Graph Basics

- Terminologies
- Representations
- Traversals

Data Structures

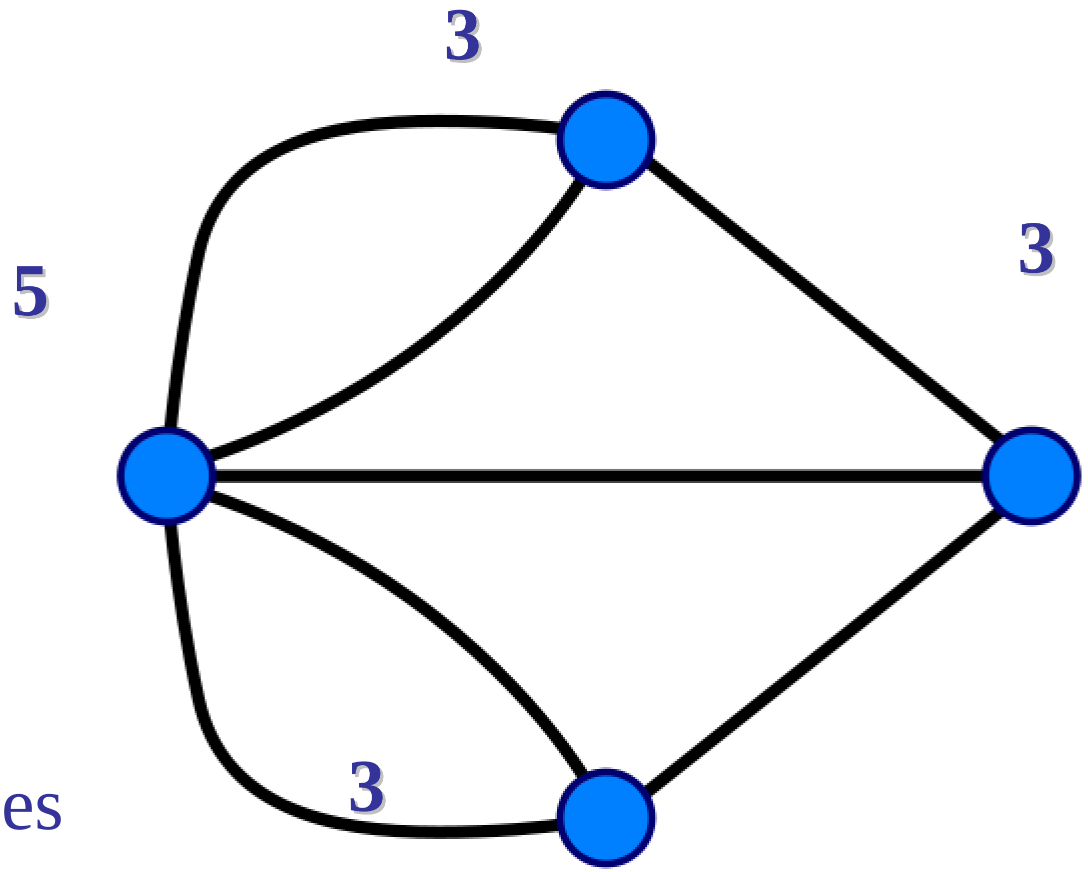
Seven Bridges of Königsberg

- **Königsberg in Prussia**
  - Kaliningrad, Russia
  - Pregel River
  - Two large islands
  - Seven bridges
- **Leonhard Euler [1735]**
  - Find a walk through the city that would cross each bridge once and exactly once
  - The first paper in the history of graph theory

Wikipedia ©

Seven Bridges of Königsberg

- G = {V, E}
  - V(G): vertex set
  - E(G): edge set
  - degree
    - Number of edges



- Vertex types
  - Odd or even degrees

$$\sum_{v_i \in V(G)} \text{degree}(v_i) = |E(G)| \times 2$$

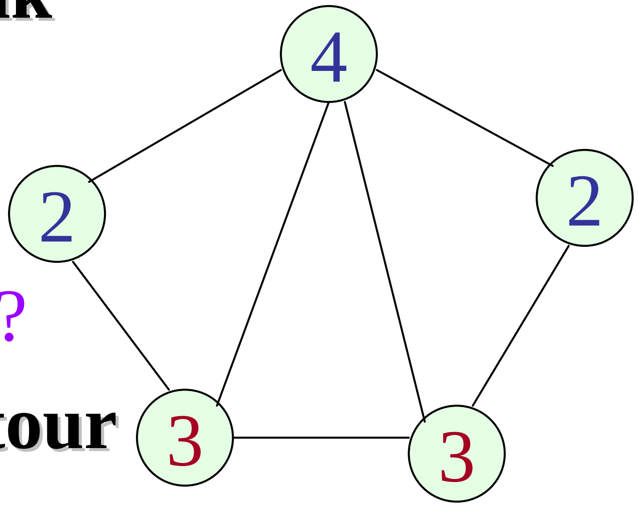
Wikipedia ©

Seven Bridges of Königsberg

Data Structures

- **Eulerian path (trial) / Euler walk**
  - visits every edge exactly once
  - 0 or 2 nodes with odd degrees

  What if 1 node with odd degrees?



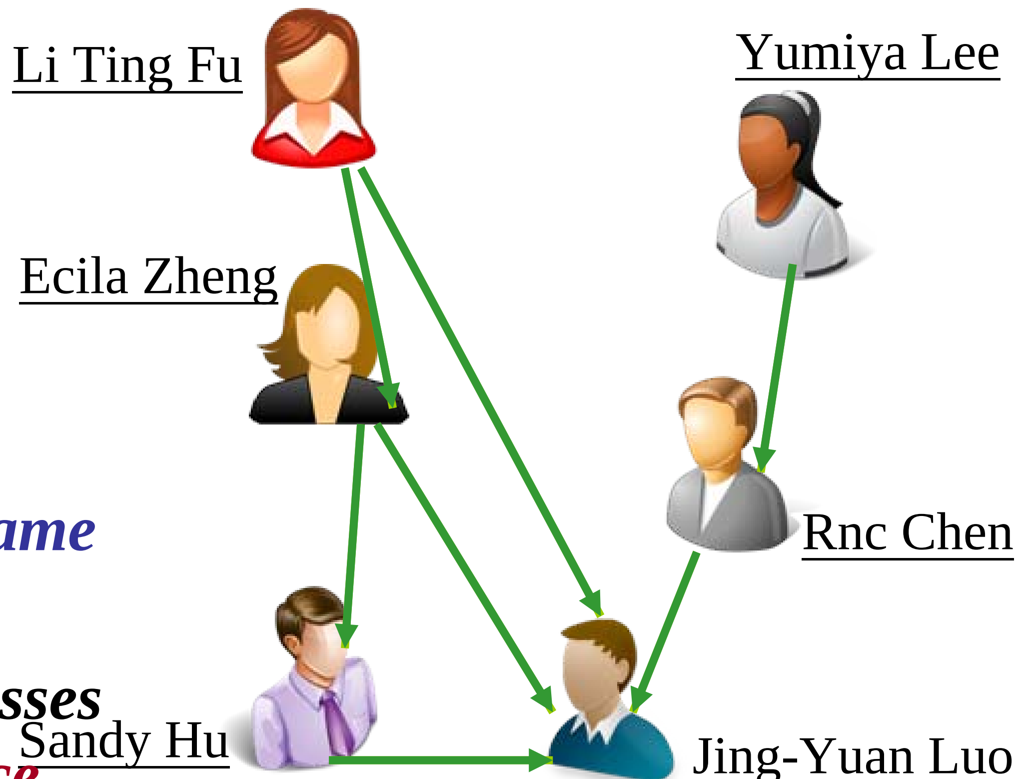
- **Eulerian circuit (cycle) / Euler tour**
  - begin and end at the same vertex
  - 0 node with odd degrees
- **Seven bridge problem**
  - Q&A?

$$\sum_{v_i \in V(G)} \text{degree}(v_i) = |E(G)| \times 2$$

Wikipedia ©

Basic Terminologies

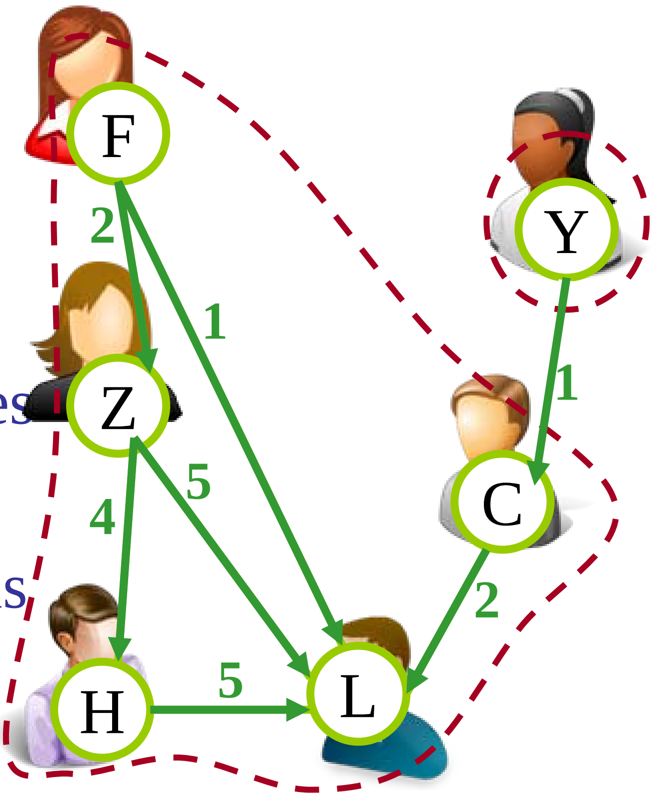
- **Undirected graph**
- **Directed graph (digraph)**
- **Adjacent vertices**
- **Edge is *incident* to vertices**
- **Path:** ***a sequence of edges***
- **Cycle:** ***begin & end at the same vertex***
- **Simple path:** ***a path that passes through any vertex only once***
- **Simple cycle:** ***a cycle that passes through the other vertices only once***



6 degrees of separation from the small world experiment

Basic Terminologies

- **Connected graph**
  - There is a path between any two vertices
  - Disconnected graph
- **Complete graph |E|=?**
  - There is an edge between any two vertices
- **Strong connected graph**
  - For any two vertices on a digraph, there is a path from one vertex to the other
  - Q&A: how to decide it?
- **Weighted graph**
  - the edges have numeric labels



connected components

Graphs As ADTs

- **Variations of an ADT *graph* are possible**
  – Vertices may or may not contain values
    - **Many problems have no need for vertex values**
    - **Relationships among vertices is what is important**
  – Either *directed* or *undirected* edges
  – Either *weighted* or *unweighted* edges
- **Insertion and deletion operations for graphs apply to vertices and edges**
- **Graphs can have traversal operations**

ADT *graph* Operations

Data Structures

```
int numVertices;   /** Number of vertices in the graph. */
int numEdges;      /** Number of edges in the graph. */
int getNumVertices();
int getNumEdges();
int getWeight(Edge e);
void add(Edge e);
void remove(Edge e);
bool isEdge(Vertex u, Vertex v);
int getDegree(Vertex v);
bool isConnected(Graph g);
edgeList traverse(Graph g);
```

Graph Representations

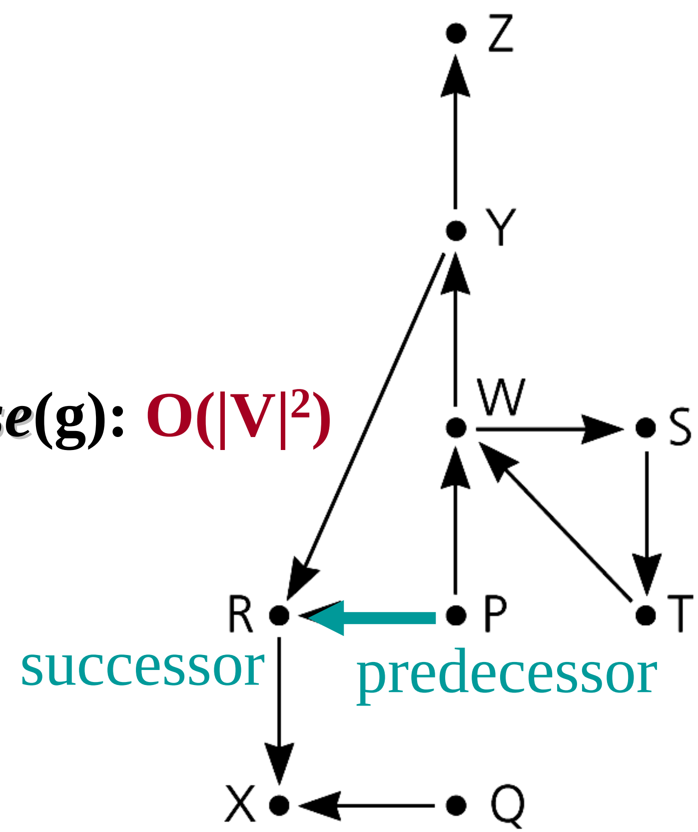
- **Most common implementations of a graph**
  1. Adjacency matrix
  2. Adjacency list
- **Adjacency matrix for a graph that has *n* vertices numbered 0, 1, …, *n* – 1**
  – An *n* by *n* array matrix such that matrix[i][j] indicates whether an edge exists from vertex *i* to vertex *j*

Adjacency Matrix: *Examples*

In-degree vs. Out-degree

	P	Q	R	S	T	W	X	Y	Z
P	0	0	1	0	0	1	0	0	0
Q	0	0	0	0	0	0	1	0	0
R	0	0	0	0	0	0	1	0	0
S	0	0	0	0	1	0	0	0	0
T	0	0	0	0	0	1	0	0	0
W	0	0	0	1	0	0	0	1	0
X	0	0	0	0	0	0	0	0	0
Y	0	0	1	0	0	0	0	0	1
Z	0	0	0	0	0	0	0	0	0

***traverse*(g): O(|V|^2)**



Adjacency Matrix: *Examples*

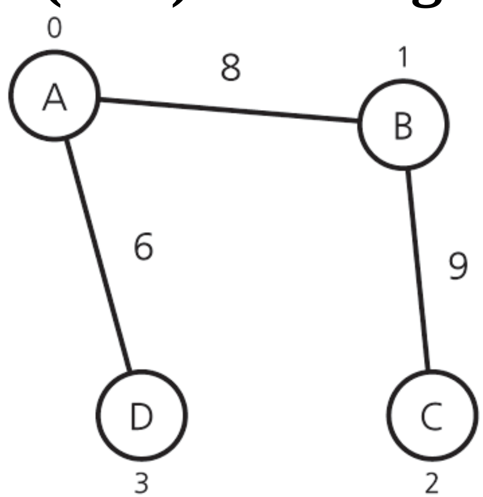
– For an unweighted graph, matrix[i][j] is

- **1 (or true) if an edge exists from vertex *i* to vertex *j***
- **0 (or false) if no edge exists from vertex *i* to vertex *j***

– For a weighted graph, matrix[i][j] is

- **The weight of the edge from vertex *i* to vertex *j***
- **∞ (or 0) if no edge exists from vertex *i* to vertex *j***

(a)



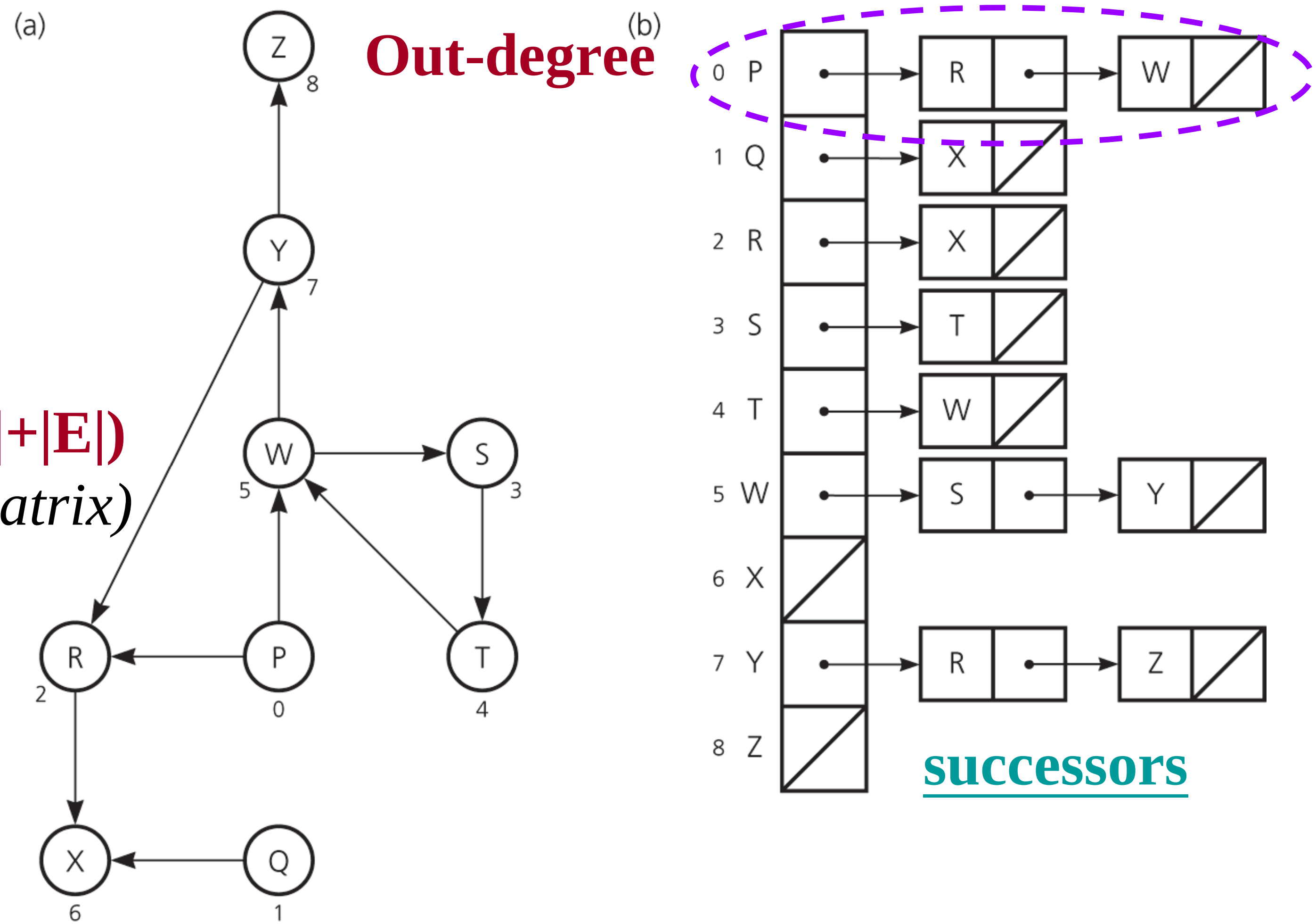
(b)

		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Adjacency List

- **Adjacency list for a directed graph that has *n* vertices numbered 0, 1, …, *n* – 1**
  - An array of *n* linked lists
  - The *i*th linked list has a node for vertex *j* if and only if an edge exists from vertex *i* to vertex *j*
  - The list's node can contain either
    - **Vertex *j*'s value, if any**
    - **An *indication* of vertex *j*'s identity**

Adjacency List: *Examples*



traverse(g): O(|V|+|E|)

(if it is a sparse matrix)

Graph Representations

- **Two common operations on graphs**

  isEdge(i,j)

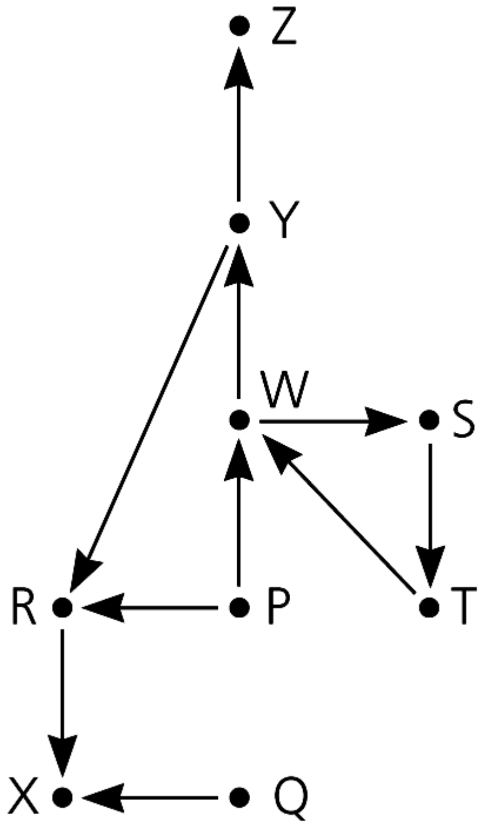
  1. Determine whether there is an edge from vertex *i* to vertex *j*
  2. Find all vertices adjacent to a given vertex *i*

  getDegree(i)

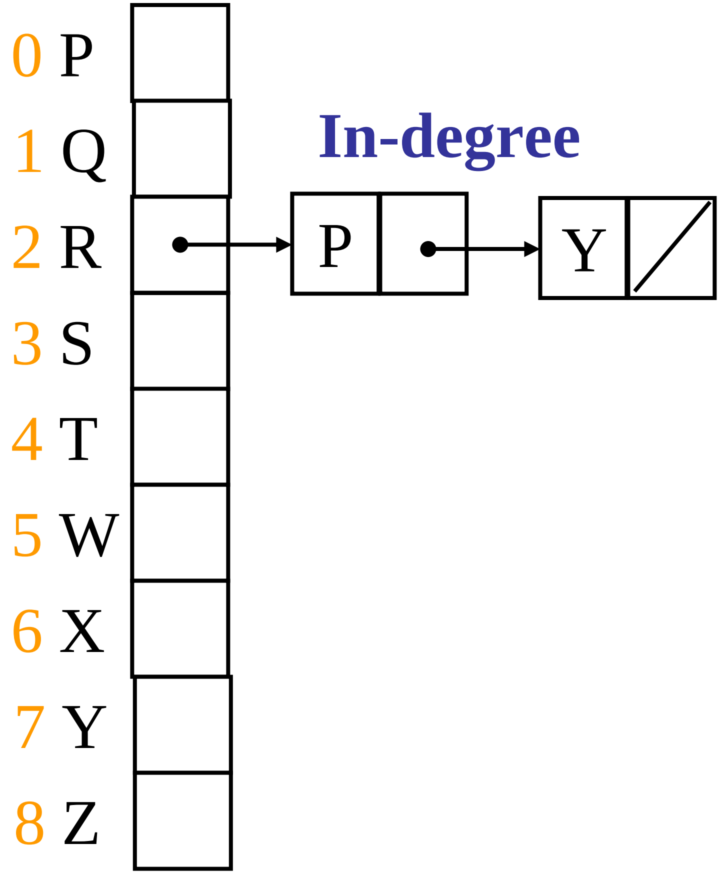
- **Adjacency matrix**
  - Supports operation 1 more efficiently
- **Adjacency list**
  - Supports operation 2 more efficiently
  - Often requires less space than an adjacency matrix

Practice 1: *Adjacency List*

- **Draw the *Inverse* Adjacency List**



In-degree



<u>**predecessors**</u>

Other Graph Representations

- **Mapping from vertex labels to array indices**

  P Q R S T W X Y Z

  0 1 2 3 4 5 6 7 8

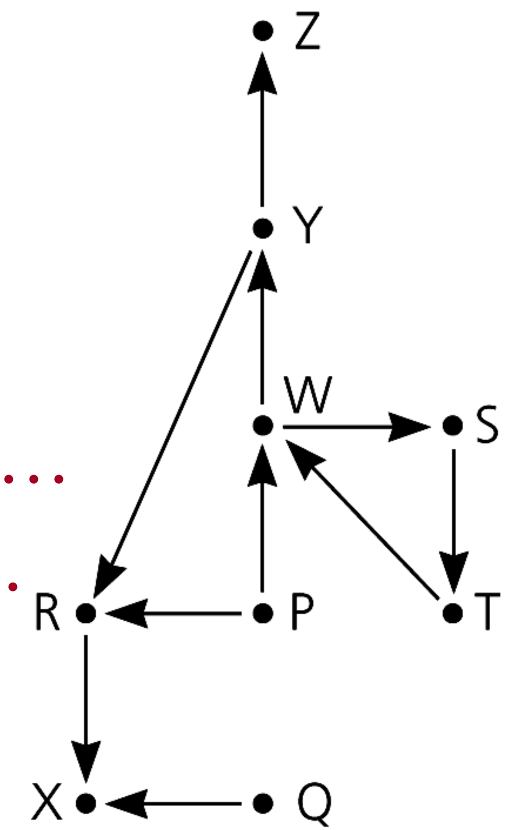
- **Sequential representation**

  – nodes + edges

  [0][1][2][3]…[7][8][9][10][11][12][13]…

  10 12 1314…19 20 20 2 5 6 6 …

  Q➔X



undirected graph: |V|+2|E|+1

Other Graph Representations

- **Mapping from vertex labels to array indices**

  A B C D

  0 1 2 3

Q&A: [loc] ➔ (u,v)?

- **Lower triangular matrix for *undirected graph***

  [0] ⇔(1,0): 1

  [1] ⇔(2,0): 1

  [2] ⇔(2,1): 1

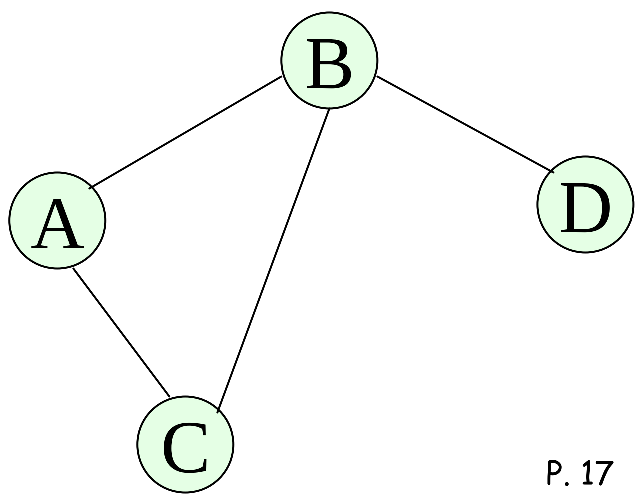
  [3] ⇔(3,0): 0

  [4] ⇔(3,1): 1

  [5] ⇔(3,2): 0

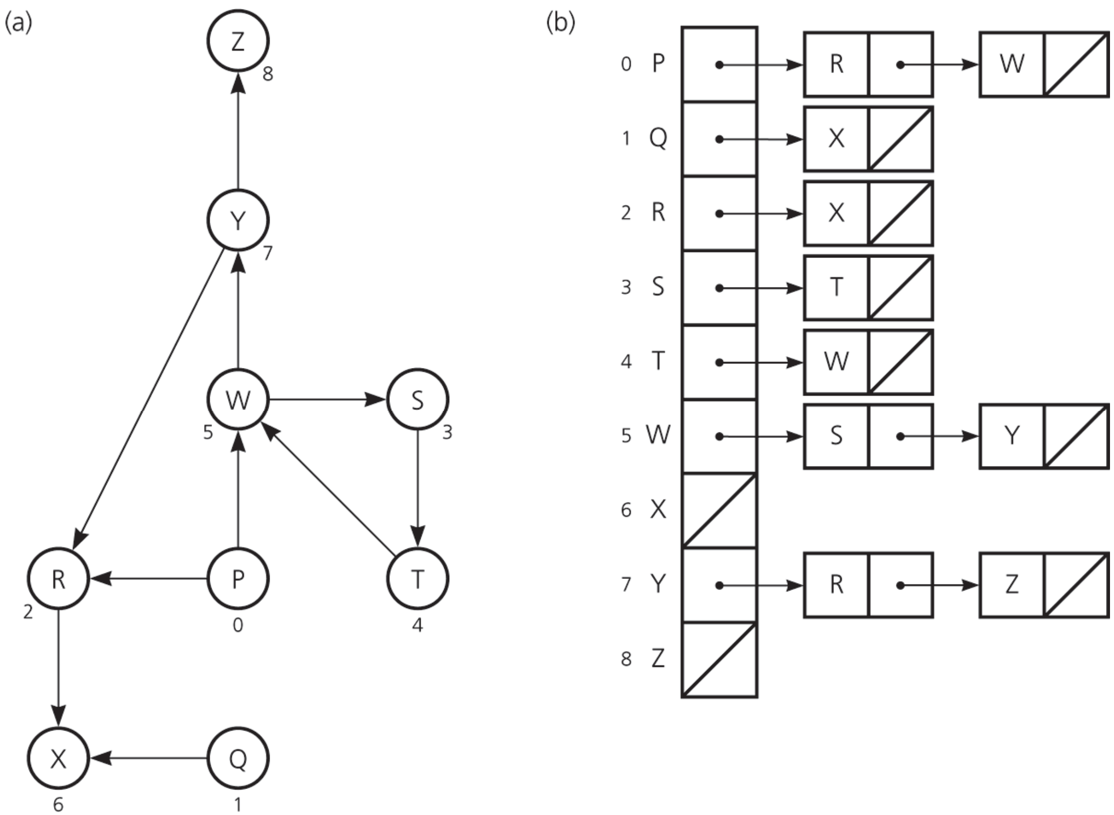
(u,v)* ➔ *loc = u*(u-1)/2 + v

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0



Self-exercise 1

1. Assume that each character, integer or address needs M bytes. Estimate the sizes of the **adjacency matrix** and the **adjacency list** to decide which requires less memory.



	P	Q	R	S	T	W	X	Y	Z
P	0	0	1	0	0	1	0	0	0
Q	0	0	0	0	0	0	1	0	0
R	0	0	0	0	0	0	1	0	0
S	0	0	0	0	1	0	0	0	0
T	0	0	0	0	0	1	0	0	0
W	0	0	0	1	0	0	0	1	0
X	0	0	0	0	0	0	0	0	0
Y	0	0	1	0	0	0	0	0	1
Z	0	0	0	0	0	0	0	0	0

Self-exercise 1

2. Draw the **sequential representation** (a single array) of the following graph.

[0]	[1]	[2]	…
?	?	?	…