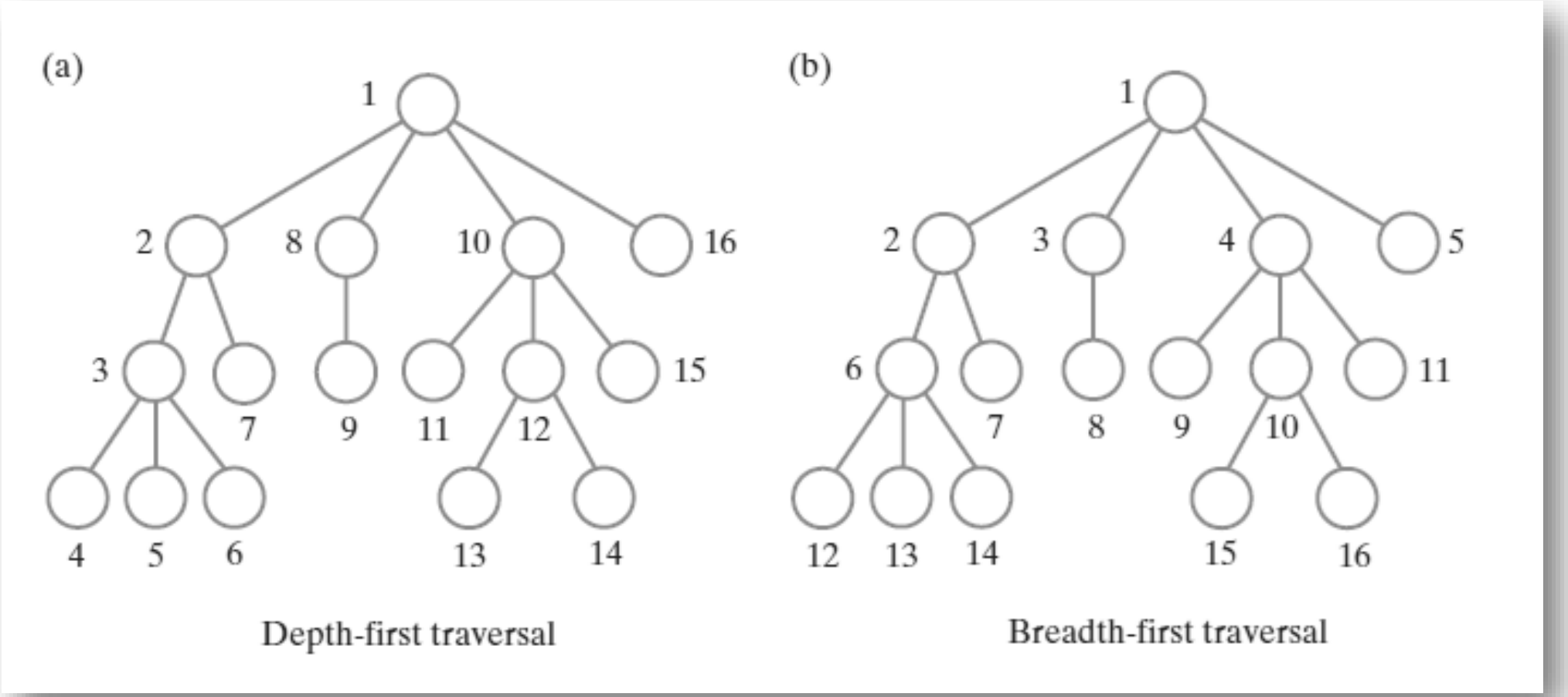



Image Source: F. M. Carrano & T. M. Henry, "Data Structures and Abstractions with Java," 4th ed. Pearson Education, Inc., 2015.

The visitation order of two traversals: (a) Depth-first traversal (preorder); (b) Breadth-first traversal (level-order).

# References


- Y. D. Liang, “Introduction to Java Programming and Data Structures,” Comprehensive version, 11th ed. Pearson Education, Inc., 2018.