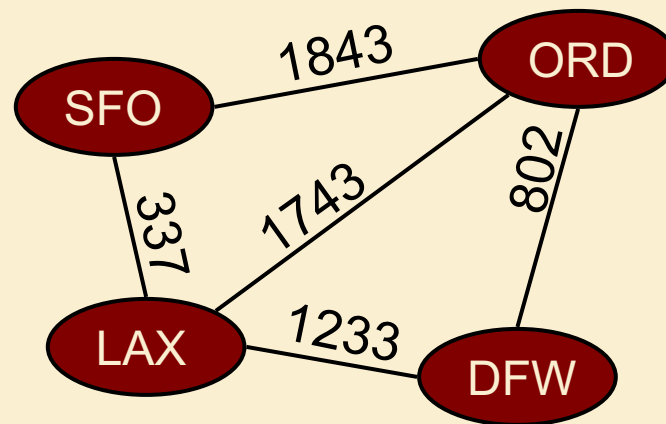
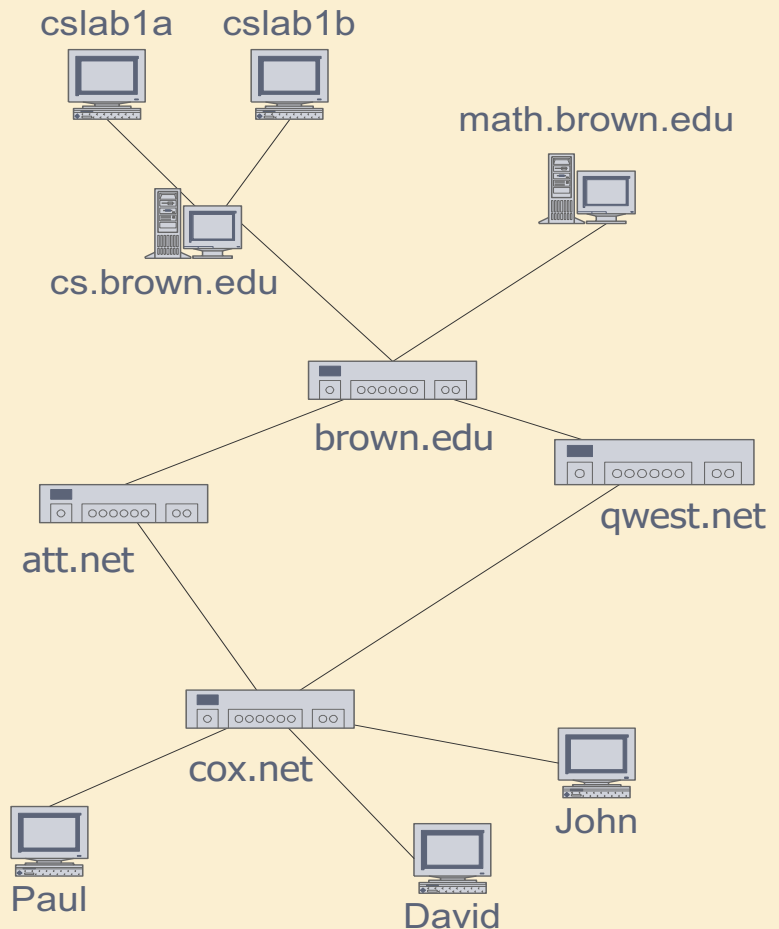


Graphs – ADTs and Implementations



Applications of Graphs

- Electronic circuits
 - ❑ Printed circuit board
 - ❑ Integrated circuit
- Transportation networks
 - ❑ Highway network
 - ❑ Flight network
- Computer networks
 - ❑ Local area network
 - ❑ Internet
 - ❑ Web
- Databases
 - ❑ Entity-relationship diagram



Outcomes

- By understanding this lecture, you should be able to:
 - ❑ Define basic terminology of graphs.
 - ❑ Use a graph ADT for appropriate applications.
 - ❑ Program standard implementations of the graph ADT.
 - ❑ Understand advantages and disadvantages of these implementations, in terms of space and run time.

Outline

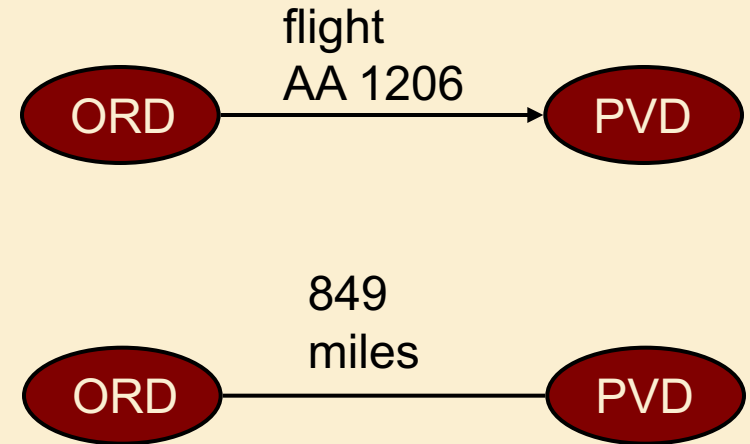
- Definitions
- Graph ADT
- Implementations

Outline

- **Definitions**
- Graph ADT
- Implementations

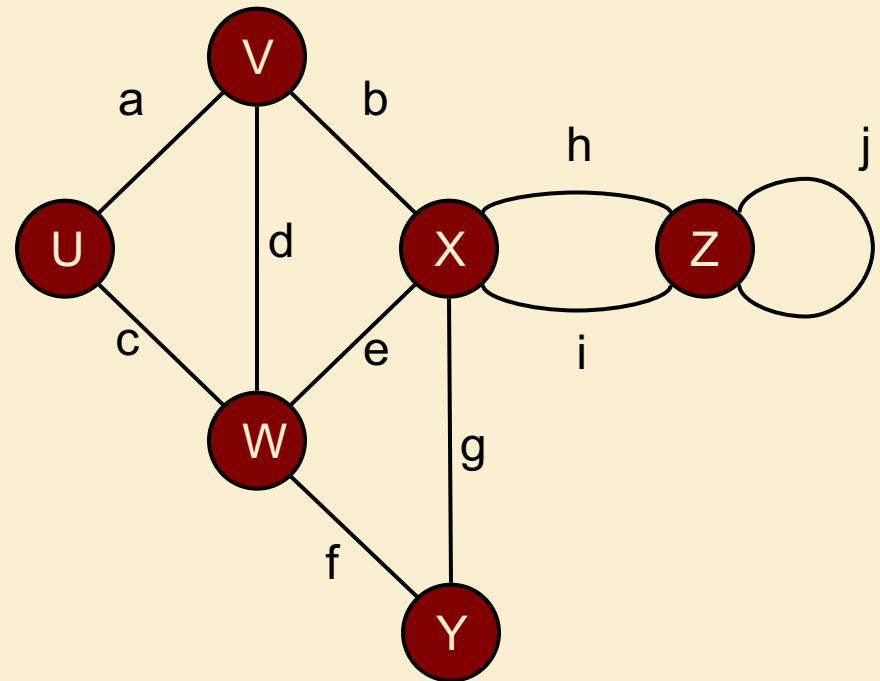
Edge Types

- Directed edge
 - ❑ ordered pair of vertices (u,v)
 - ❑ first vertex u is the origin
 - ❑ second vertex v is the destination
 - ❑ e.g., a flight
- Undirected edge
 - ❑ unordered pair of vertices (u,v)
 - ❑ e.g., a flight route
- Directed graph (Digraph)
 - ❑ all the edges are directed
 - ❑ e.g., route network
- Undirected graph
 - ❑ all the edges are undirected
 - ❑ e.g., flight network



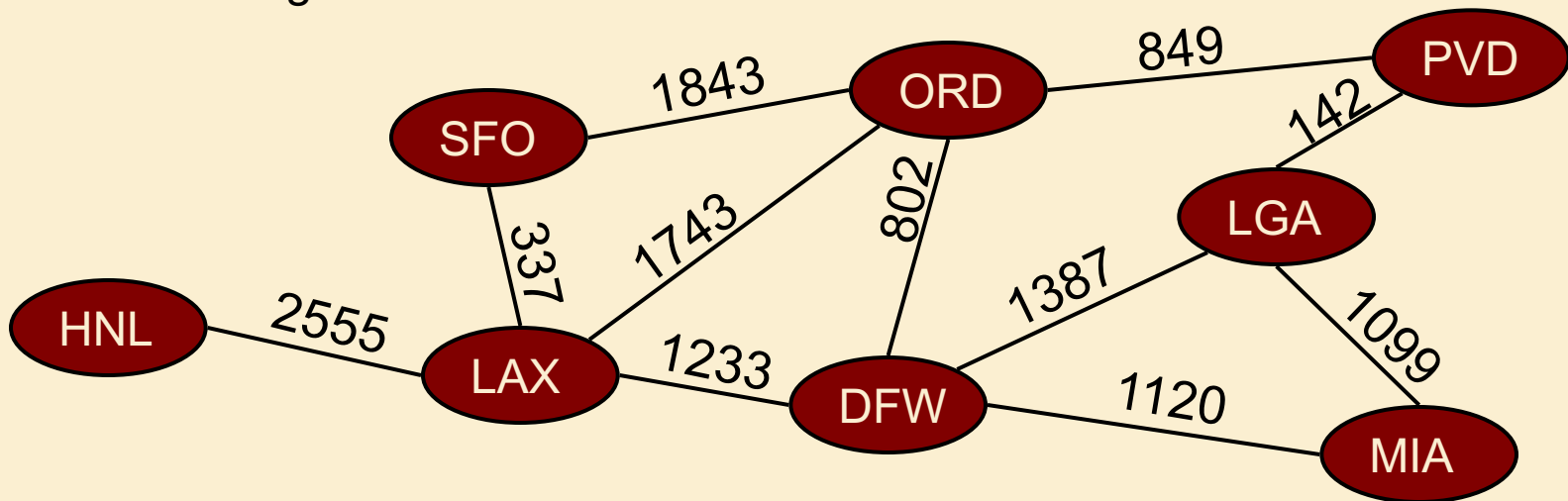
Vertices and Edges

- End vertices (or endpoints) of an edge
 - ❑ U and V are the endpoints of a
- Edges incident on a vertex
 - ❑ a, d, and b are incident on V
- Adjacent vertices
 - ❑ U and V are adjacent
- Degree of a vertex
 - ❑ X has degree 5
- Parallel edges
 - ❑ h and i are parallel edges
- Self-loop
 - ❑ j is a self-loop



Graphs

- A graph is a pair (V, E) , where
 - ❑ V is a set of nodes, called vertices
 - ❑ E is a collection of pairs of vertices, called edges
 - ❑ Vertices and edges are positions and store elements
- Example:
 - ❑ A vertex represents an airport and stores the three-letter airport code
 - ❑ An edge represents a flight route between two airports and stores the mileage of the route



Paths



Path

- sequence of alternating vertices and edges
- begins with a vertex
- ends with a vertex
- each edge is preceded and followed by its endpoints



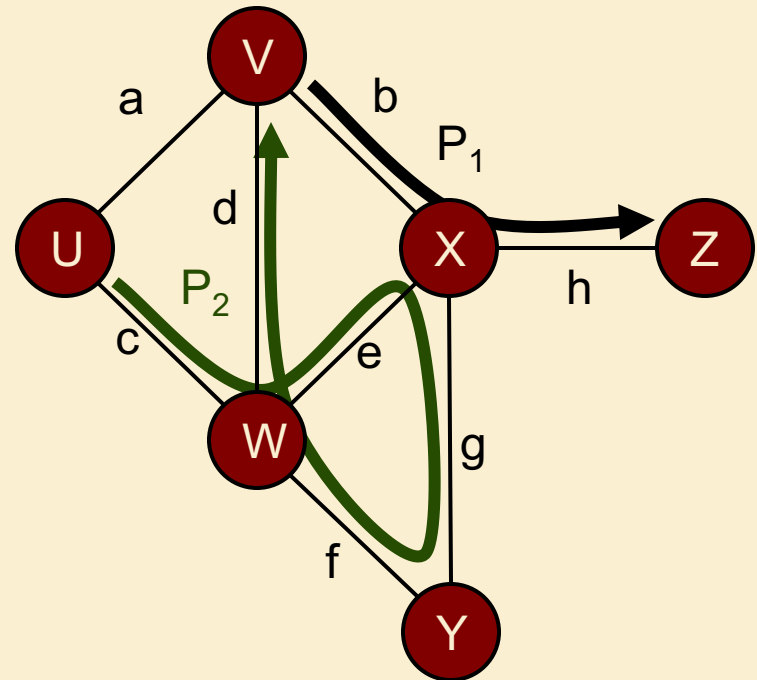
Simple path

- path such that all its vertices and edges are distinct



Examples

- $P_1 = (V, b, X, h, Z)$ is a simple path
- $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$ is a path that is not simple



Cycles

➤ Cycle

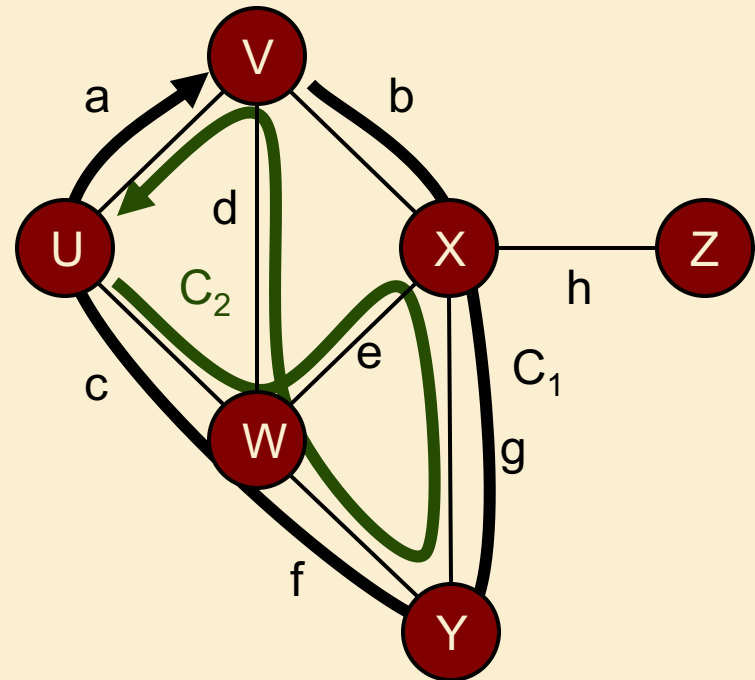
- ❑ circular sequence of alternating vertices and edges
- ❑ each edge is preceded and followed by its endpoints

➤ Simple cycle

- ❑ cycle such that all its vertices and edges are distinct (except for its first and last vertex)

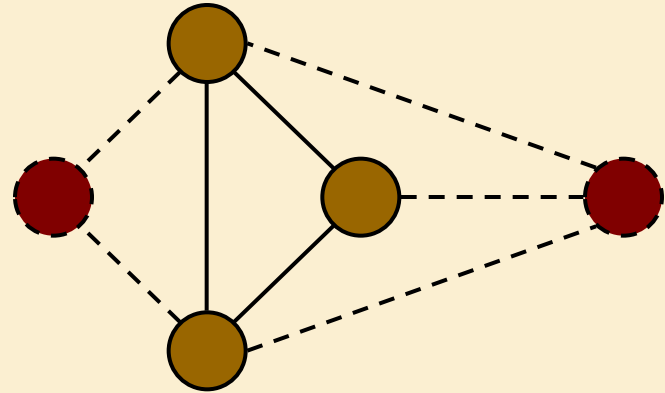
➤ Examples

- ❑ $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ is a simple cycle
- ❑ $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ is a cycle that is not simple

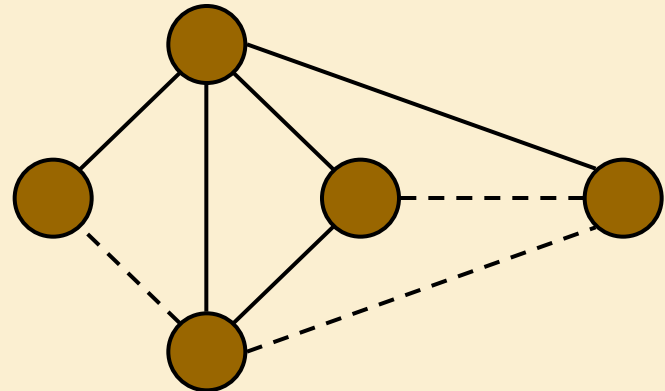


Subgraphs

- A subgraph S of a graph G is a graph such that
 - ❑ The vertices of S are a subset of the vertices of G
 - ❑ The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



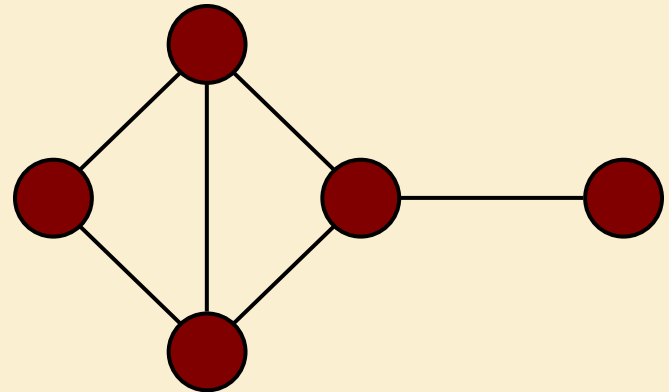
Subgraph



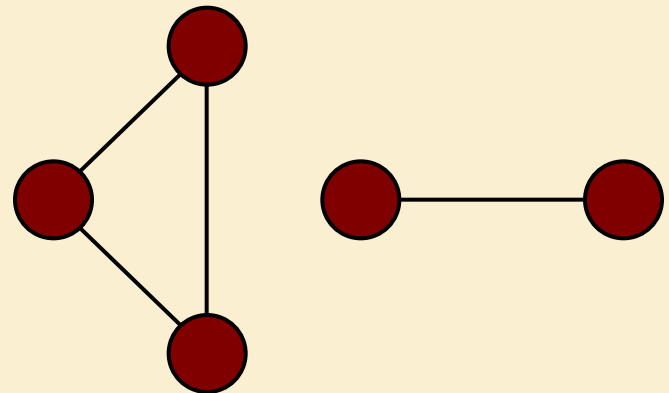
Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G

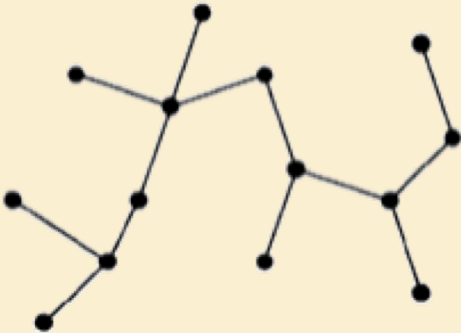


Connected graph

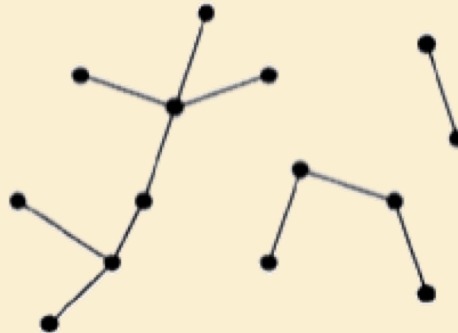


Non connected graph with two connected components

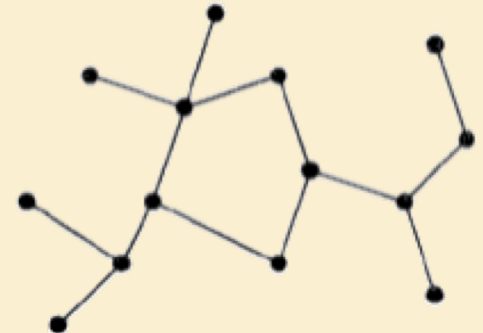
Trees



Tree



Forest



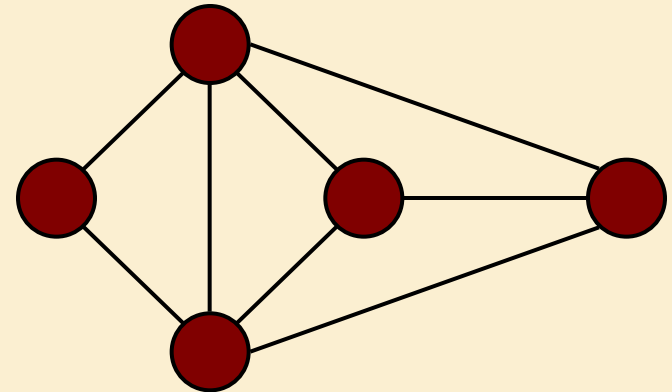
Graph with Cycle

A tree is a **connected**, **acyclic**, **undirected** graph.

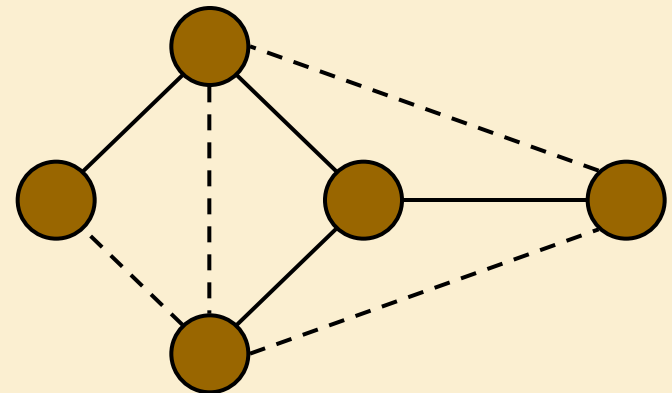
A forest is a **set** of trees (not necessarily connected)

Spanning Trees

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



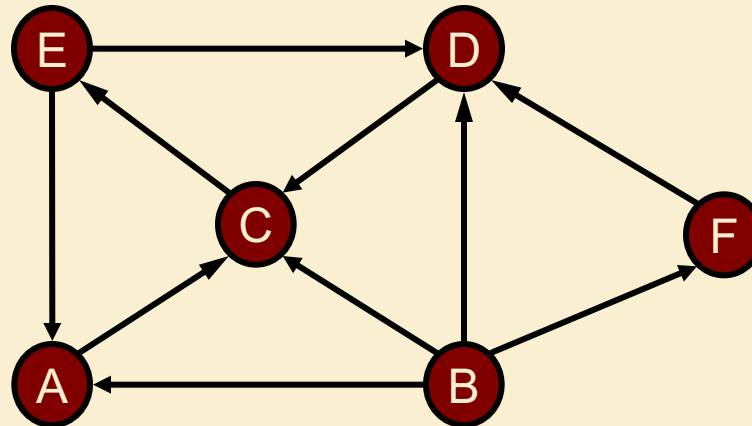
Spanning tree

Reachability in Directed Graphs

➤ A node w is **reachable** from v if there is a directed path originating at v and terminating at w .

❑ E is reachable from B

❑ B is not reachable from E



Properties

Property 1

$$\sum_v \deg(v) = 2|E|$$

Proof: each edge is counted twice

Notation

$|V|$ number of vertices

$|E|$ number of edges

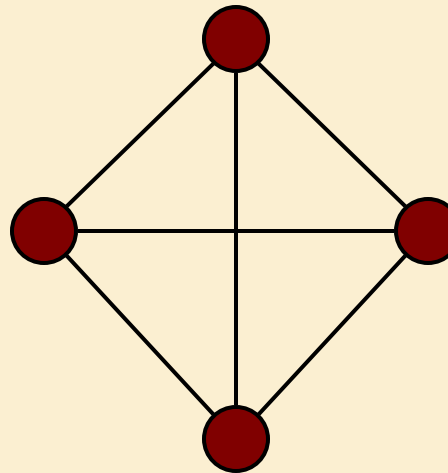
$\deg(v)$ degree of vertex v

Property 2

In an undirected graph with no self-loops and no parallel edges

$$|E| \leq |V|(|V| - 1)/2$$

Proof: each vertex has degree at most $(|V| - 1)$



Example

- $|V| = 4$
- $|E| = 6$
- $\deg(v) = 3$

Q: What is the bound for a digraph?

A: $|E| \leq |V|(|V| - 1)$

Outline

- Definitions
- **Graph ADT**
- Implementations

Main Methods of the Graph ADT

➤ Accessor methods

- ❑ **numVertices()**: Returns the number of vertices in the graph
- ❑ **numEdges()**: Returns the number of vertices in the graph
- ❑ **getEdge(u, v)**: Returns edge from u to v
- ❑ **endVertices(e)**: an array of the two endvertices of e
- ❑ **opposite(v, e)**: the vertex opposite to v on e
- ❑ **outDegree(v)**: Returns number of outgoing edges
- ❑ **inDegree(v)**: Returns number of incoming edges

Main Methods of the Graph ADT

➤ Update methods

- ❑ **insertVertex(x)**: insert a vertex storing element x
- ❑ **insertEdge(u, v, x)**: insert an edge (u,v) storing element x
- ❑ **removeVertex(v)**: remove vertex v (and its incident edges)
- ❑ **removeEdge(e)**: remove edge e

Main Methods of the Graph ADT

➤ Iterator methods

- ❑ `incomingEdges(v)`: Incoming edges to v
- ❑ `outgoingEdges(v)`: Outgoing edges from v
- ❑ `vertices()`: all vertices in the graph
- ❑ `edges()`: all edges in the graph

Outline

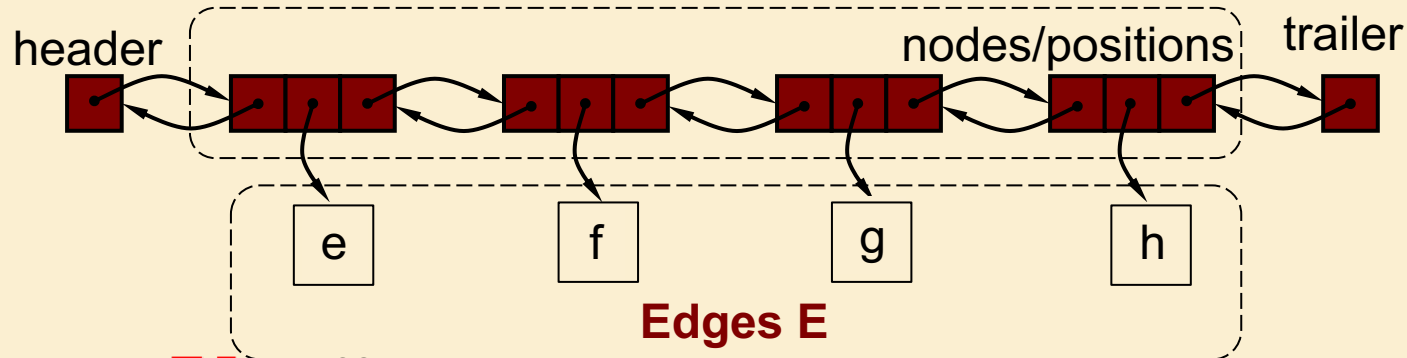
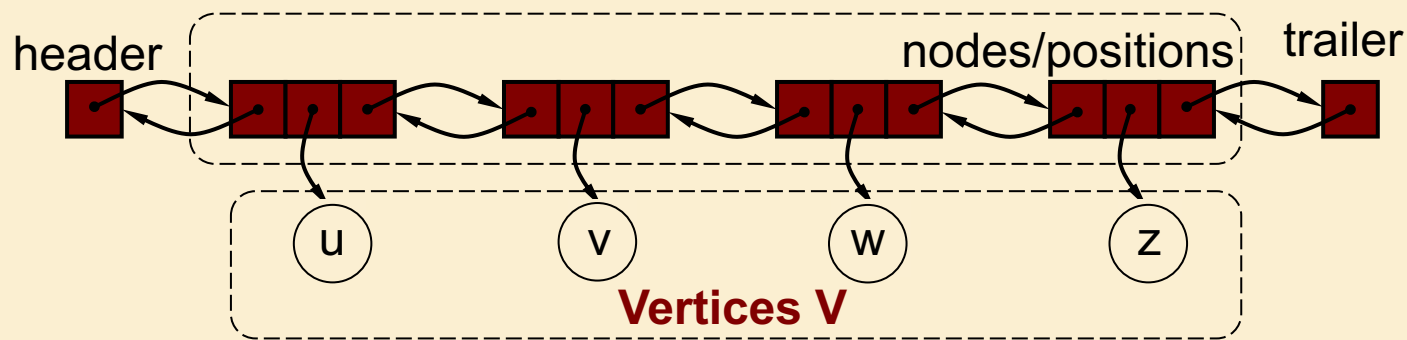
- Definitions
- Graph ADT
- **Implementations**

GTG Implementation (net.datastructures)

- There are many ways to implement the Graph ADT.
- We will follow the textbook implementation.

Vertex and Edge Lists

- A graph consists of a collection of vertices V and a collection of edges E .
- Each of these will be represented as a Positional List (Ch.7.3).
- In `net.datastructures`, Positional Lists are implemented as doubly-linked lists.

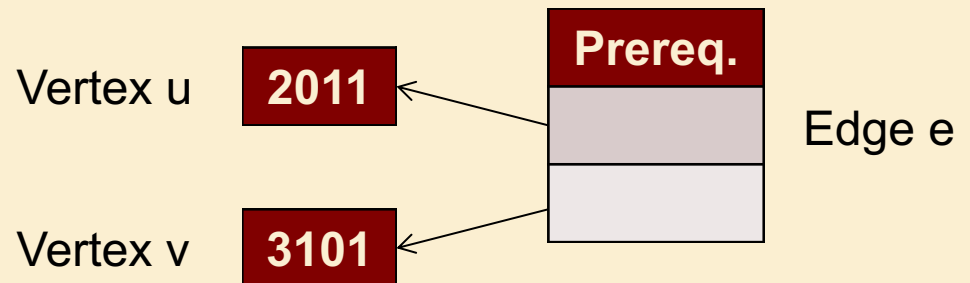
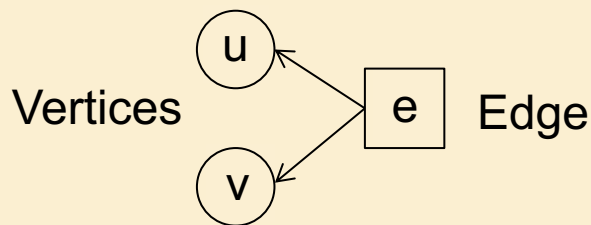


End of Lecture

Mar 22, 2018

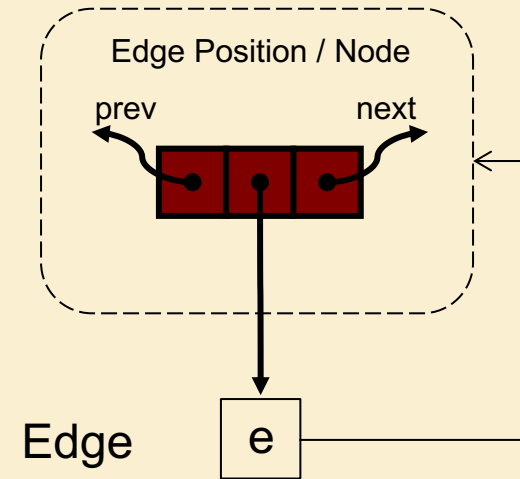
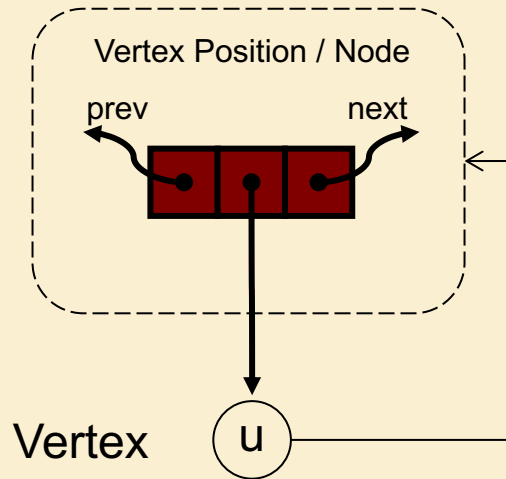
Vertices and Edges

- Each vertex v stores an element containing information about the vertex.
 - ❑ For example, if the graph represents course dependencies, the vertex element might store the course number.
- Each edge e stores an element containing information about the edge.
 - ❑ e.g., pre-requisite, co-requisite.
- In addition, each edge must store references to the vertices it connects.



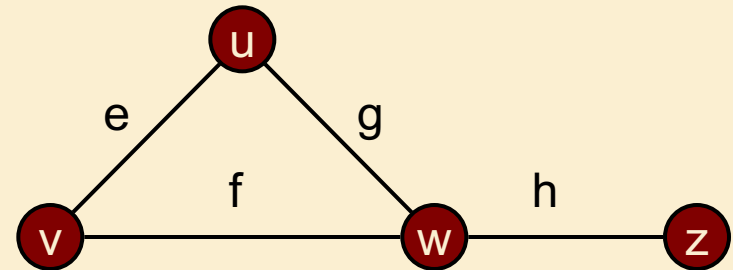
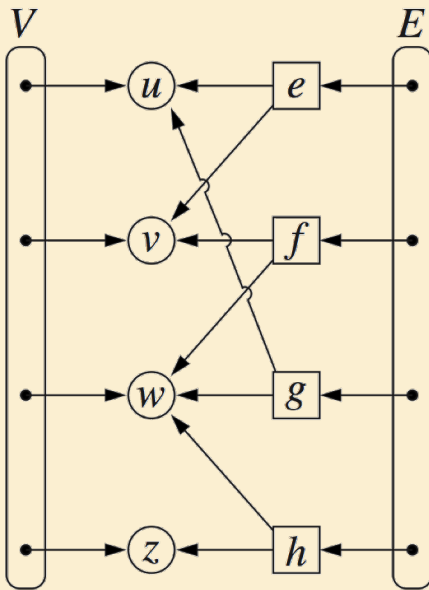
Vertices and Edges

- To facilitate efficient removal of vertices and edges, we will make both location aware:
 - A reference to the Position in the Positional List will be stored in the element.



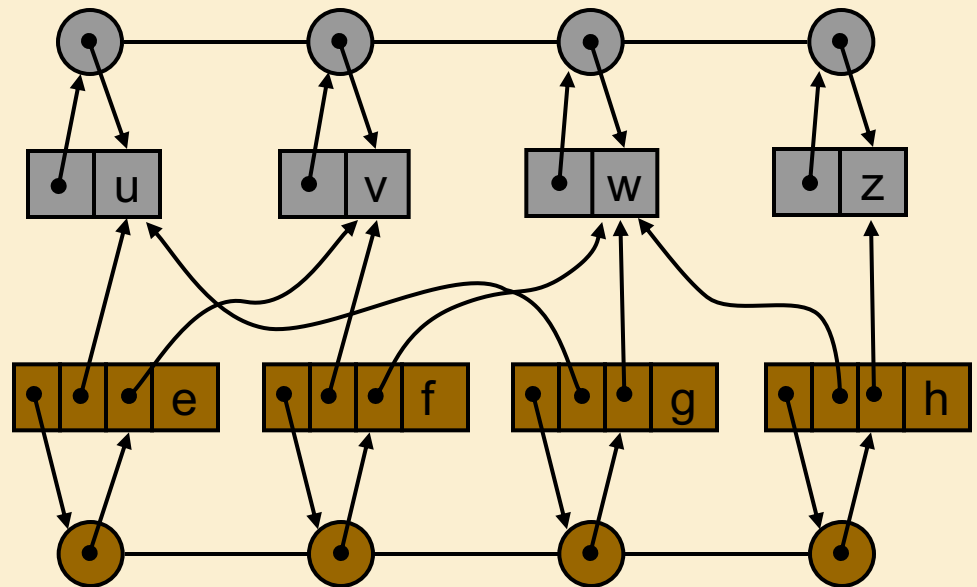
Edge List Implementation

- This organization yields an Edge List Structure



Vertex List

Edge List



Performance of Edge List Implementation

- Edge List implementation does not provide efficient access to edge information from vertex list.

<ul style="list-style-type: none">▪ n vertices, m edges▪ no parallel edges▪ no self-loops	Edge List
Space	$n + m$
incomingEdges(v) outgoingEdges(v)	m
getEdge(u, v)	m
insertVertex(x)	1
insertEdge(u, v, x)	1
removeVertex(v)	m
removeEdge(e)	1

Other Graph Implementations

➤ Can we come up with a graph implementation that improves the efficiency of these basic operations?

- ☐ Adjacency List
- ☐ Adjacency Map
- ☐ Adjacency Matrix

Other Graph Implementations

➤ Can we come up with a graph implementation that improves the efficiency of these basic operations?

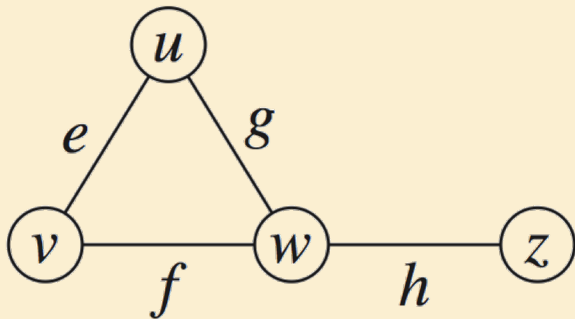
- ❑ **Adjacency List**

- ❑ Adjacency Map

- ❑ Adjacency Matrix

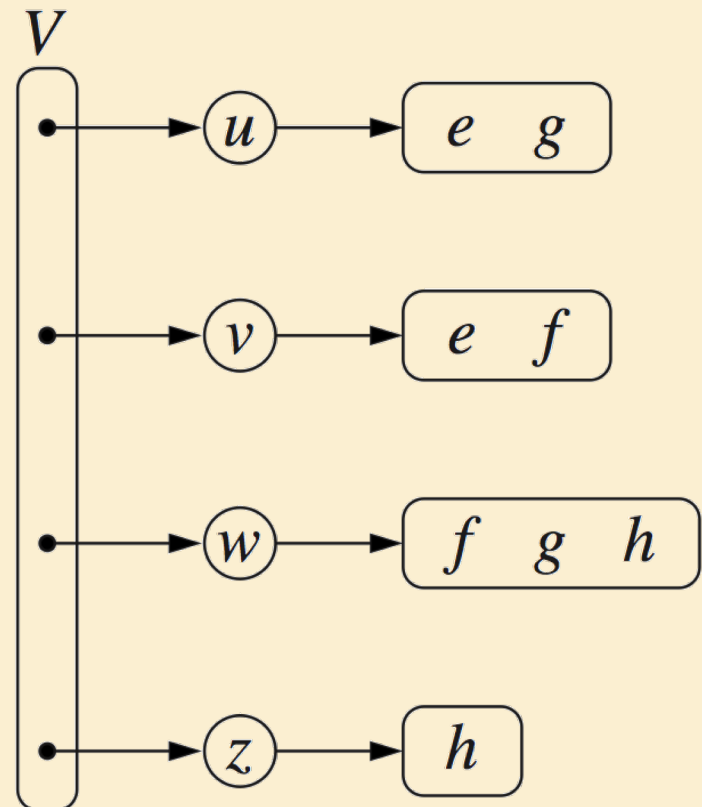
Adjacency List Implementation

- An Adjacency List implementation augments each vertex element with Positional Lists of incoming and outgoing edges.



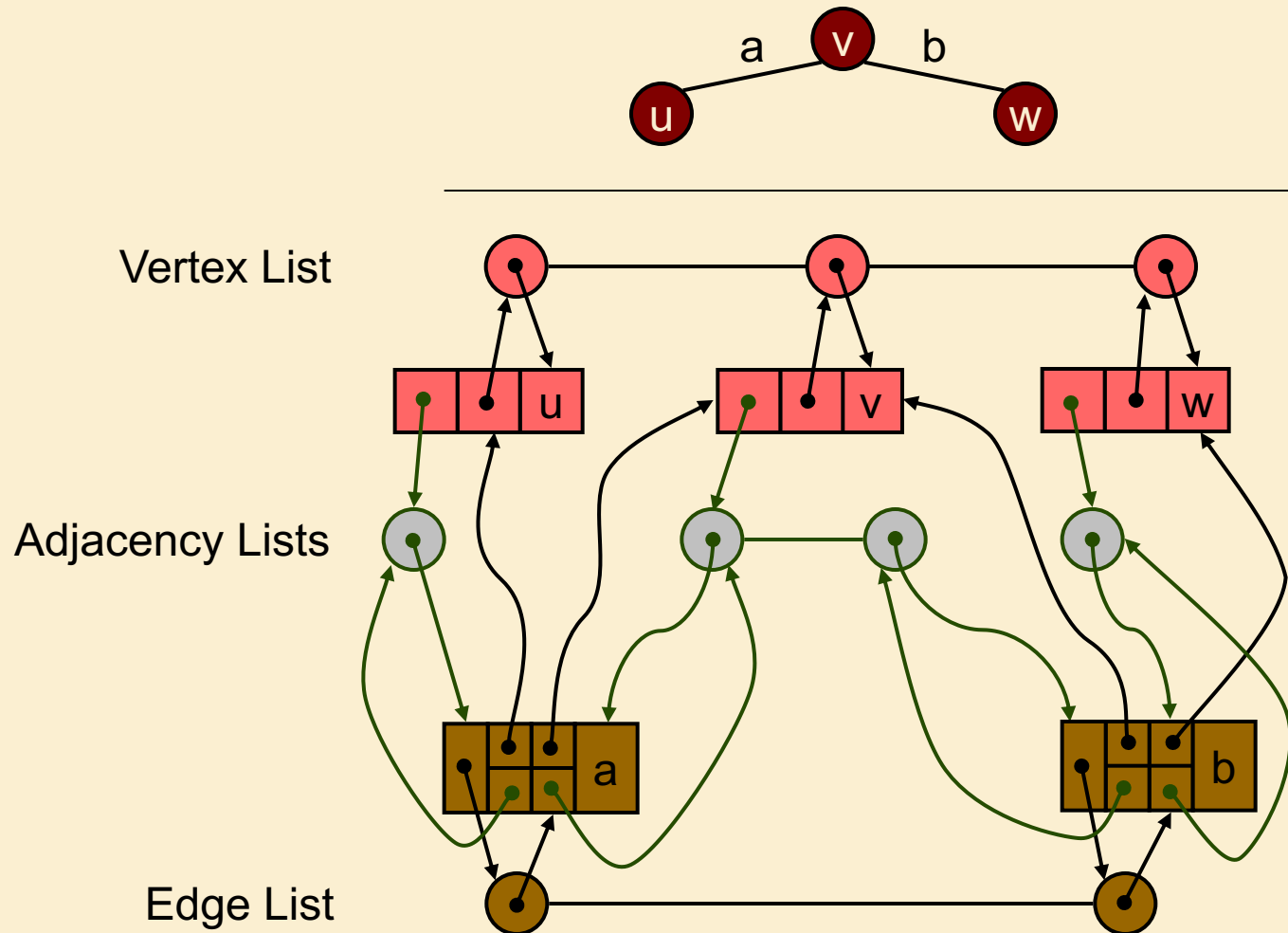
Vertex List

Adjacency Lists



Adjacency List Implementation

- An Adjacency List implementation augments each vertex element with lists of incoming and outgoing edges.



Performance of Adjacency List Implementation

- Adjacency List implementation improves efficiency without increasing space requirements.

<ul style="list-style-type: none"> ▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops 	Edge List	Adjacency List
Space	$n + m$	$n + m$
incomingEdges(v) outgoingEdges(v)	m	$\deg(v)$
getEdge(u, v)	m	$\min(\deg(u), \deg(v))$
insertVertex(x)	1	1
insertEdge(u, v, x)	1	1
removeVertex(v)	m	$\deg(v)$
removeEdge(e)	1	1

Other Graph Implementations

➤ Can we come up with a graph implementation that improves the efficiency of these basic operations?

- ❑ Adjacency List

- ❑ **Adjacency Map**

- ❑ Adjacency Matrix

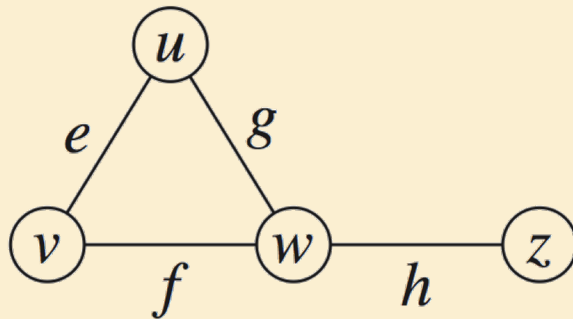
Adjacency Map Implementation

➤ An Adjacency Map implementation augments each vertex element with an Adjacency Map of edges

□ Each entry consists of:

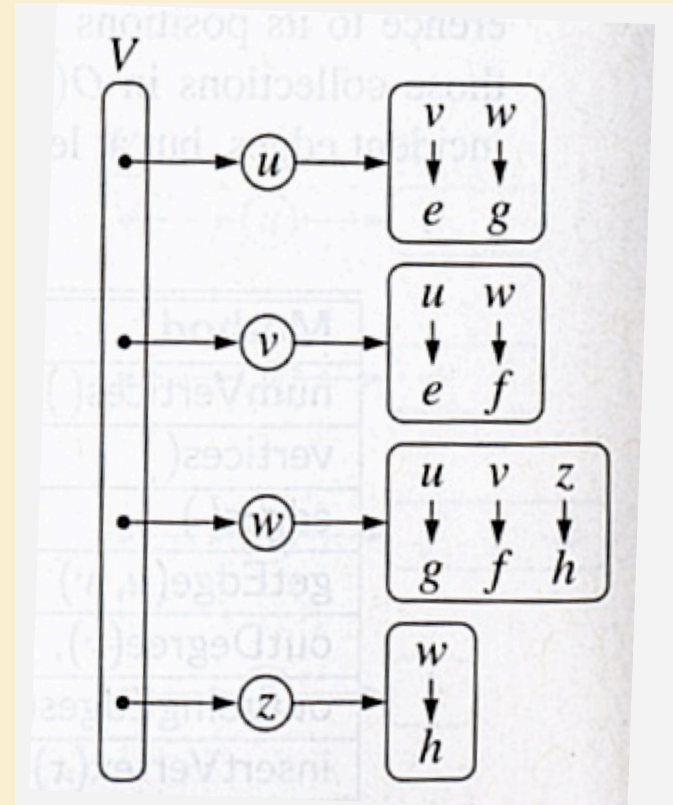
- ✧ Key = opposite vertex
- ✧ Value = edge

□ Implemented as a hash table.



Vertex List

Adjacency Maps



Performance of Adjacency Map Implementation

- Adjacency Map implementation improves expected run time of `getEdge(u,v)`:

<ul style="list-style-type: none"> ▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops 	Edge List	Adjacency List	Adjacency Map
Space	$n + m$	$n + m$	$n + m$
incomingEdges(v), outgoingEdges(v)	m	deg(v)	deg(v)
getEdge(u, v)	m	min(deg(u), deg(v))	1 (exp.)
insertVertex(x)	1	1	1
insertEdge(u, v, x)	1	1	1 (exp.)
removeVertex(v)	m	deg(v)	deg(v)
removeEdge(e)	1	1	1 (exp.)

Other Graph Implementations

➤ Can we come up with a graph implementation that improves the efficiency of these basic operations?

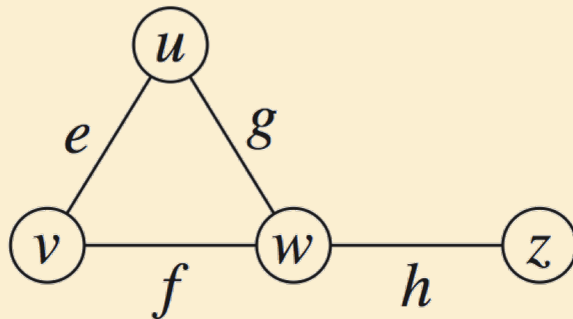
- ❑ Adjacency List

- ❑ Adjacency Map

- ❑ **Adjacency Matrix**

Adjacency Matrix Implementation

- In an Adjacency Matrix implementation we map each of the n vertices to an integer index from $[0 \dots n-1]$.
- Then a 2D $n \times n$ array A is maintained:
 - If edge (i, j) exists, $A[i, j]$ stores a reference to the edge.
 - If edge (i, j) does not exist, $A[i, j]$ is set to null.



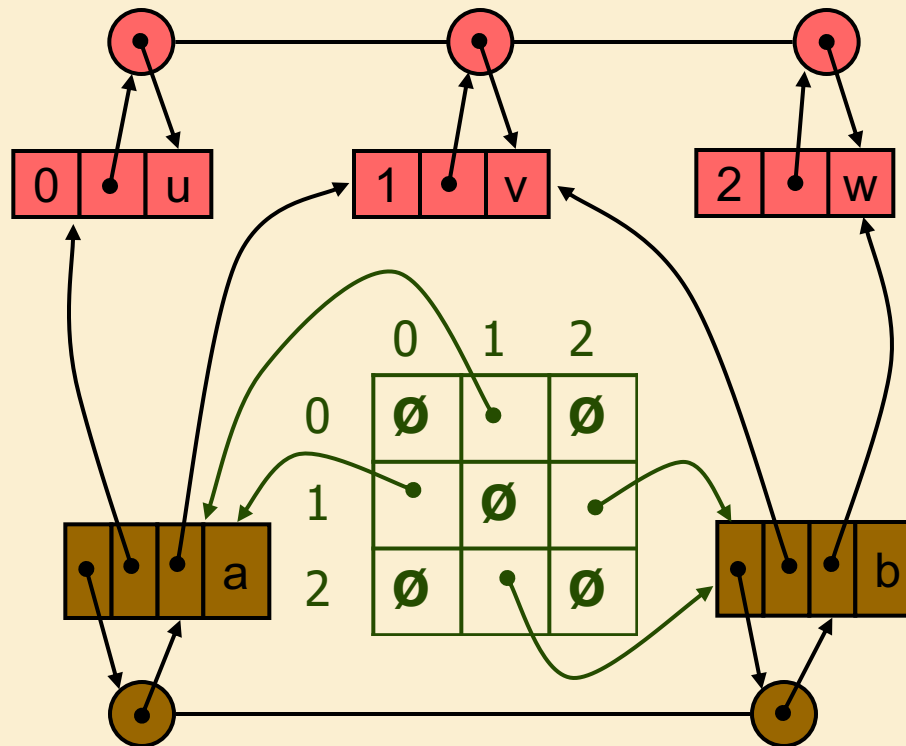
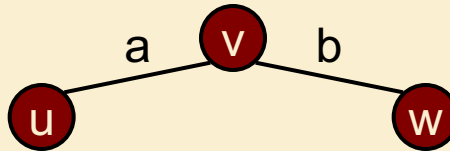
Vertex List

$u \longrightarrow 0$
 $v \longrightarrow 1$
 $w \longrightarrow 2$
 $z \longrightarrow 3$

Adjacency Matrix

	0	1	2	3
0		e	g	
1	e		f	
2	g	f		h
3			h	

Adjacency Matrix Structure



Performance of Adjacency Matrix Implementation

- Requires more space.
- Slow to get incoming / outgoing edges
- Very slow to insert or remove a vertex (array must be resized)

<ul style="list-style-type: none"> ▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops 	Edge List	Adjacency List	Adjacency Map	Adjacency Matrix
Space	$n + m$	$n + m$	$n + m$	n^2
incomingEdges(v), outgoingEdges(v)	m	deg(v)	deg(v)	n
getEdge(u, v)	m	min(deg(u), deg(v))	1 (exp.)	1
insertVertex(x)	1	1	1	n^2
insertEdge(u, v, x)	1	1	1 (exp.)	1
removeVertex(v)	m	deg(v)	deg(v)	n^2
removeEdge(e)	1	1	1 (exp.)	1

Outline

- Definitions
- Graph ADT
- Implementations

Outcomes

- By understanding this lecture, you should be able to:
 - ❑ Define basic terminology of graphs.
 - ❑ Use a graph ADT for appropriate applications.
 - ❑ Program standard implementations of the graph ADT.
 - ❑ Understand advantages and disadvantages of these implementations, in terms of space and run time.