# ZAMAN UNIVERSITY



Data Structures and Algorithms

Chapter 6

# Graphs

# Outline



- Terminology and Representations
- Graph Traversals
- Shortest-Paths Problems
- Minimum-Cost Spanning Trees

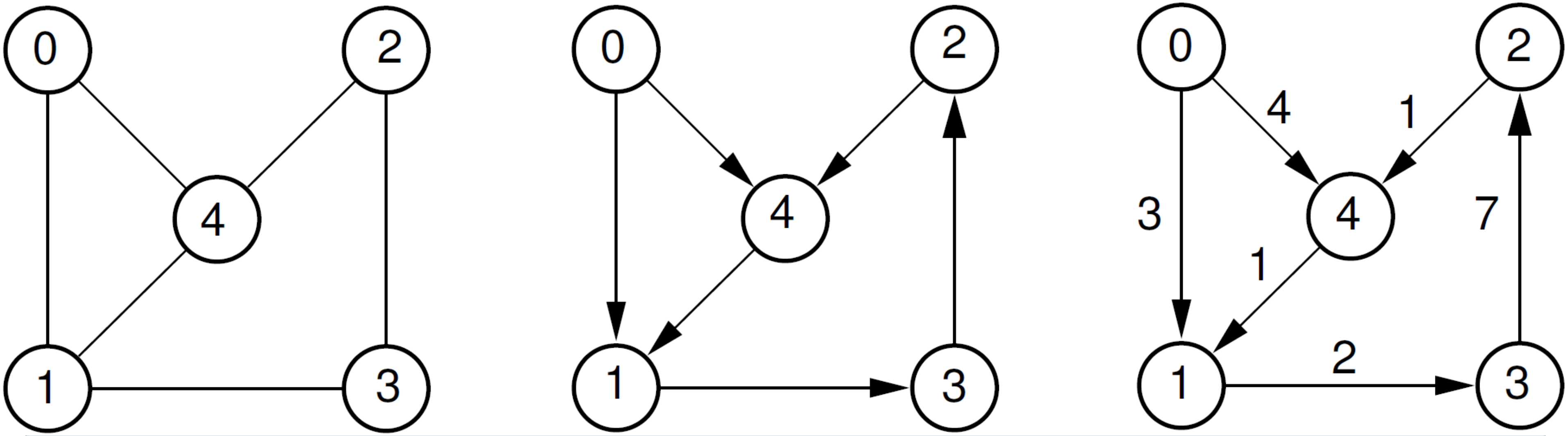
# What is Graph?



- Graphs provide the ultimate in data structure flexibility
- Graphs can model both real-world systems and abstract problems
- Here is a small sampling of the range of problems that graphs are applied to:
  - Modeling connectivity in computer and communications networks;
  - Representing a map as a set of locations with distances between locations; for computing shortest routes between locations;
  - Modeling flow capacities in transportation networks;
  - Finding a path from a starting condition to a goal condition; for example, in artificial intelligence problem solving;
  - Modeling computer algorithms, showing transitions from one program state to another;
  - And …

# Example of Graphs






*a. A Graph* *b. A Directed Graph (Digraph)* *c. A Weighted Graph*

# Outline



- Terminology and Representations
- Graph Traversals
- Shortest-Paths Problems
- Minimum-Cost Spanning Trees

# Terminology



- A graph G consist of
  1. Set of vertices *V* (called nodes), $V = \{v_1, v_2, v_3, \ldots,\}$ and
  2. Set of edges *E* (i.e., $E=\{e_1, e_2, e_3, \ldots\}$)
- Thus, a graph can be represents as $G = (V, E)$, where *V* is a finite and non empty set at vertices and *E* is a set of pairs of vertices called edges.
- Consider a graph G in *Figure c*, then the vertex V and edge E can be represented as: $V=\{0, 1, 2, 3, 4\}$ and $E=\{(0,1), (0,4), (1,3), (3,2), (2,4), (4,1)\}$.

# Representations



- There are two standard ways of maintaining a graph *G* in the memory of a computer:
  - Sequential representation of a graph using adjacent;
  - Linked representation of a graph using linked list.

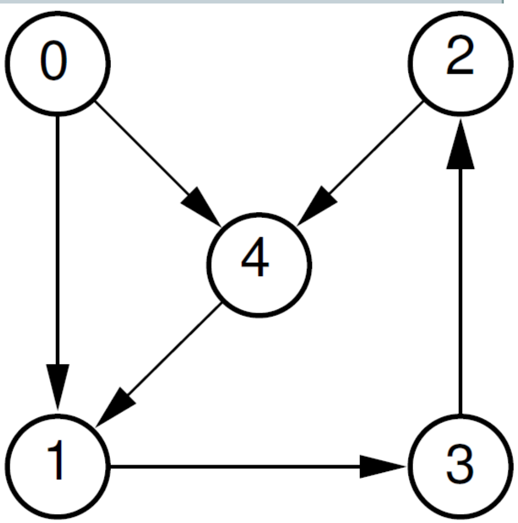
# Adjacency Matrix Representation



- Graph $G = (V, E)$ with $n$ vertices, is an $n \times n$ matrix.
- The adjacency matrix A of a directed graph $G$ can be represented with the following conditions:
  - $A_{ij} = 1$ {if there is an edge from $V_i$ to $V_j$ or if the edge $(i, j)$ is member of $E$.}
  - $A_{ij} = 0$ {if there is no edge from $V_i$ to $V_j$}

**A=**

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 |
| 3 | 0 | 0 | 1 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 |



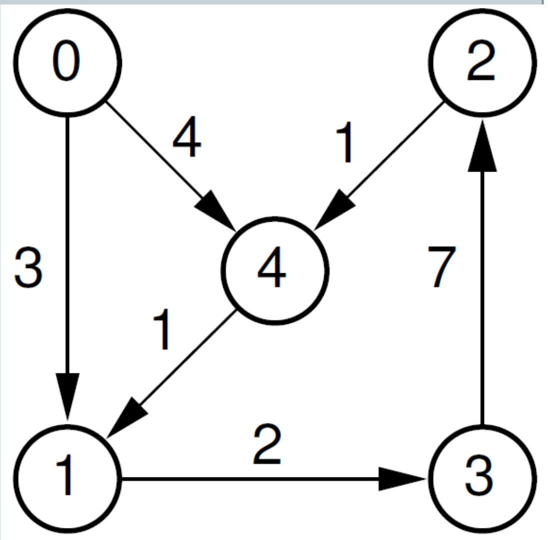
# Adjacency Matrix Representation for Directed Weighted Graph



- The adjacency matrix A for a directed weighted graph $G = (V, E, W_e)$ can be represented:
  - $A_{ij} = W_e$ {if there is an edge from $V_i$ to $V_j$ then represent its weight $W_{ij}$}
  - $A_{ij} = -1$ {if there is no edge from $V_i$ to $V_j$}

**A=**

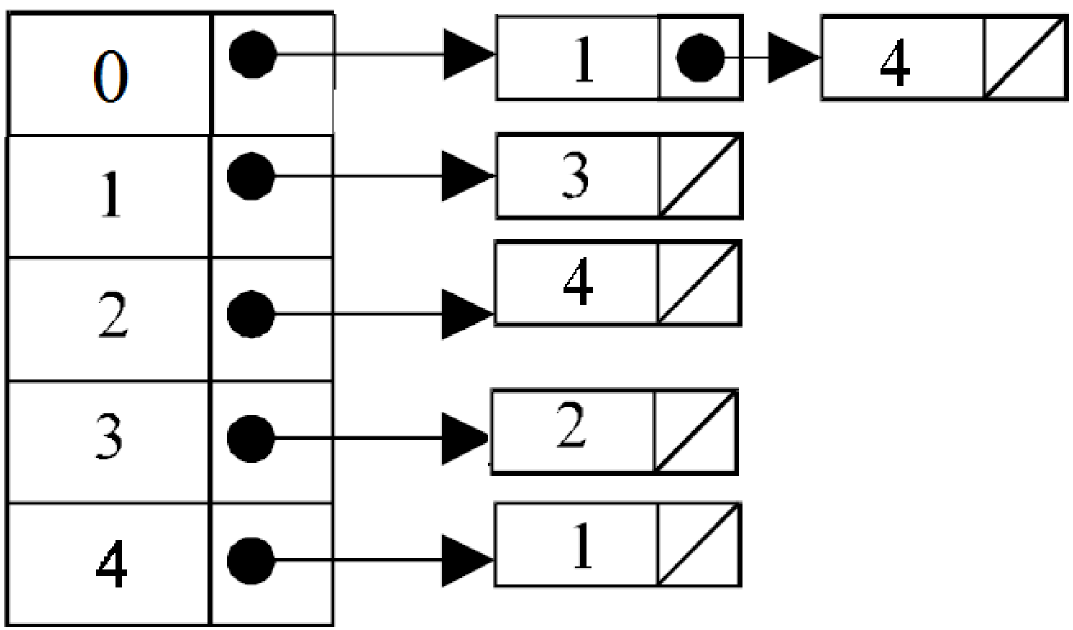
| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | -1 | 3 | -1 | -1 | 4 |
| 1 | -1 | -1 | -1 | 2 | -1 |
| 2 | -1 | -1 | -1 | -1 | 1 |
| 3 | -1 | -1 | 7 | -1 | -1 |
| 4 | -1 | 1 | -1 | -1 | -1 |

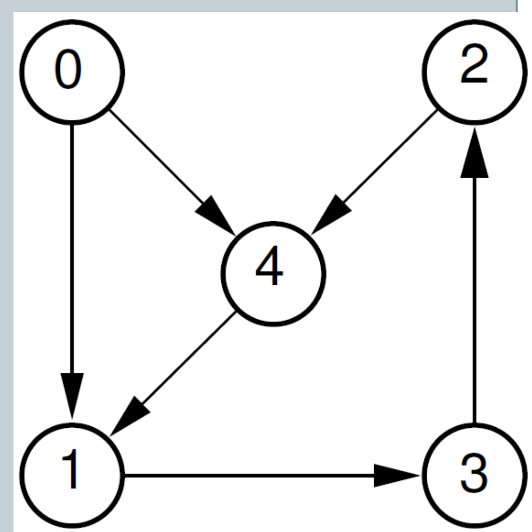


# Linked List Representation



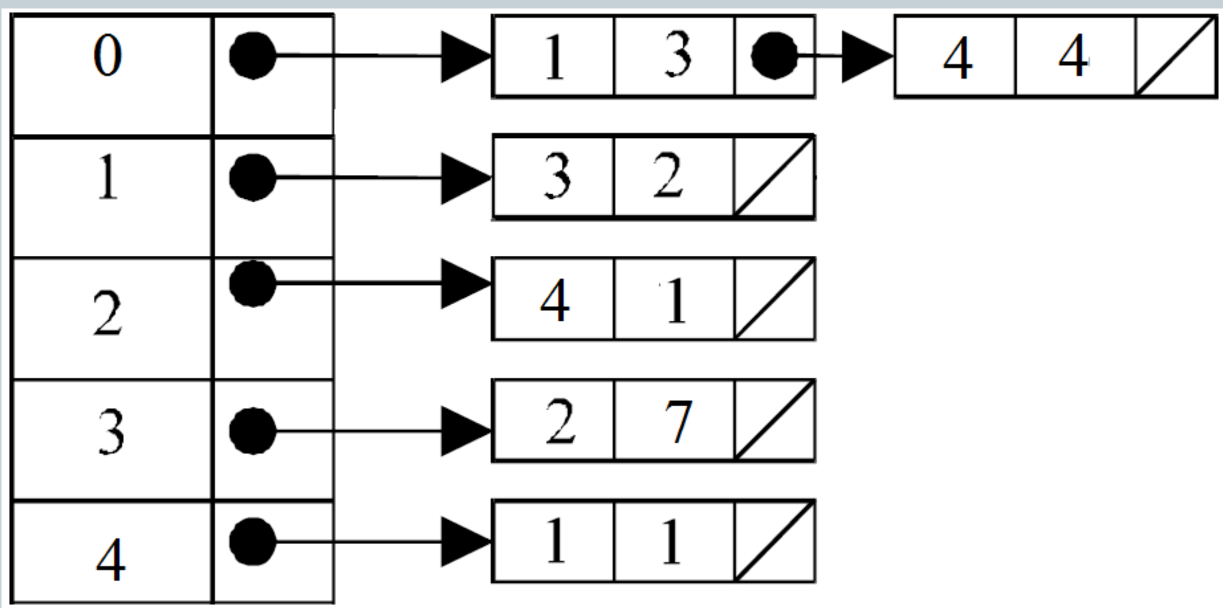
- In this representation (also called adjacency list representation), graph is stored as linked structure.

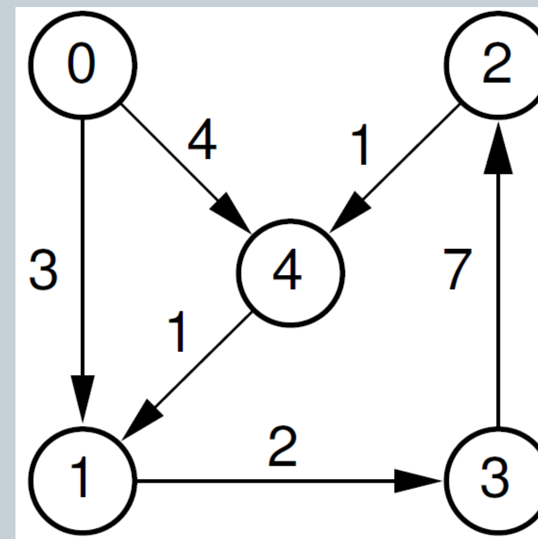


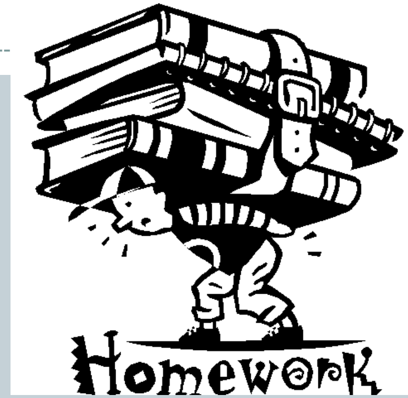



# Linked List Representation for Direct Weighted Graph








Write a program to represent graph as Adjacent Matrix and Linked List

# Outline



- Terminology and Representations
- Graph Traversals
- Shortest-Paths Problems
- Minimum-Cost Spanning Trees

# Graph Traversals



- As in tree, traversing a graph consists of visiting each vertex only one time
- Simple traversal algorithms used for trees CANNOT be applied in graph, since graphs may include cycles, thus the tree traversal algorithms would result infinite loops.
- To prevent this, for each visited vertex will be marked to avoid revisiting.

# Outline



- Terminology and Representations
- Graph Traversals
  - Depth First Search
  - Breadth First Search
- Shortest-Paths Problems
- Minimum-Cost Spanning Trees

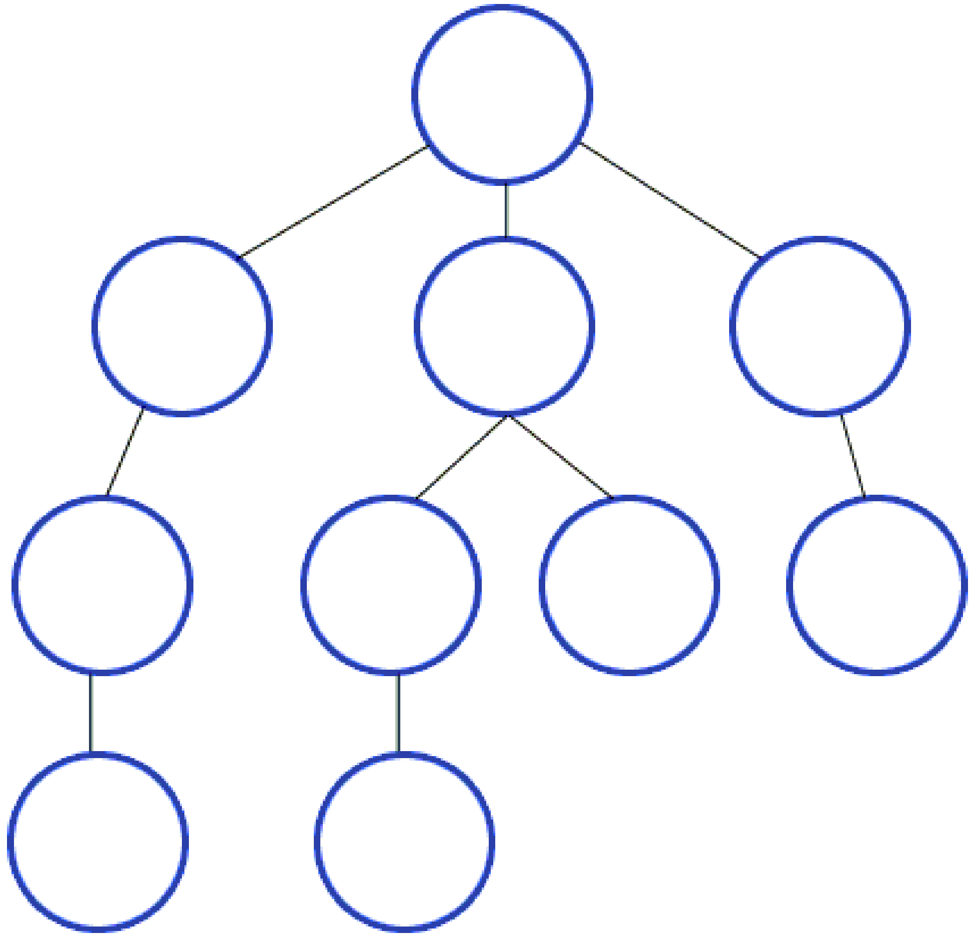
# Graph Traversals: Depth First Search (s, G) - Iterative



1. Initialize: Set Stack = { s }, VisitedList = { }
2. Terminate if Stack is empty
3. Select a vertex, *n*, from Stack
4. Visit *n* and save *n* to VisitedList
5. Expand: Define Successors *m* of vertex *n* *in* G. For each successor, *m*, insert *m* in Stack only if $m \notin$ [ Stack U VisitedList ]
6. Loop: Go to step 2

# Animation of Depth First Search





https://commons.wikimedia.org

# Outline



- Terminology and Representations
- Graph Traversals
  - Depth First Search
  - Breadth First Search
- Shortest-Paths Problems
- Minimum-Cost Spanning Trees

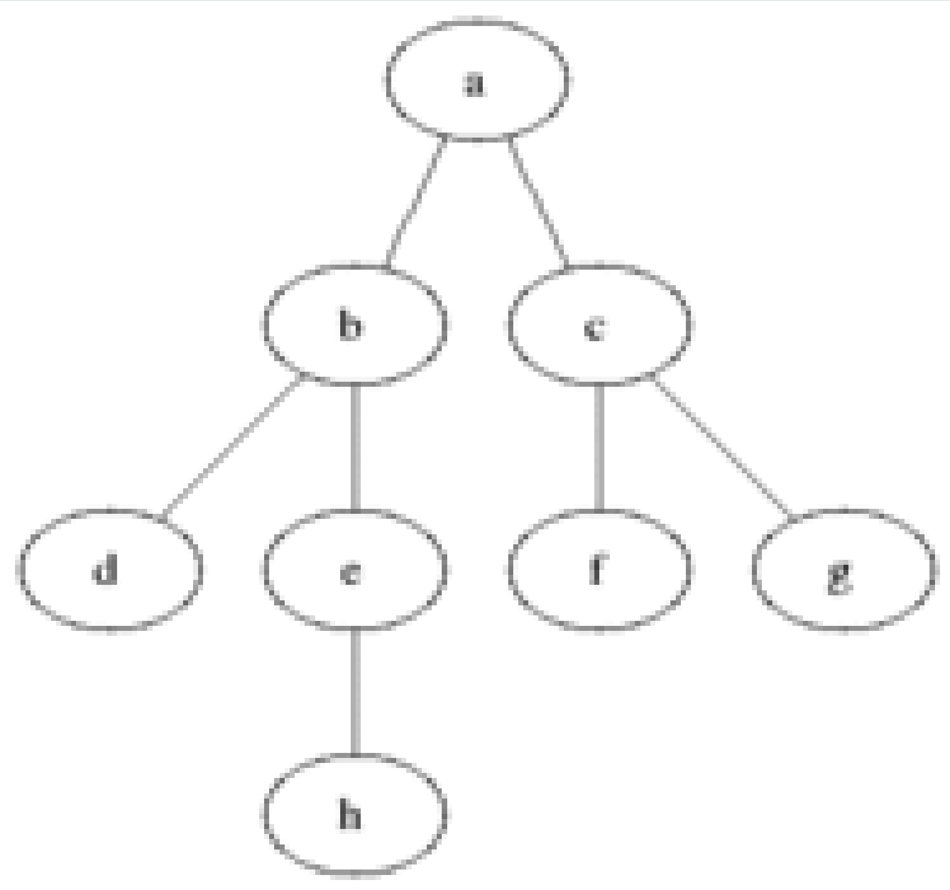
# Graph Traversals: Breadth First Search (s, G)

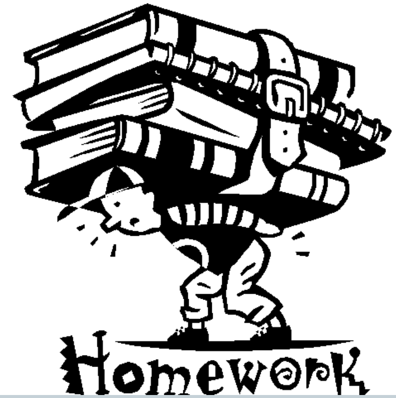


1. Initialize: Set Queue = { s }, VisitedList = { }
2. Terminate if Queue is empty
3. Select a vertex, *n*, from Queue
4. Visit *n* and save *n* to VisitedList
5. Expand: Define Successors *m* of vertex *n* in G. For each successor, *m*, insert *m* in Queue only if $m \notin$ [Queue U VisitedList ]
6. Loop: Go to step 2

# Animation of Breadth First Search








Write functions of **Depth First Search** and **Breadth First Search** based on pseudo-code in the slide.

To be continued…