

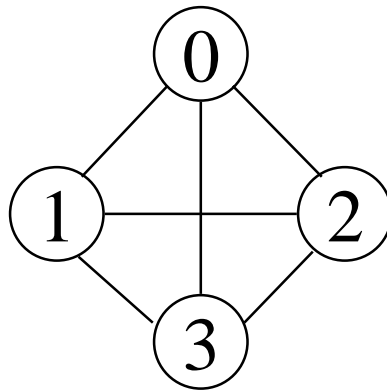
# GRAPHS

# Definition

- A **graph**  $G$  consists of two sets
  - a finite, nonempty set of vertices  $V(G)$
  - a finite, possible empty set of edges  $E(G)$
  - $G(V,E)$  represents a graph
- An **undirected graph** is one in which the pair of vertices in a edge is unordered,  $(v_0, v_1) = (v_1, v_0)$
- A **directed graph** is one in which each edge is a directed pair of vertices,  $\langle v_0, v_1 \rangle \neq \langle v_1, v_0 \rangle$

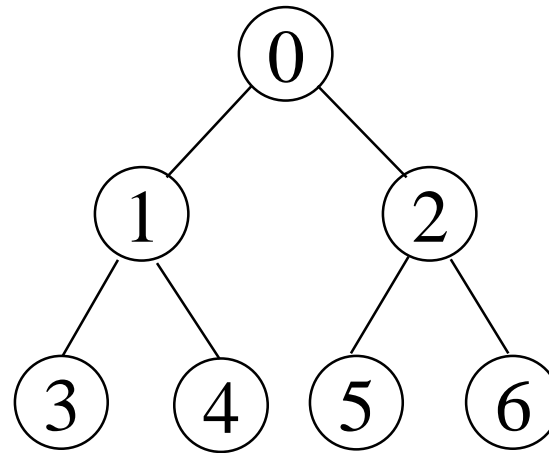
**tail**  $\xrightarrow{\hspace{1.5cm}}$  **head**

# Examples for Graph



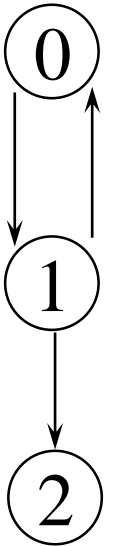
$G_1$

complete graph



$G_2$

incomplete graph



$G_3$

$$V(G_1) = \{0, 1, 2, 3\}$$

$$E(G_1) = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$$

$$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$$

$$E(G_2) = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\}$$

$$V(G_3) = \{0, 1, 2\}$$

$$E(G_3) = \{<0, 1>, <1, 0>, <1, 2>\}$$

complete undirected graph:  $n(n-1)/2$  edges

complete directed graph:  $n(n-1)$  edges

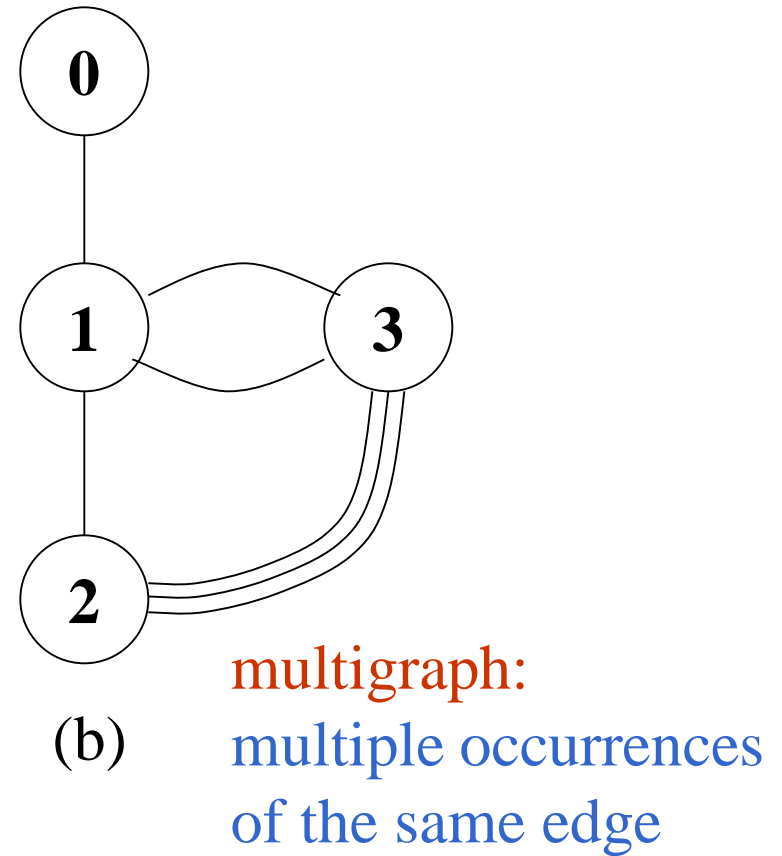
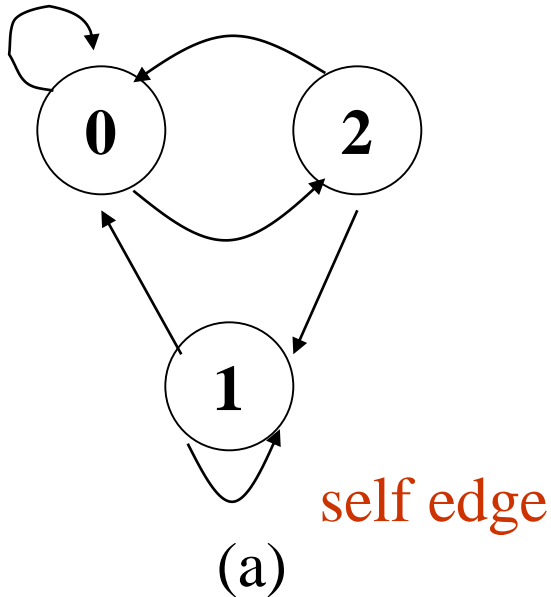
# Complete Graph

- A complete graph is a graph that has the maximum number of edges
  - for **undirected graph** with  $n$  vertices, the maximum number of edges is  $n(n-1)/2$
  - for **directed graph** with  $n$  vertices, the maximum number of edges is  $n(n-1)$
  - example:  $G_1$  is a complete graph

# Adjacent and Incident

- If  $(v_0, v_1)$  is an edge in an undirected graph,
  - $v_0$  and  $v_1$  are **adjacent**
  - The edge  $(v_0, v_1)$  is incident on vertices  $v_0$  and  $v_1$
- If  $\langle v_0, v_1 \rangle$  is an edge in a directed graph
  - $v_0$  is **adjacent to**  $v_1$ , and  $v_1$  is **adjacent from**  $v_0$
  - The edge  $\langle v_0, v_1 \rangle$  is incident on  $v_0$  and  $v_1$

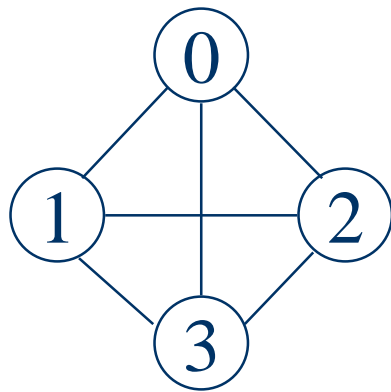
**\*Figure 6.3:**Example of a graph with feedback loops and a multigraph (p.260)



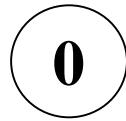
# Subgraph and Path

- A **subgraph** of  $G$  is a graph  $G'$  such that  $V(G')$  is a subset of  $V(G)$  and  $E(G')$  is a subset of  $E(G)$
- A **path** from vertex  $v_p$  to vertex  $v_q$  in a graph  $G$ , is a sequence of vertices,  $v_p, v_{i1}, v_{i2}, \dots, v_{in}, v_q$ , such that  $(v_p, v_{i1}), (v_{i1}, v_{i2}), \dots, (v_{in}, v_q)$  are edges in an undirected graph
- The **length of a path** is the number of edges on it

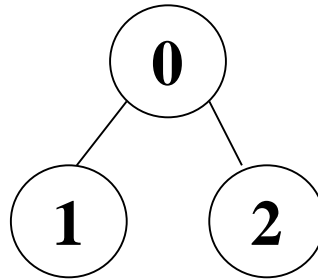
**Figure 6.4:** subgraphs of  $G_1$  and  $G_3$  (p.261)



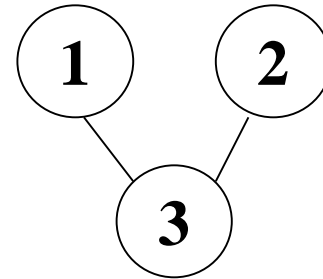
$G_1$



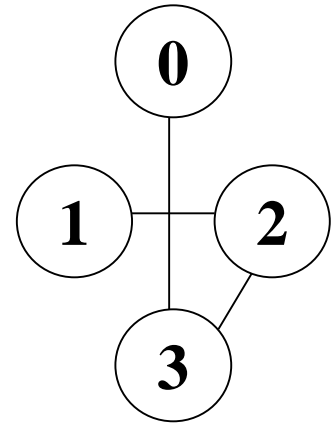
(i)



(ii)



(iii)

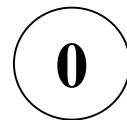


(iv)

(a) Some of the subgraph of  $G_1$

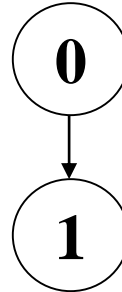


$G_3$

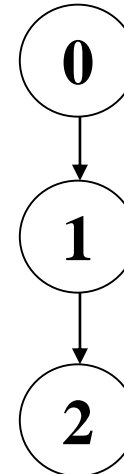


單一

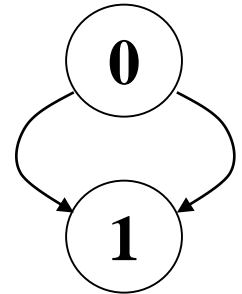
(i)



(ii)



(iii)



分開



(iv)

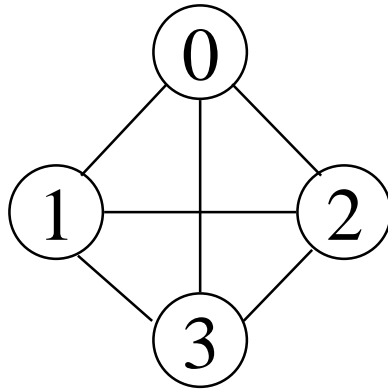
(b) Some of the subgraph of  $G_3$



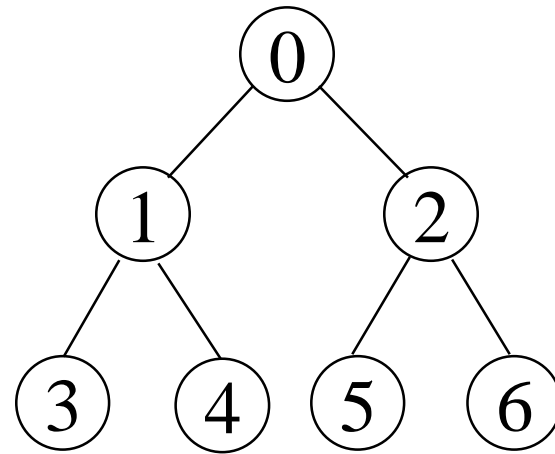
# Simple Path and Style

- A **simple path** is a path in which all vertices, except possibly the first and the last, are distinct
- A **cycle** is a simple path in which the first and the last vertices are the same
- In an undirected graph  $G$ , two **vertices**,  $v_0$  and  $v_1$ , are **connected** if there is a path in  $G$  from  $v_0$  to  $v_1$
- An undirected **graph** is **connected** if, for every pair of distinct vertices  $v_i, v_j$ , there is a path from  $v_i$  to  $v_j$

connected



$G_1$



$G_2$

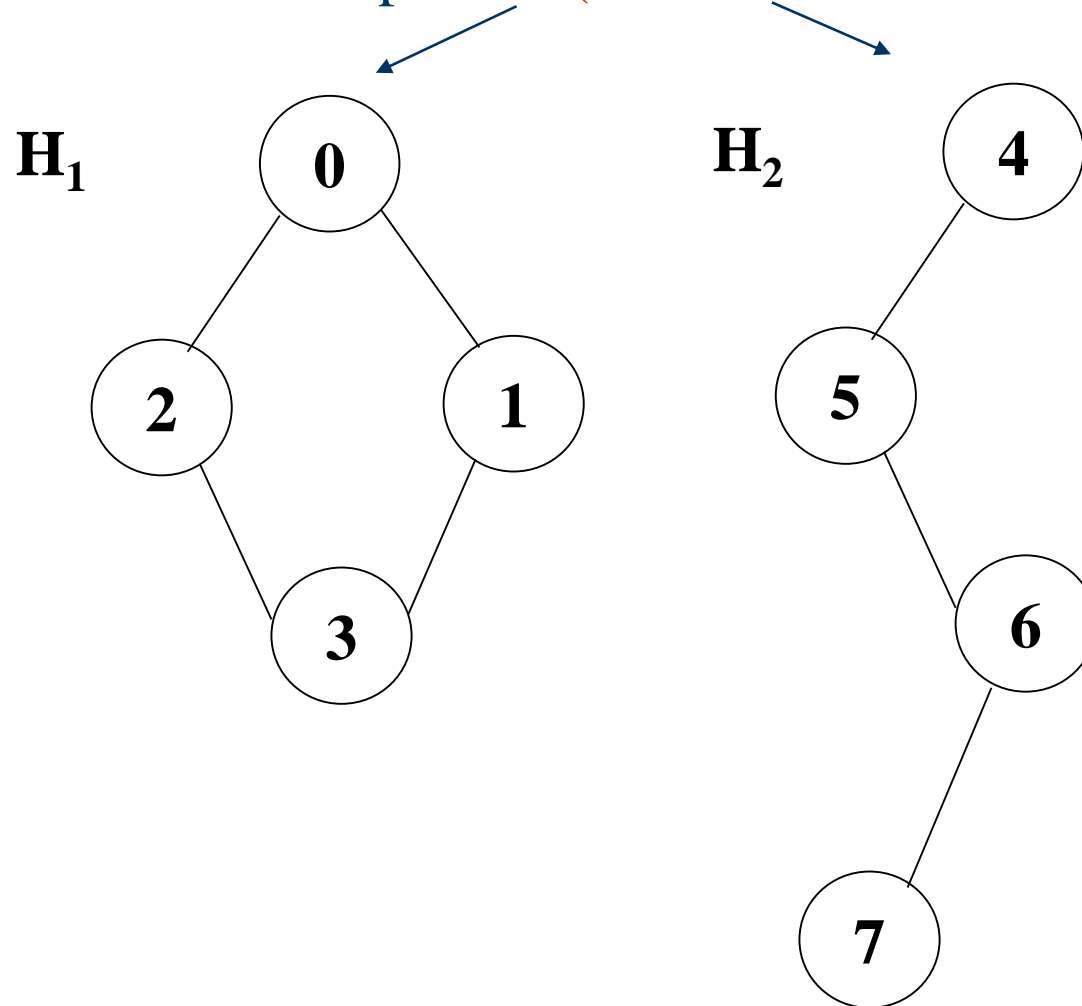
tree (acyclic graph)

# Connected Component

- A **connected component** of an undirected graph is a maximal connected subgraph.
- A **tree** is a graph that is connected and acyclic.
- A directed graph is **strongly connected** if there is a directed path from  $v_i$  to  $v_j$  and also from  $v_j$  to  $v_i$ .
- A **strongly connected component** is a maximal subgraph that is strongly connected.

**\*Figure 6.5: A graph with two connected components (p.262)**

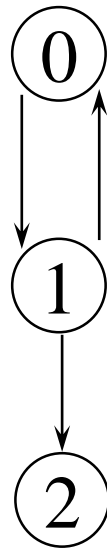
connected component (maximal connected subgraph)



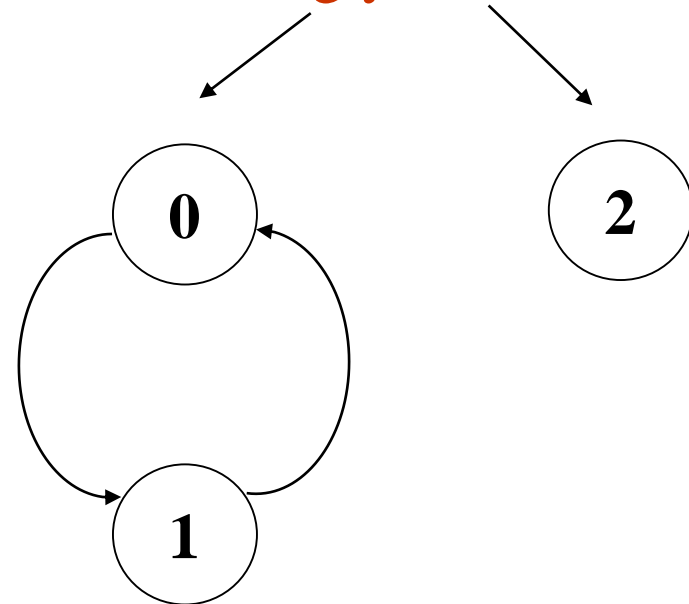
$G_4$  (not connected)  
CHAPTER 6

**\*Figure 6.6: Strongly connected components of  $G_3$  (p.262)**

not strongly connected      strongly connected component  
(maximal strongly connected subgraph)



$G_3$



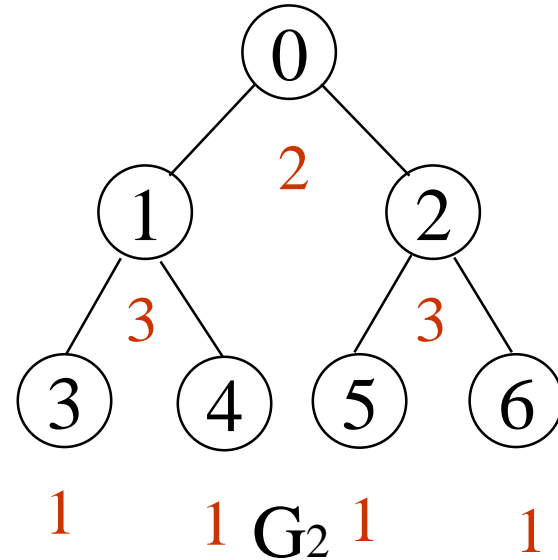
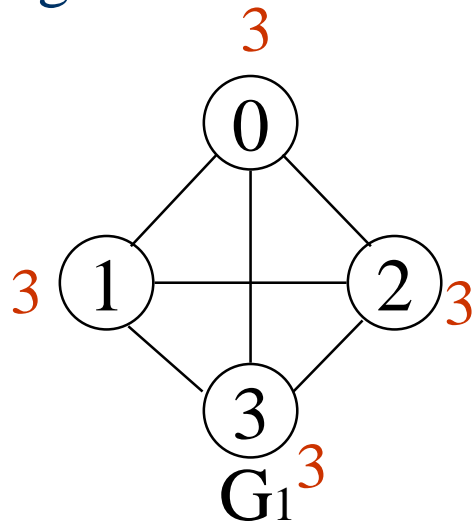
# Degree

- The **degree** of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the **in-degree** of a vertex  $v$  is the number of edges that have  $v$  as the head
  - the **out-degree** of a vertex  $v$  is the number of edges that have  $v$  as the tail
  - if  $d_i$  is the degree of a vertex  $i$  in a graph  $G$  with  $n$  vertices and  $e$  edges, the number of edges is

$$e = \left( \sum_{i=0}^{n-1} d_i \right) / 2$$

undirected graph

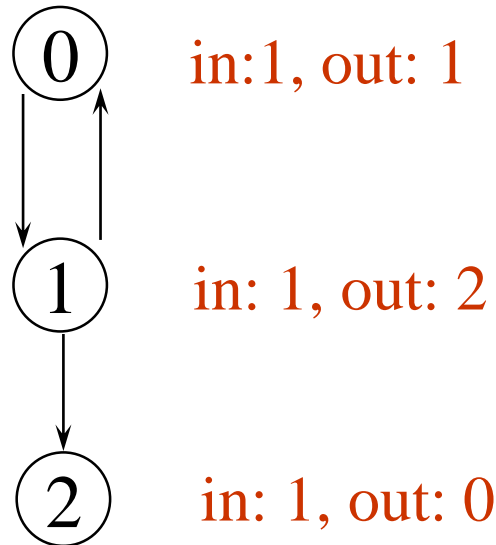
degree



directed graph

in-degree

out-degree



# ADT for Graph

structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all  $graph \in Graph$ ,  $v$ ,  $v_1$  and  $v_2 \in Vertices$

*Graph* Create() $::=$ return an empty graph

*Graph* InsertVertex( $graph$ ,  $v$ ) $::=$  return a graph with  $v$  inserted.  $v$  has no incident edge.

*Graph* InsertEdge( $graph$ ,  $v_1, v_2$ ) $::=$  return a graph with new edge between  $v_1$  and  $v_2$

*Graph* DeleteVertex( $graph$ ,  $v$ ) $::=$  return a graph in which  $v$  and all edges incident to it are removed

*Graph* DeleteEdge( $graph$ ,  $v_1$ ,  $v_2$ ) $::=$ return a graph in which the edge ( $v_1$ ,  $v_2$ ) is removed

*Boolean* IsEmpty( $graph$ ) $::=$  if ( $graph == empty\ graph$ ) return TRUE  
else return FALSE

*List* Adjacent( $graph, v$ ) $::=$  return a list of all vertices that are adjacent to  $v$



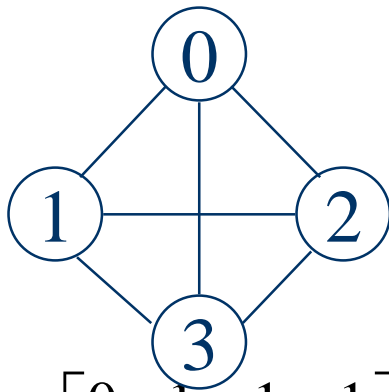
# Graph Representations

- Adjacency Matrix
- Adjacency Lists
- Adjacency Multilists

# Adjacency Matrix

- Let  $G=(V,E)$  be a graph with  $n$  vertices.
- The **adjacency matrix** of  $G$  is a two-dimensional  $n$  by  $n$  array, say `adj_mat`
- If the edge  $(v_i, v_j)$  is in  $E(G)$ , `adj_mat[i][j]=1`
- If there is no such edge in  $E(G)$ , `adj_mat[i][j]=0`
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

# Examples for Adjacency Matrix



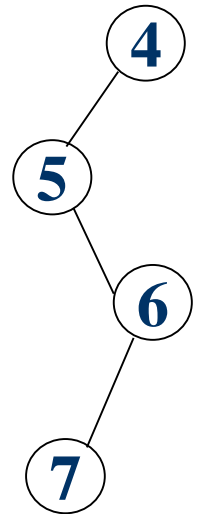
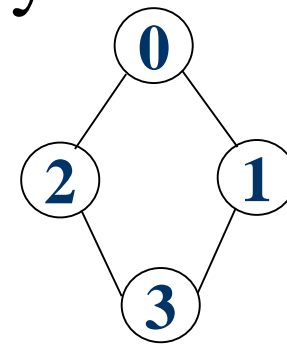
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$G_1$



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

$G_2$



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

$G_4$

symmetric

undirected:  $n^2/2$   
directed:  $n^2$

# Merits of Adjacency Matrix

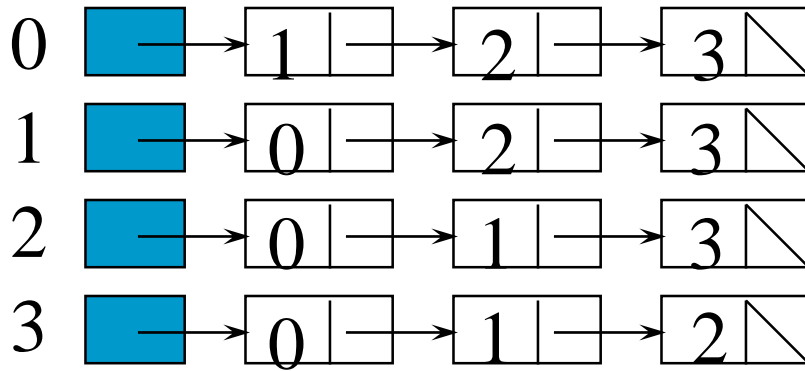
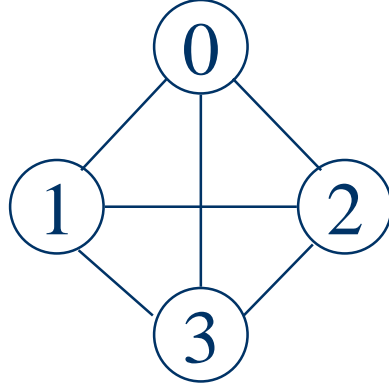
- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is  $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For a digraph, the row sum is the out\_degree, while the column sum is the in\_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j, i] \quad outd(vi) = \sum_{j=0}^{n-1} A[i, j]$$

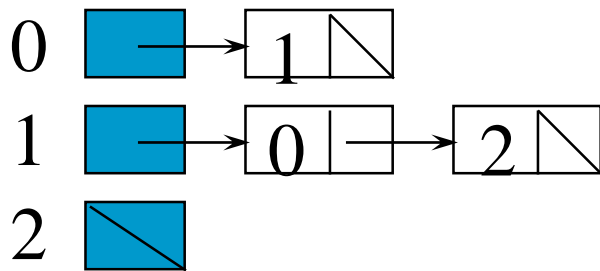
# Data Structures for Adjacency Lists

Each row in adjacency matrix is represented as an adjacency list.

```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use *
```



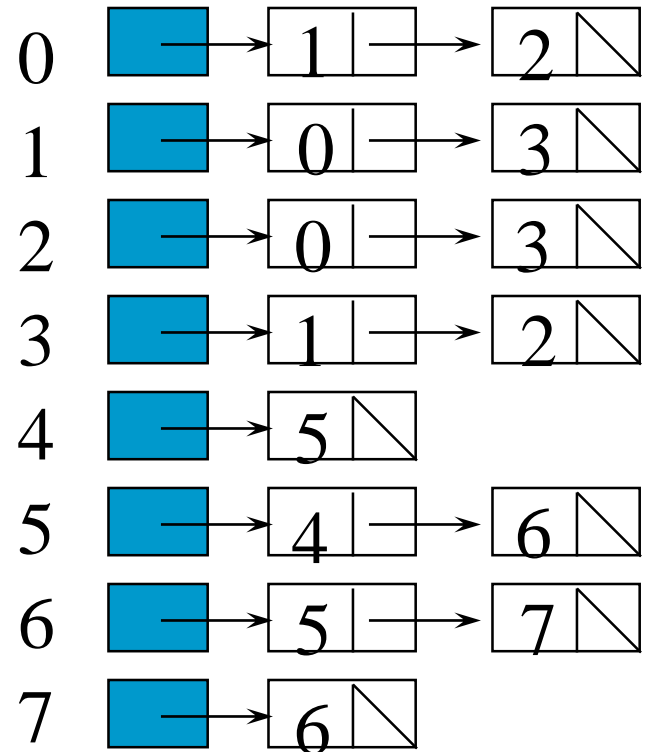
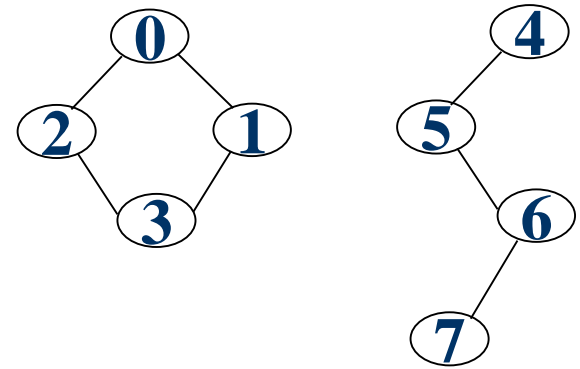
$G_1$



$G_3$



CHAPTER



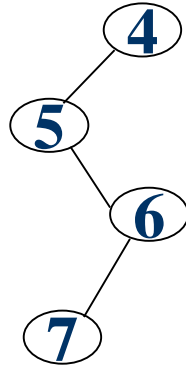
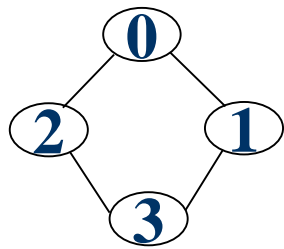
$G_4$

An undirected graph with  $n$  vertices and  $e$  edges  $\implies n$  head nodes and  $2e$  list nodes

# Interesting Operations

- **degree of a vertex** in an undirected graph
  - # of nodes in adjacency list
- **# of edges** in a graph
  - determined in  $O(n+e)$
- **out-degree** of a vertex in a directed graph
  - # of nodes in its adjacency list
- **in-degree** of a vertex in a directed graph
  - traverse the whole data structure

# Compact Representation



node[0] ... node[n-1]: starting point for vertices

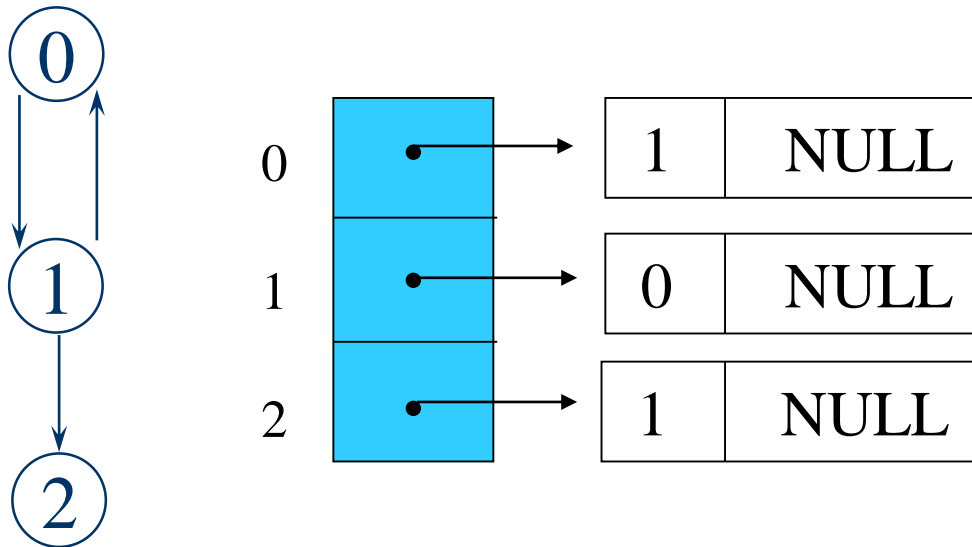
node[n]: n+2e+1

node[n+1] ... node[n+2e]: head node of edge

[0]	9		[8]	23		[16]	2	
[1]	11	0	[9]	1	4	[17]	5	
[2]	13		[10]	2	5	[18]	4	
[3]	15	1	[11]	0		[19]	6	
[4]	17		[12]	3	6	[20]	5	
[5]	18	2	[13]	0		[21]	7	
[6]	20		[14]	3	7	[22]	6	
[7]	22	3	[15]	1				



**Figure 6.10:** Inverse adjacency list for  $G_3$

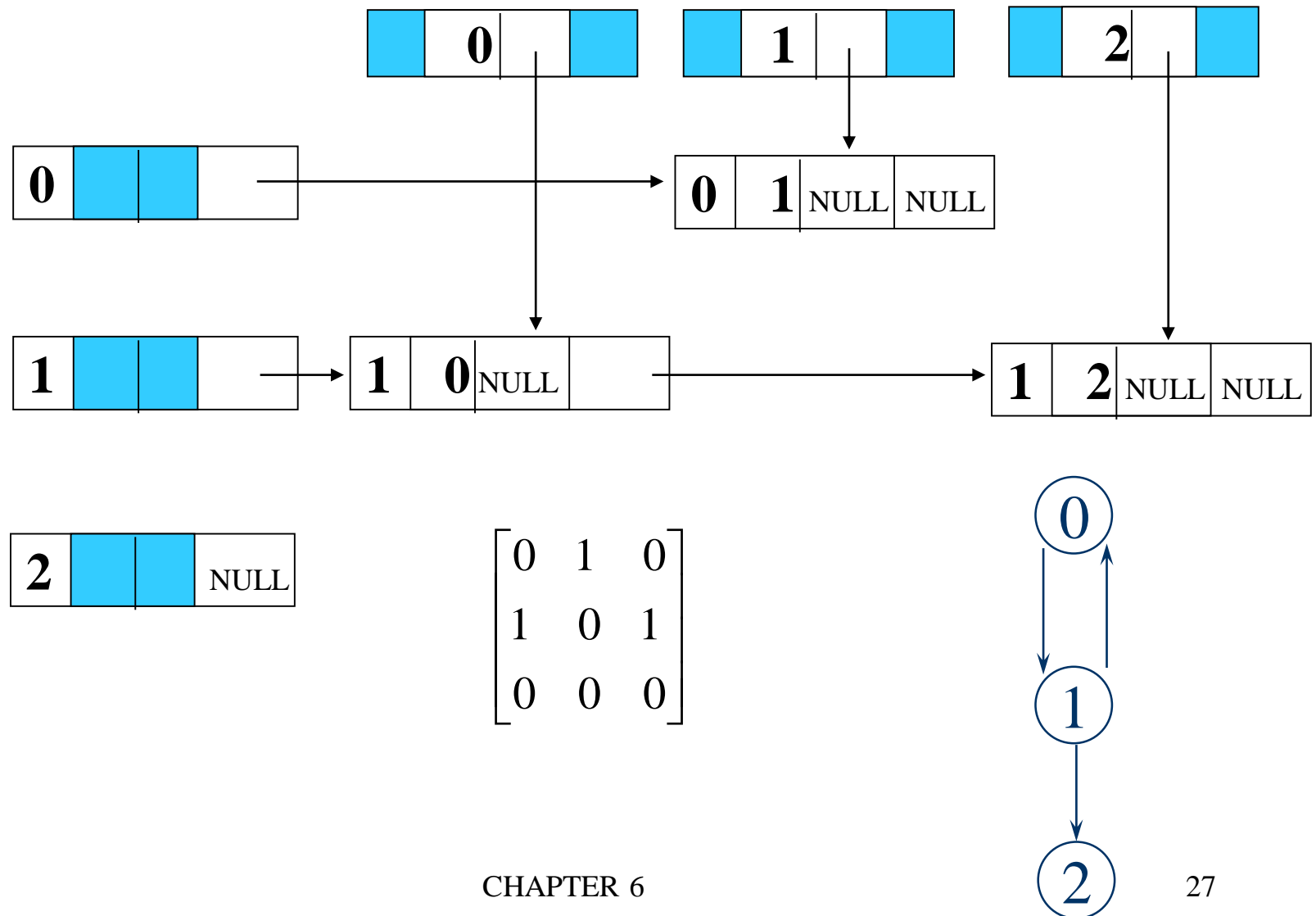


Determine in-degree of a vertex in a fast way.

**Figure 6.11: Alternate node structure for adjacency lists (p.267)**

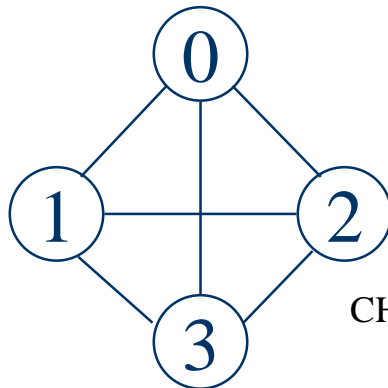
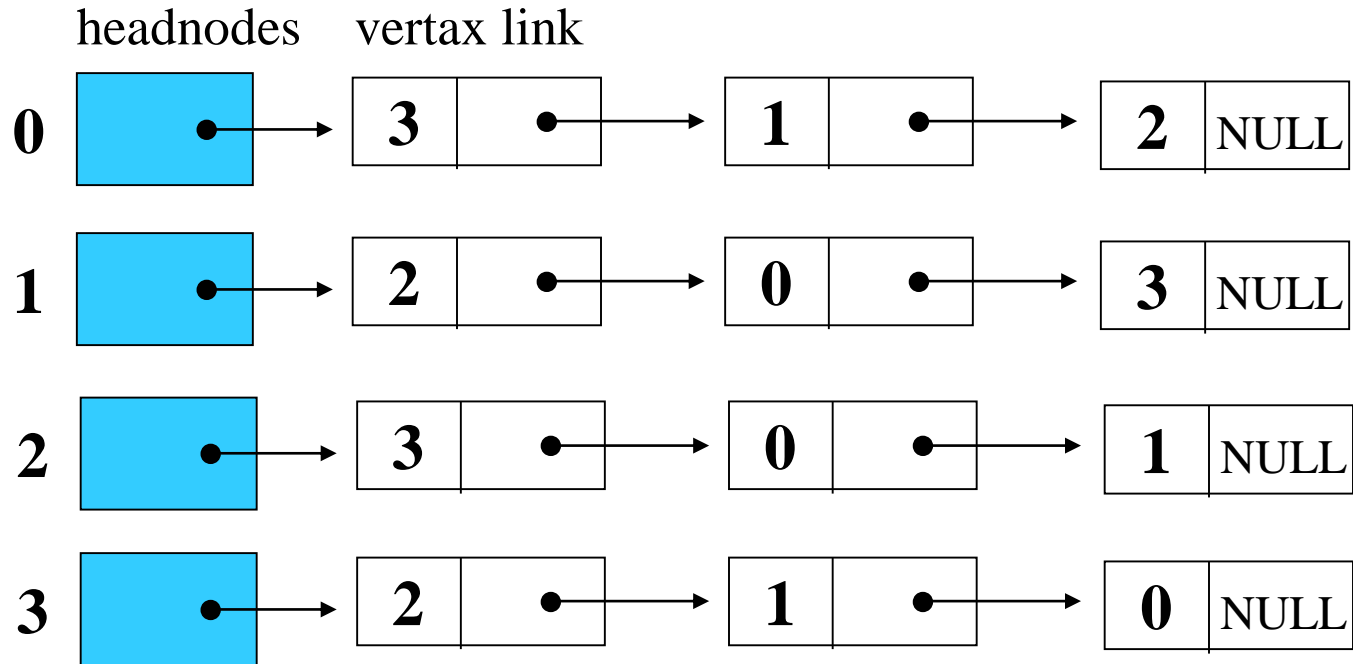
tail	head	column link for head	row link for tail
------	------	----------------------	-------------------

**Figure 6.12:** Orthogonal representation for graph  $G_3$ (p.268)



**Figure 6.13:** Alternate order adjacency list for  $G_1$  (p.268)

Order is of no significance.



# Adjacency Multilists

- An edge in an undirected graph is represented by two nodes in adjacency list representation.

- Adjacency Multilists

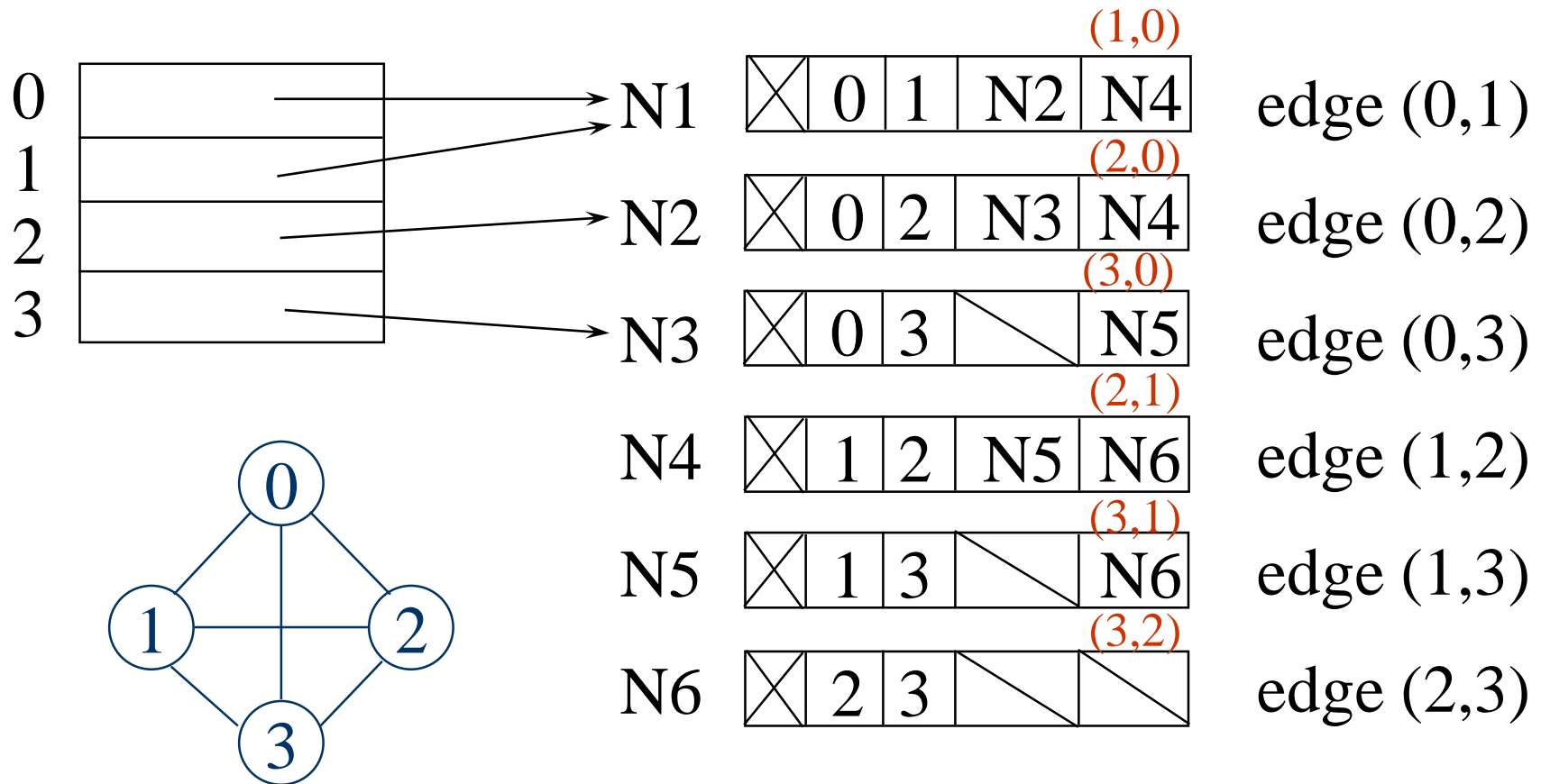
- lists in which nodes may be shared among several lists.

(an edge is shared by two different paths)

marked	vertex1	vertex2	path1	path2
--------	---------	---------	-------	-------

# Example for Adjacency Multlists

Lists: vertex 0: M1->M2->M3, vertex 1: M1->M4->M5  
 vertex 2: M2->M4->M6, vertex 3: M3->M5->M6



# Adjacency Multilists

```
typedef struct edge *edge_pointer;  
typedef struct edge {  
    short int marked;  
    int vertex1, vertex2;  
    edge_pointer path1, path2;  
};  
edge_pointer graph[MAX_VERTICES];
```

marked	vertex1	vertex2	path1	path2
--------	---------	---------	-------	-------

# Some Graph Operations

## ■ Traversal

Given  $G=(V,E)$  and vertex  $v$ , find all  $w \in V$ , such that  $w$  connects  $v$ .

- Depth First Search (DFS)

preorder tree traversal

- Breadth First Search (BFS)

level order tree traversal

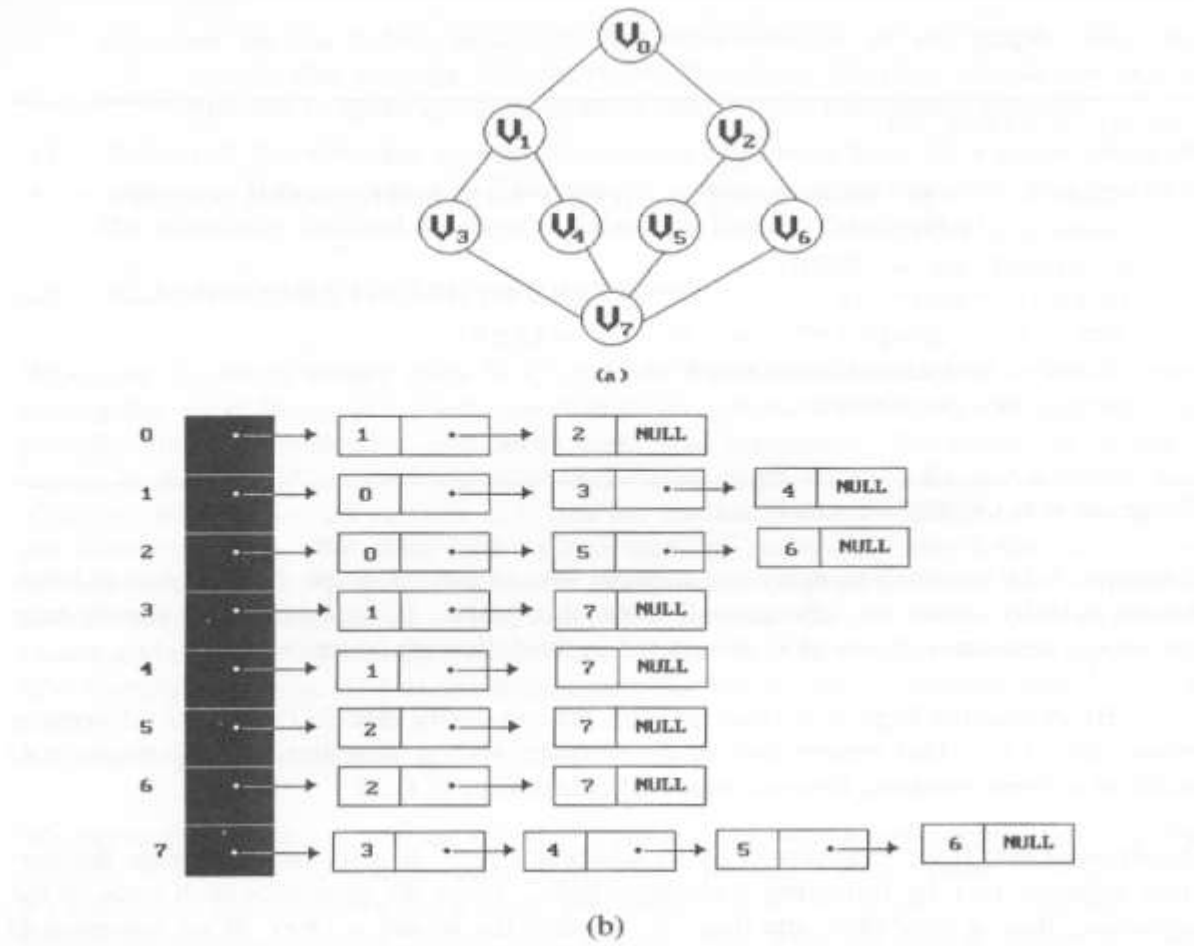
## ■ Connected Components

## ■ Spanning Trees



**\*Figure 6.19: Graph  $G$  and its adjacency lists (p.274)**

depth first search:  $v_0, v_1, v_3, v_7, v_4, v_5, v_2, v_6$



breadth first search:  $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$

# Depth First Search

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs(int v)
{
    node_pointer w;
    visited[v]= TRUE;
    printf("%5d", v);
    for (w=graph[v]; w; w=w->link)
        if (!visited[w->vertex])
            dfs(w->vertex);
}
```

Data structure  
adjacency list:  $O(e)$   
adjacency matrix:  $O(n^2)$

# Breadth First Search

```
typedef struct queue *queue_pointer;  
typedef struct queue {  
    int vertex;  
    queue_pointer link;  
};  
void addq(queue_pointer *,  
          queue_pointer *, int);  
int deleteq(queue_pointer *);
```

# Breadth First Search *(Continued)*

```
void bfs(int v)
{
    node_pointer w;
    queue_pointer front, rear;
    front = rear = NULL;
    printf("%5d", v);
    visited[v] = TRUE;
    addq(&front, &rear, v);
```

adjacency list:  $O(e)$   
adjacency matrix:  $O(n^2)$

```

while (front) {
    v= deleteq(&front);
    for (w=graph[v]; w; w=w->link)
        if (!visited[w->vertex]) {
            printf("%5d", w->vertex);
            addq(&front, &rear, w->vertex);
            visited[w->vertex] = TRUE;
        }
    }
}

```

# Connected Components

```
void connected(void)
{
    for (i=0; i<n; i++) {
        if (!visited[i]) {
            dfs(i);
            printf("\n");
        }
    }
}
```

adjacency list:  $O(n+e)$   
adjacency matrix:  $O(n^2)$