Chapter 6
Graphs

Yi-Fen Liu

Department of IECS, FCU

References:
- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures (2nd Edition)*
- Slides are credited from Prof. Chung, NTHU

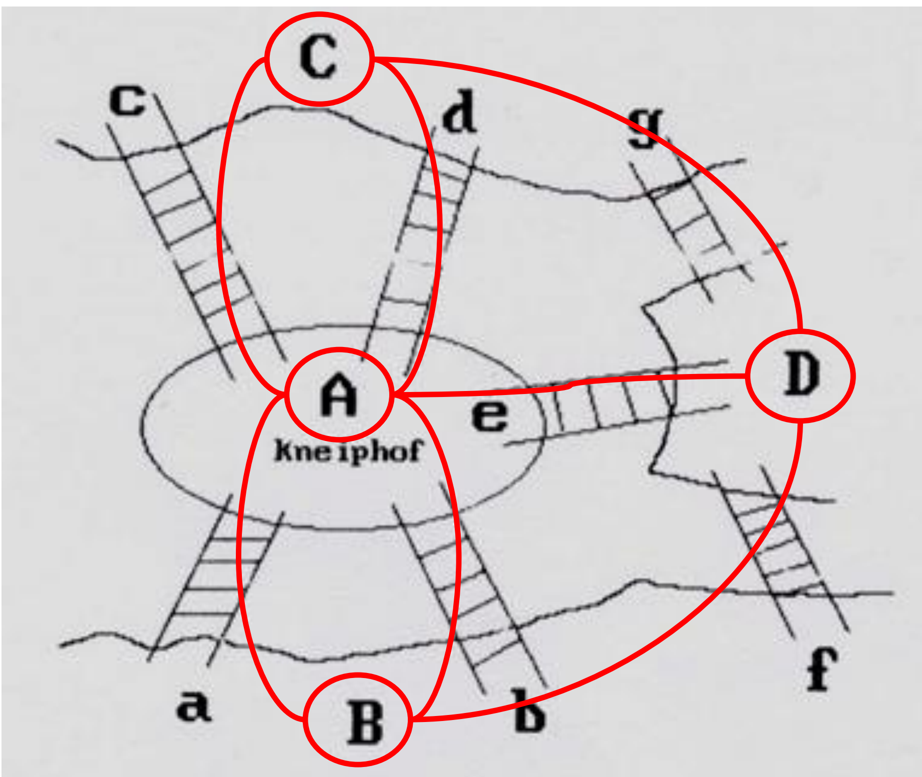
Outline

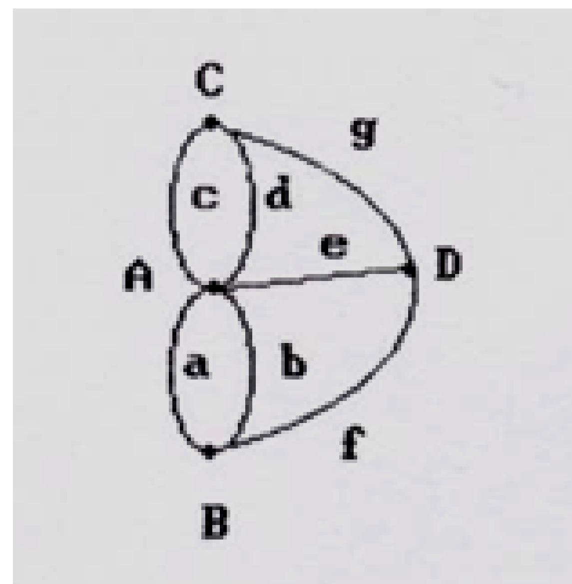
- The Graph Abstract Data Type
- Elementary Graph Operations
- Minimum Cost Spanning Trees
- Shortest Paths
- Topological Sorts

THE GRAPH ADT

The Graph ADT (1)

- Introduction
  - A graph problem example: Köenigsberg bridge problem





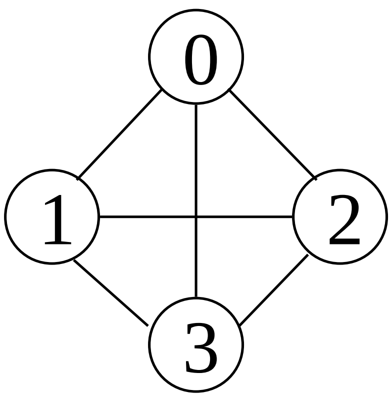
The Graph ADT (2)

- Definitions
  - 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 an 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, $<v_0, v_1>$!= $<v_1, v_0>$

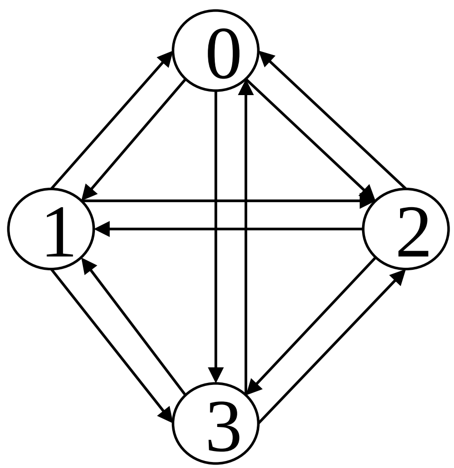
tail $\longrightarrow$ head

The Graph ADT (3)

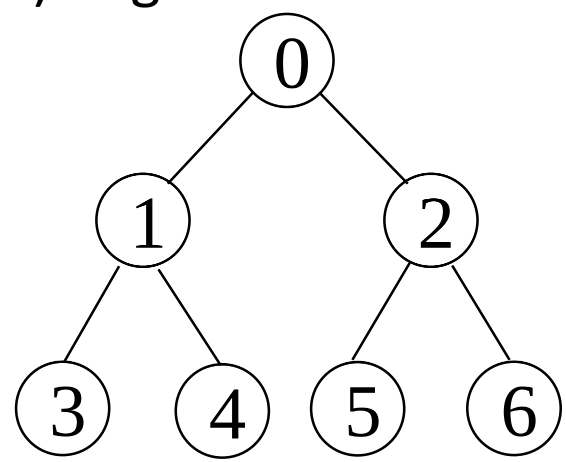
- Examples for Graph
  - Complete undirected graph: n(n-1)/2 edges
  - Complete directed graph: n(n-1) edges



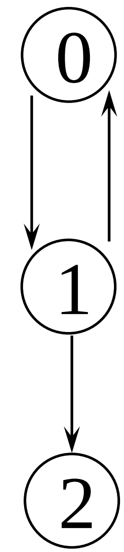
G1



complete graph



G2



G3

incomplete graph

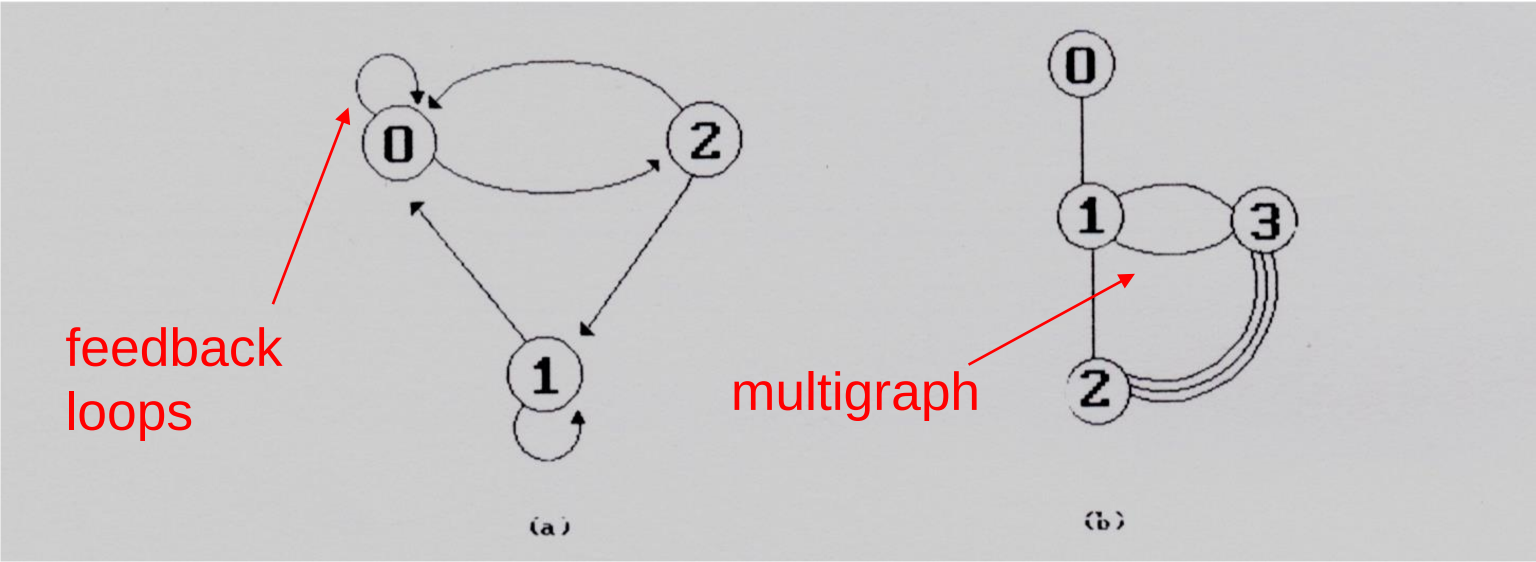
$V(G_1)=\{0,1,2,3\}$ $E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

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

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

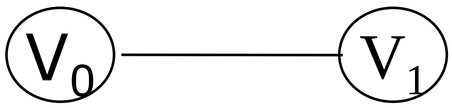
The Graph ADT (4)

- Restrictions on graphs
  - A graph may not have an edge from a vertex, *i*, back to itself. Such edges are known as self loops
  - A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data referred to as a multigraph

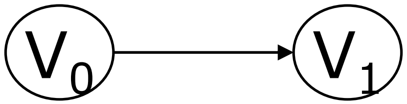


The Graph ADT (5)

- 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

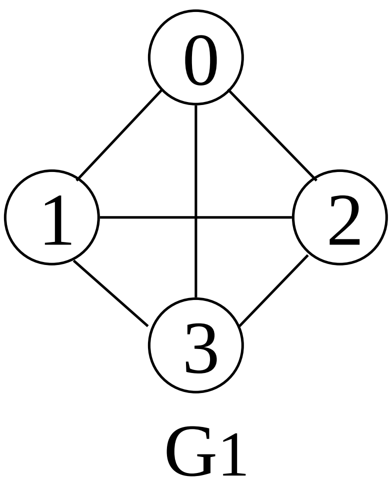


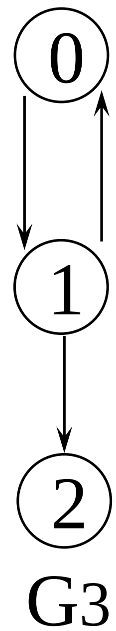
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

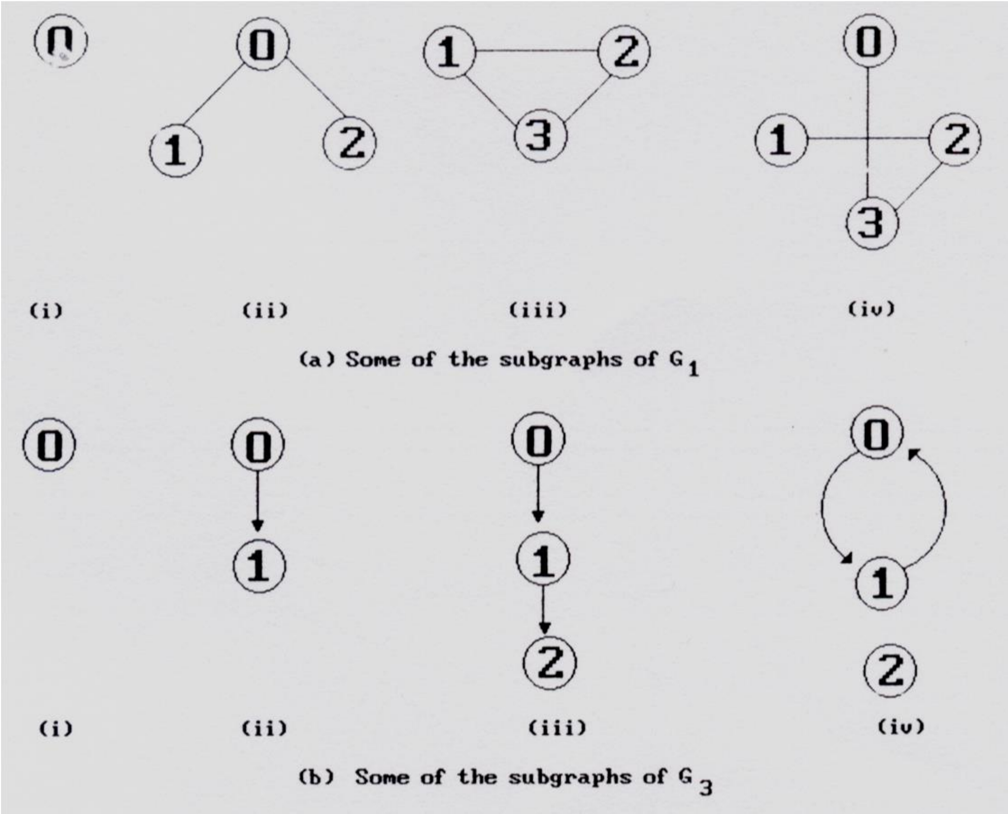


The Graph ADT (6)

- A subgraph of G is a graph G' such that V(G') $\subseteq$ V(G) and E(G') $\subseteq$ E(G).

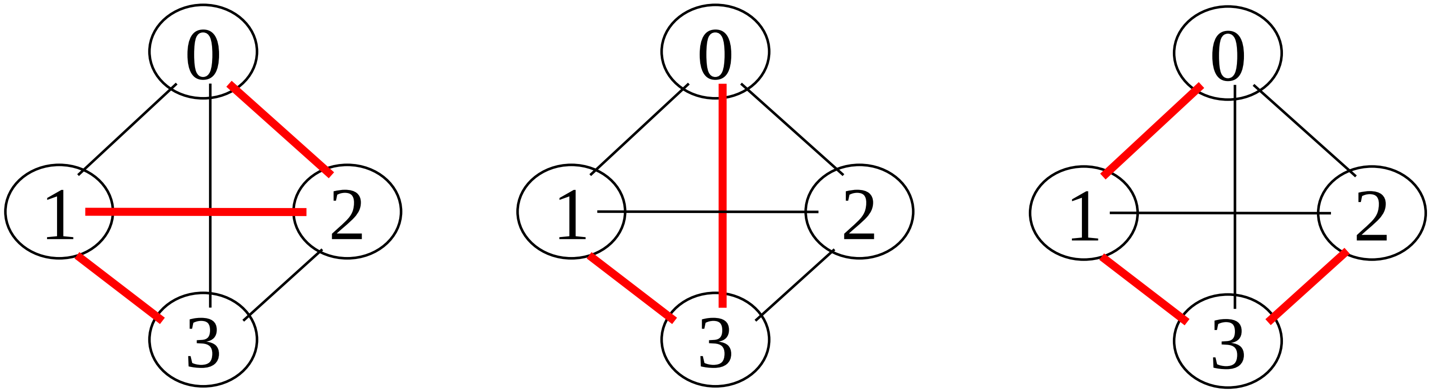






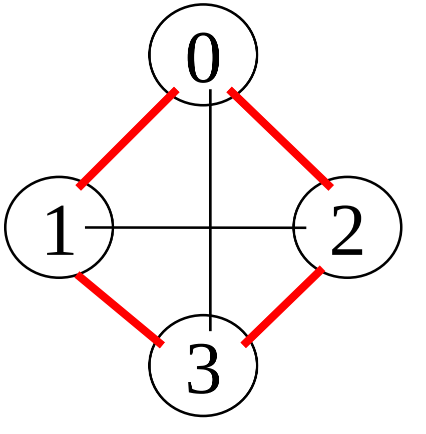
The Graph ADT (7)

- Path
  - A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p, v_1, v_2, ..., v_n, v_q, such that (v_p, v_1), (v_1, v_2), ..., (v_n, v_q) are edges in an undirected graph
    - A path such as (0, 2), (2, 1), (1, 3) is also written as 0, 2, 1, 3
  - The length of a path is the number of edges on it



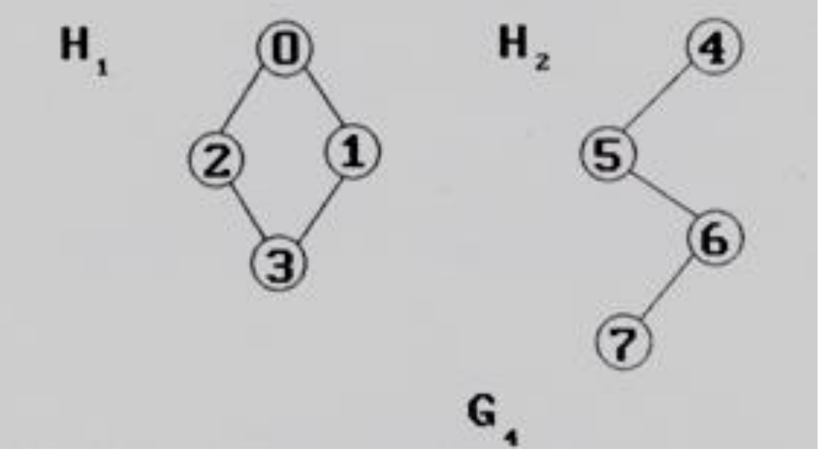
The Graph ADT (8)

- Simple path and cycle
  - Simple path (simple directed path)
    - 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



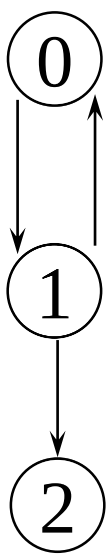
The Graph ADT (9)

- Connected graph
  - 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 component
  - A connected component of an undirected graph is a maximal connected subgraph
  - A tree is a graph that is connected and acyclic (i.e, has no cycle)



The Graph ADT (10)

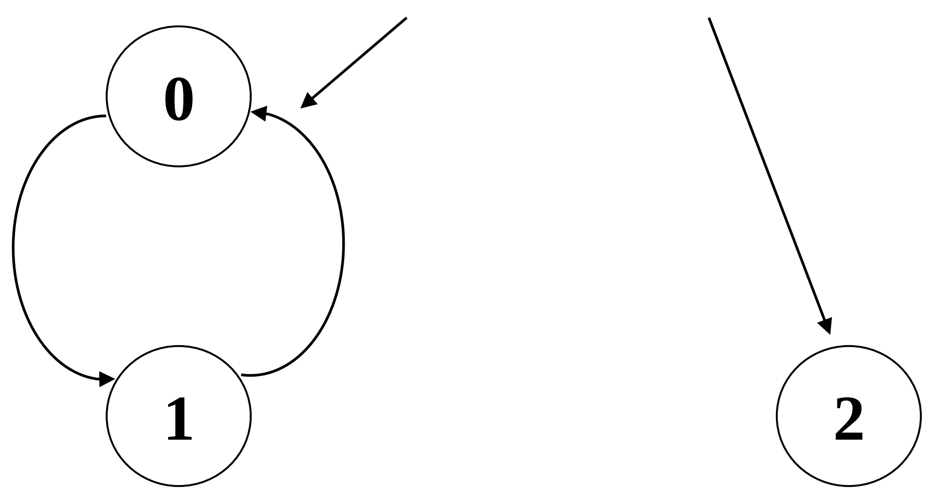
- Strongly Connected Component
  - A directed graph is strongly connected if there is a directed path from v_i to v_j and also from v_j to v_i
  - A strongly connected component is a maximal subgraph that is strongly connected



G3

not strongly connected

strongly connected component
(maximal strongly connected subgraph)



The Graph ADT (11)

- Degree
  - The degree of a vertex is the number of edges incident to that vertex
- For directed graph
  - in-degree (v) : the number of edges that have v as the head
  - out-degree (v) : 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

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

The Graph ADT (12)

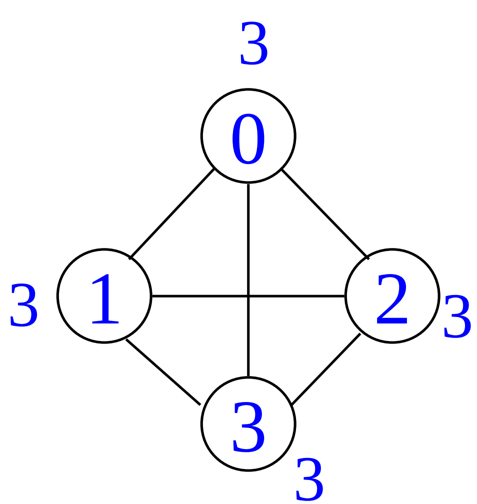
- We shall refer to a directed graph as a digraph. When we us the term graph, we assume that it is an undirected graph

directed graph

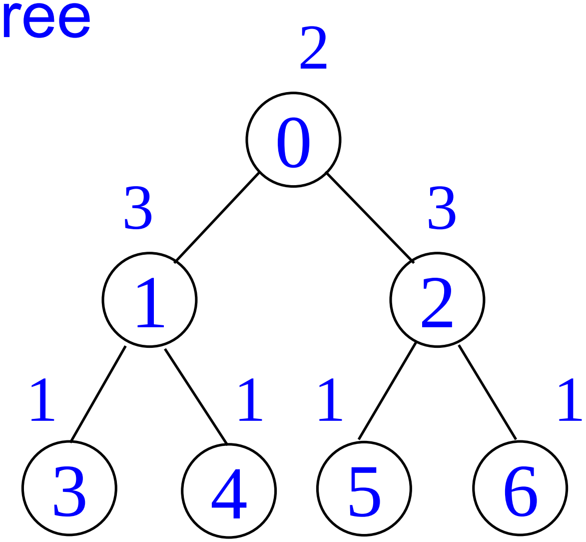
undirected graph

in-degree & out-degree

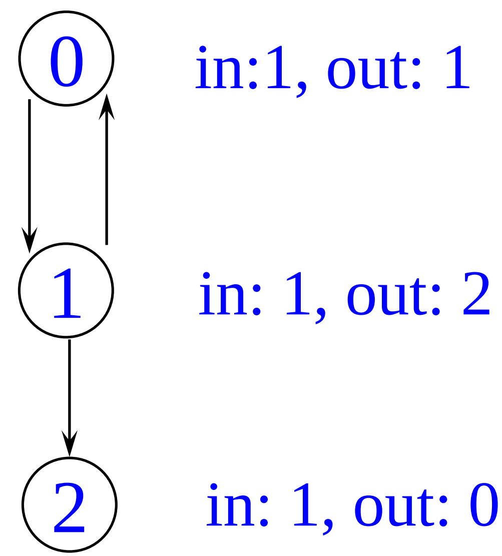
degree



G1



G2



G3

The Graph ADT (13)

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 $v_2 \in Vertices$

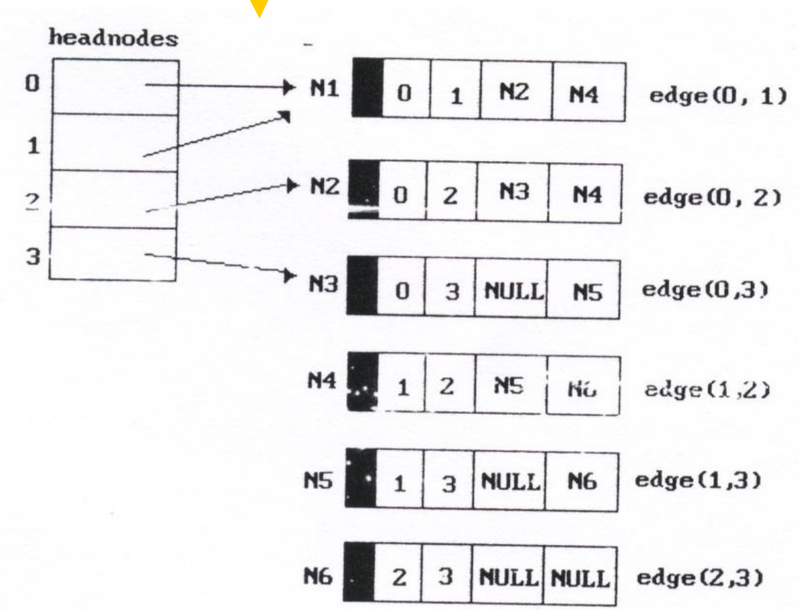
Graph Create()	::=	**return** an empty graph.
Graph InsertVertex(*graph*, *v*)	::=	**return** a graph with *v* inserted. *v* has no incident edges.
Graph InsertEdge(*graph*, v_1, v_2)	::=	**return** a graph with a new edge between v_1 and v_2.
Graph DeleteVertex(*graph*, *v*)	::=	**return** a graph in 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. Leave the incident nodes in the graph.
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*.

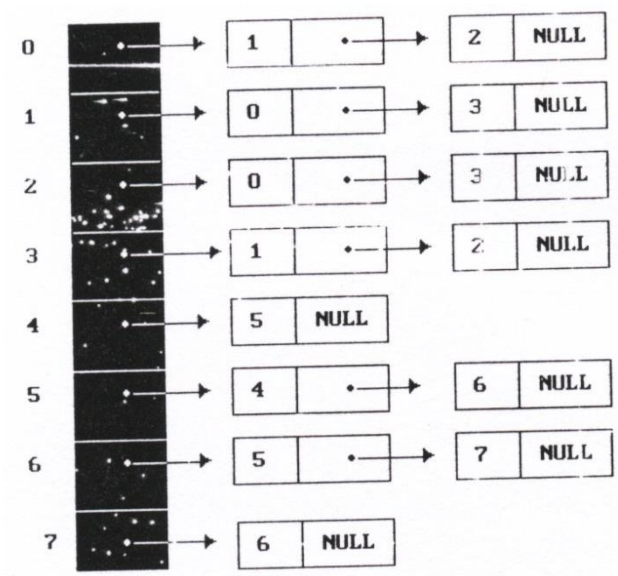
Abstract data type *Graph*

Graph Representations (1)

- Adjacency Matrix
- Adjacency Lists
- Adjacency Multilists

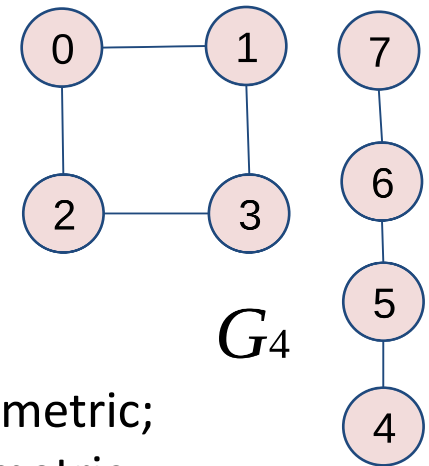
$$
\begin{array}{c} \\ 0\\1\\2\\3\\4\\5\\6\\7 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\left[\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}\right]
\end{array}
$$



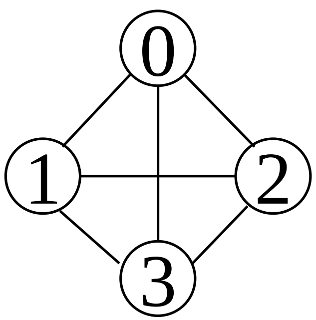


Graph Representations (2)

- Adjacency Matrix
  - Let G = (V,E) be a graph with n vertices.
  - The adjacency matrix of G is a two-dimensional n x n array, say adj_mat
  - If the edge (v_i, v_j) is(not) in E(G), adj_mat[i][j]=1(0)
  - The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

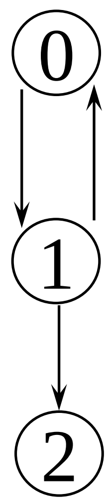


G_4



G_1

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



G_3

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

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

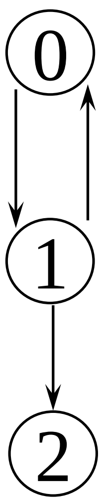
Graph Representations (3)

- Merits of Adjacency Matrix
  - For an undirected graph, the degree of any vertex, i, is its row sum:

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

$$\sum_{j=0}^{n-1} adj\_mat[i][j]$$

  - For a directed graph, the row sum is the out-degree, while the column sum is the in-degree.



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

  - The complexity of checking edge number or examining if G is connect
    - G is undirected: $O(n^2/2)$
    - G is directed: $O(n^2)$

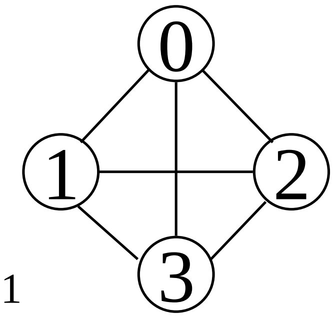
Graph Representations (4)

- Adjacency lists
  - There is one list for each vertex in G. The nodes in list i represent the vertices that are adjacent from vertex i
  - For an undirected graph with n vertices and e edges, this representation requires n head nodes and 2e list nodes
  - C declarations for adjacency lists

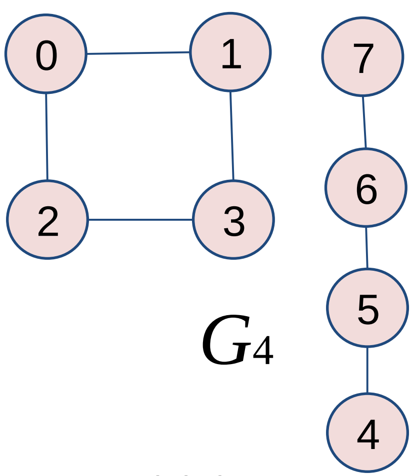
```
#define MAX_VERTICES 50 /*maximum number of vertices*/
typedef struct node *node_pointer;
typedef struct node {
        int vertex;
        struct node *link;
        };
node_pointer graph[MAX_VERTICES];
int n = 0; /* vertices currently in use */
```

Graph Representations (5)

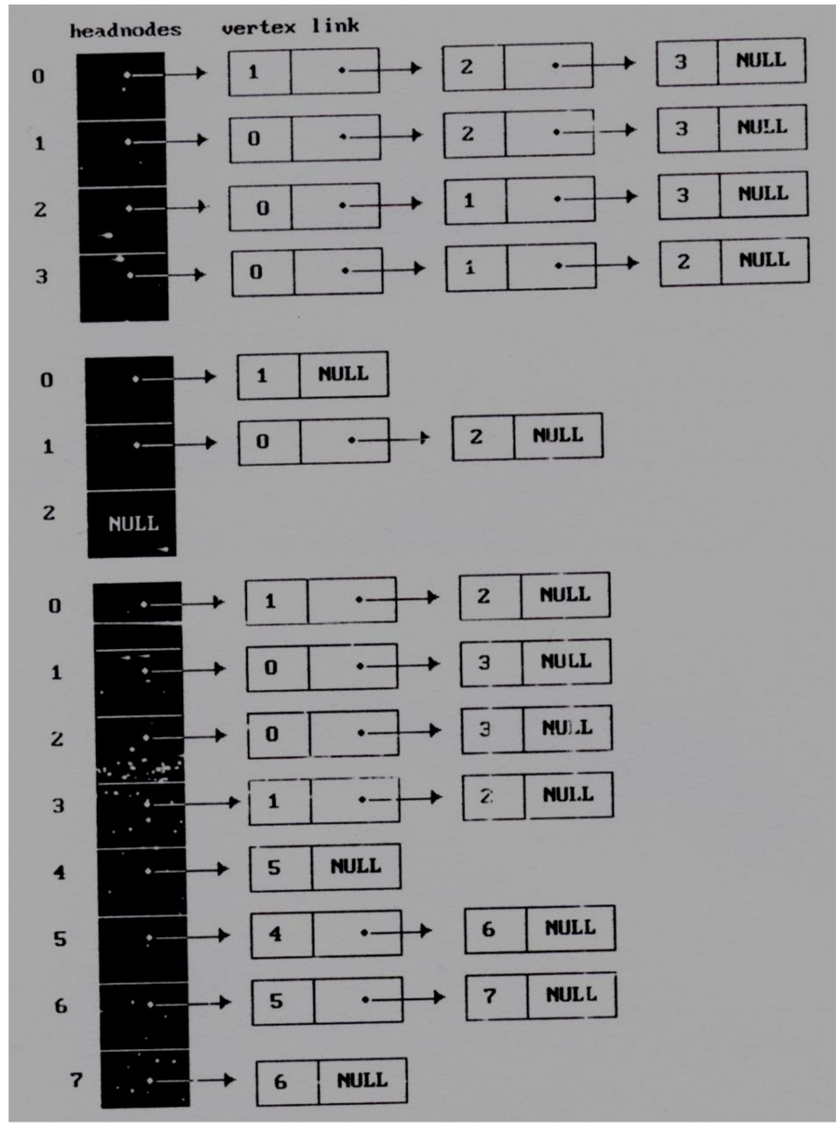
- Example



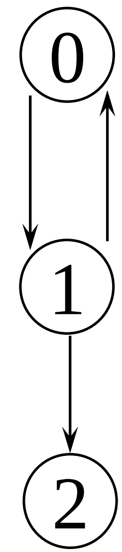
G_1



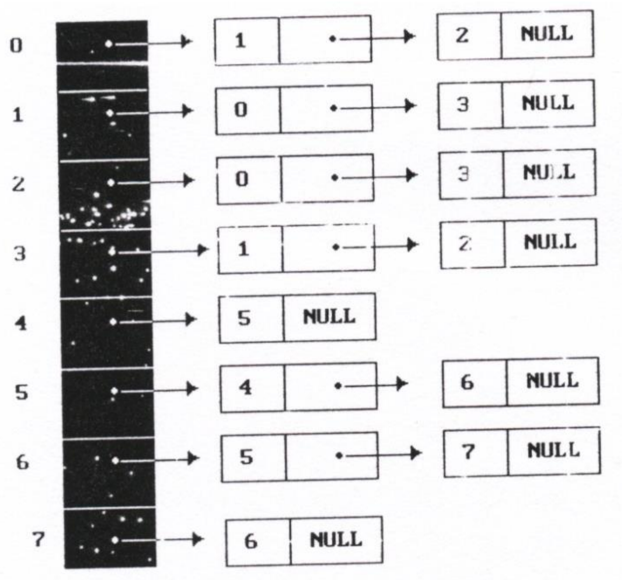
G_4

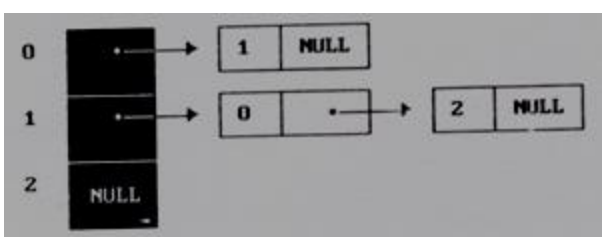


G_3



Graph Representations (6)

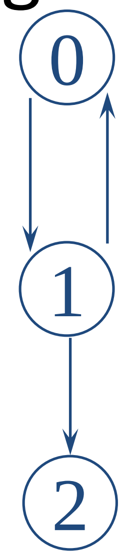


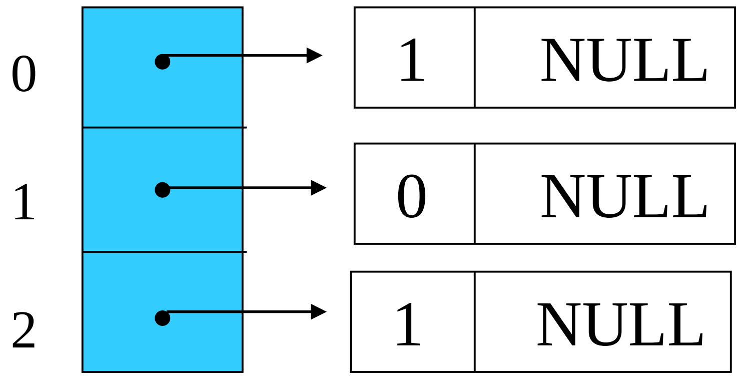


- Interesting Operations
  - degree of a vertex in an undirected graph
    - # of nodes in adjacency list
  - check # of edges in a graph
    - determined in O(n+e)
  - out-degree of a vertex in a directed graph
    - # of nodes in its adjacency list
  - in-degree of a vertex in a directed graph
   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

Graph Representations (7)

- Finding In-degree of Vertices





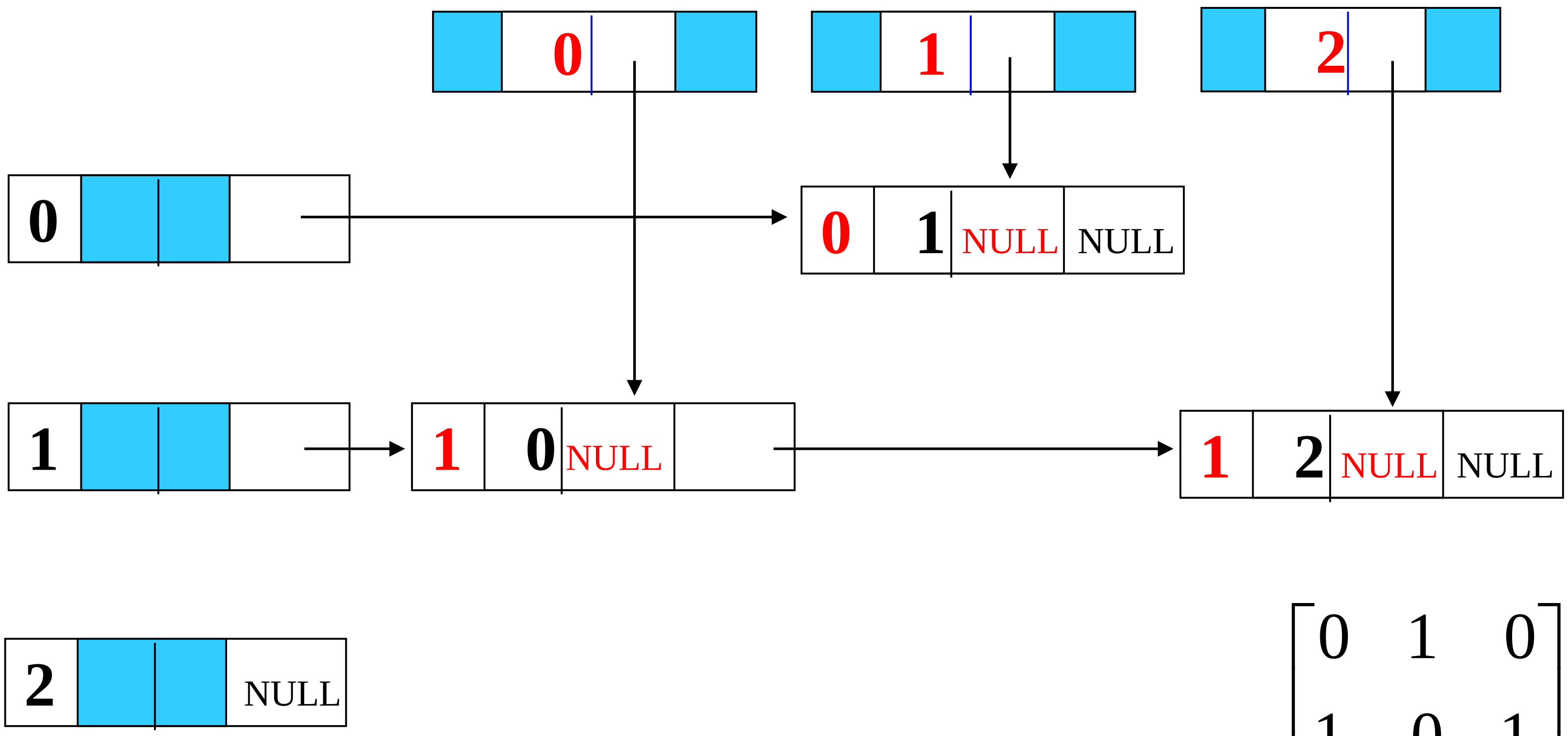
Inverse adjacency list of G_3

tail	head	column link for tail	row link for head

Alternate node structure of adjacency lists

Graph Representations (8)

- Example of Changing Node Structure



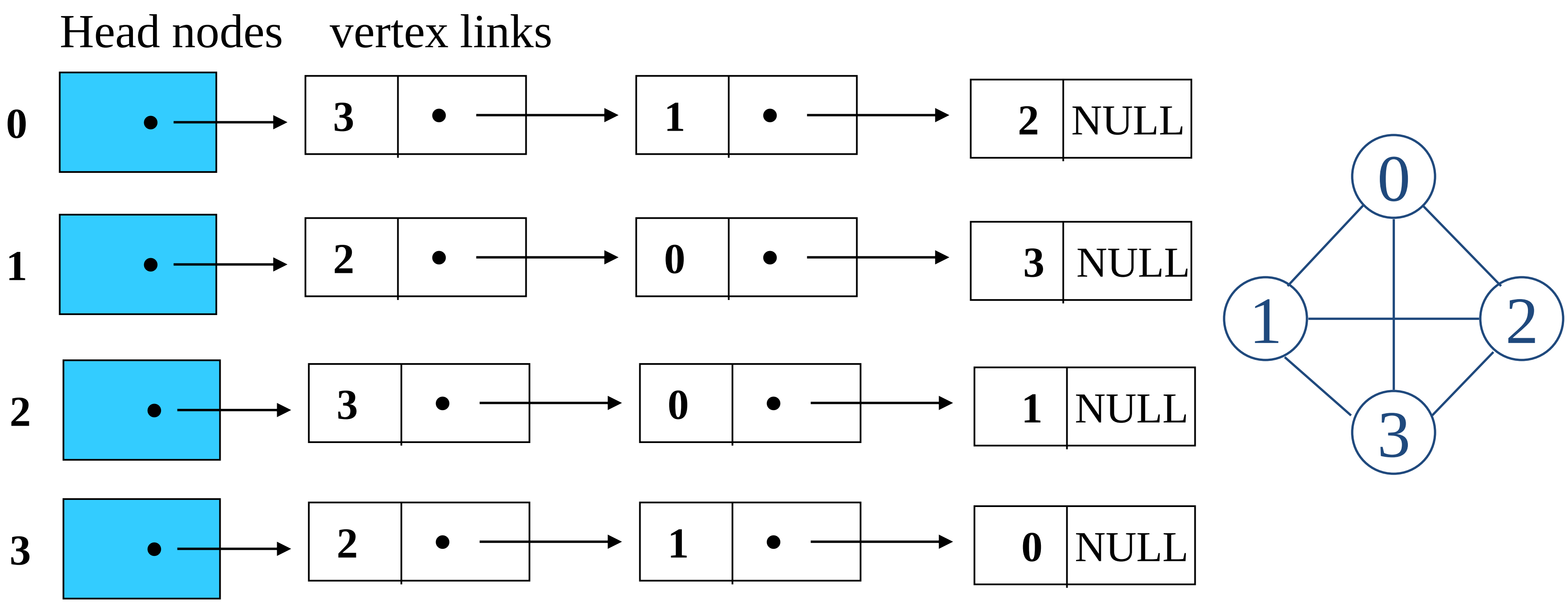
Orthogonal representation of G_3

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

Graph Representations (9)

- Vertices in Any Order

Order is of no significance



Graph Representations (10)

- Adjacency Multilists

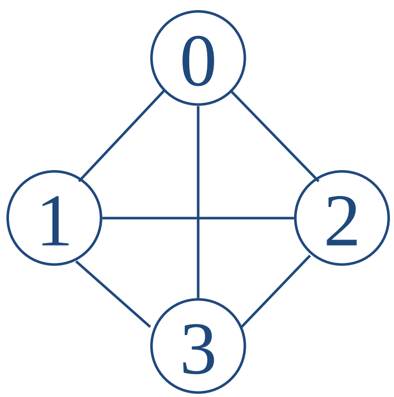
marked	vertex1	vertex2	path1	path2

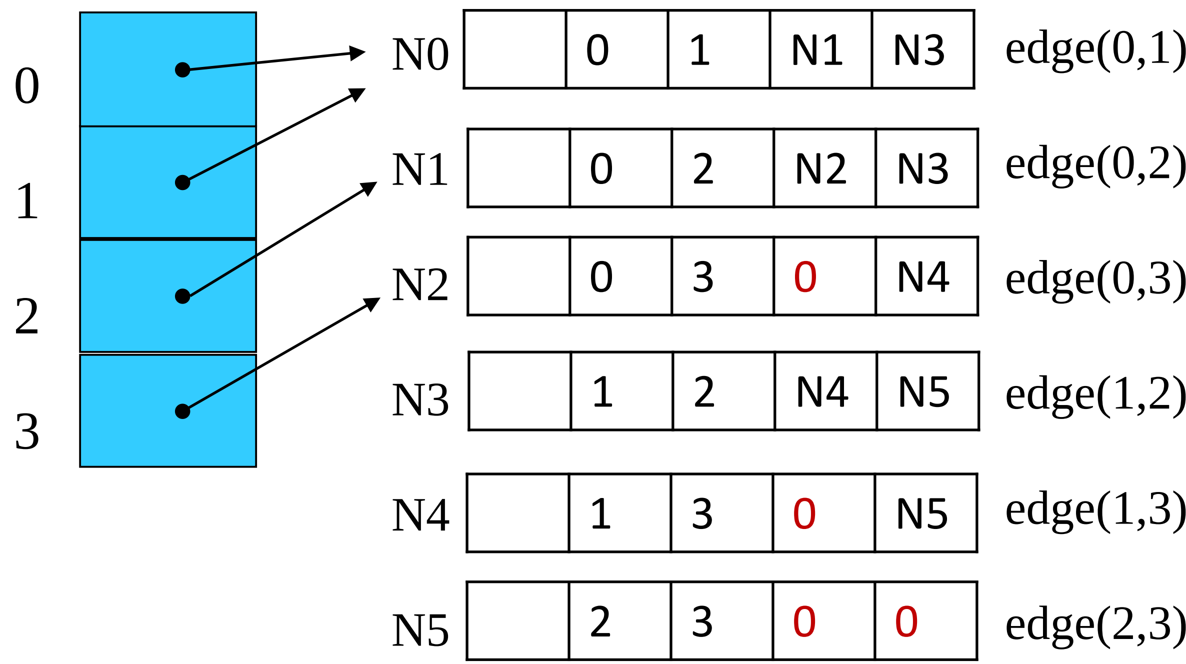
  - Lists in which nodes may be shared among several lists. (an edge is shared by two different paths)
  - There is exactly one node for each edge.
  - This node is on the adjacency list for each of the two vertices it is incident to

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

Graph Representations (11)

- Example for Adjacency Multlists





The lists are

vertex 0: N0 → N1 → N2
vertex 1: N0 → N3 → N4
vertex 2: N1 → N3 → N5
vertex 3: N2 → N4 → N5

Graph Representations (12)

- Weighted edges
  - The edges of a graph have weights assigned to them.
  - These weights may represent as
    - the distance from one vertex to another
    - cost of going from one vertex to an adjacent vertex.
  - adjacency matrix: adj_mat[i][j] would keep the weights.
  - adjacency lists: add a weight field to the node structure.
  - A graph with weighted edges is called a network

ELEMENTARY GRAPH OPERATIONS

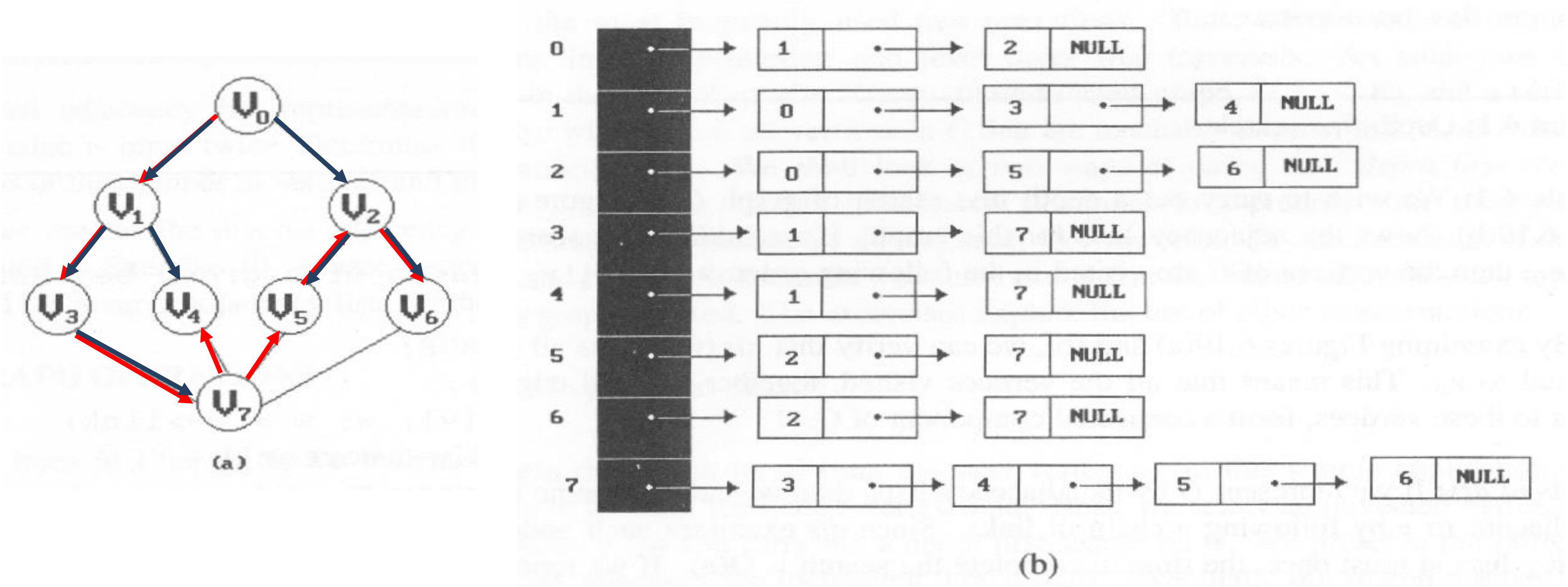
Graph Operations (1)

- Traversal
  Given G=(V, E) and vertex v, find all $w \in V$, such that w connects v
  - Depth First Search (DFS): preorder traversal
  - Breadth First Search (BFS): level order traversal
- Spanning Trees
- Biconnected Components

Graph Operations (2)

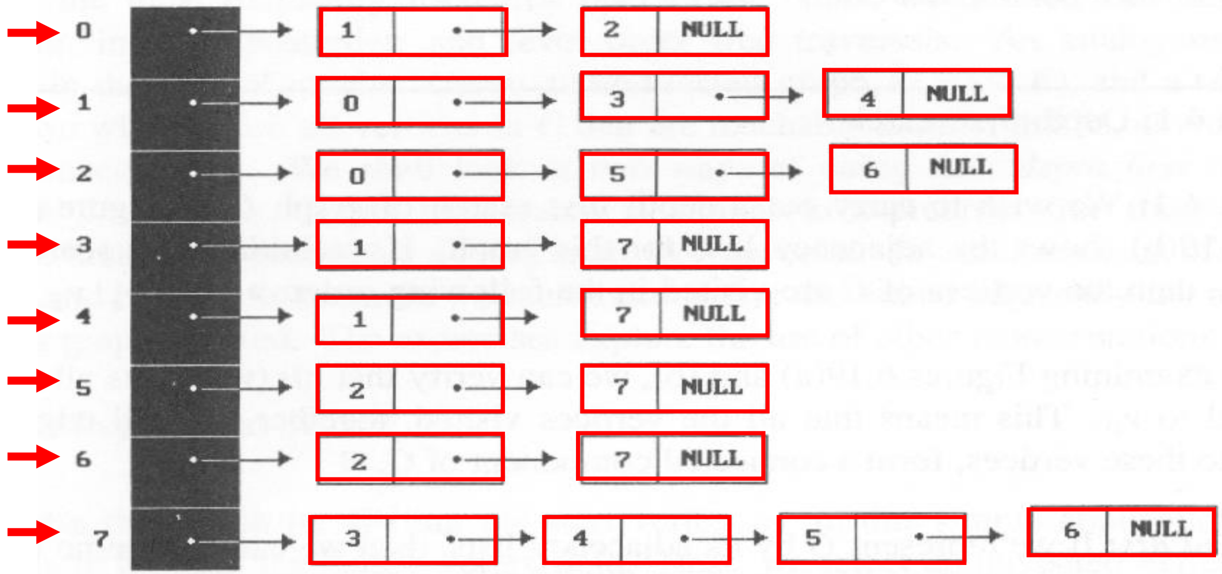
- Example for traversal
  (using Adjacency List representation of Graph)

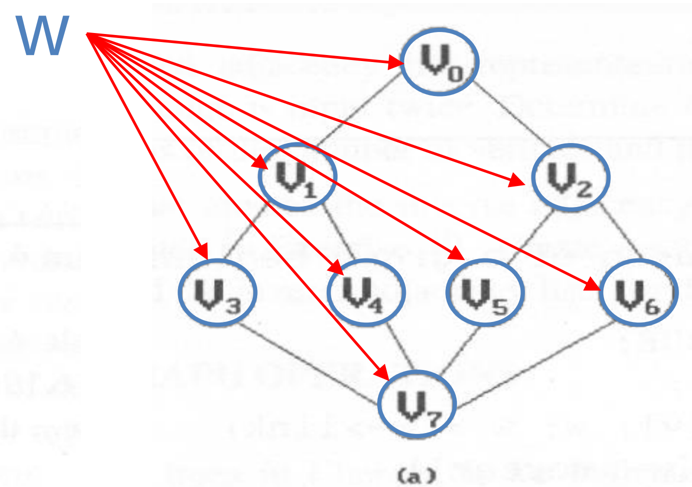
depth first search (DFS): $v_0, v_1, v_3, v_7, v_4, v_5, v_2, v_6$



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

- Depth First Search



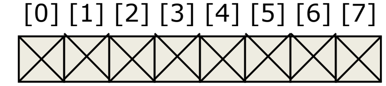


Data structure
adjacency list: O(e)
adjacency matrix: O(n^2)

```
void dfs(int v)
{
/* depth first search of a graph begi
   node_pointer w;
   visited[v] = TRUE;
   printf("%5d",v);
   for (w = graph[v]; w; w = w->link)
      if (!visited[w->vertex])
         dfs(w->vertex);
}
```

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

visited: [0] [1] [2] [3] [4] [5] [6] [7]



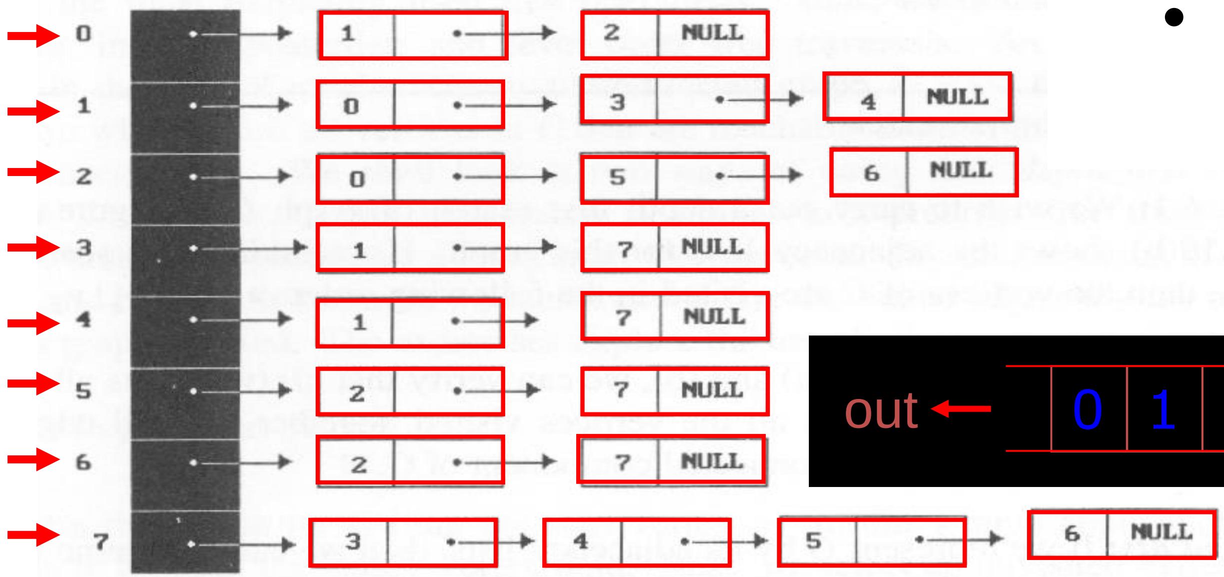
output: 0 1 3 7 4 5 2 6

Graph Operations (4)

- Breadth First Search
  - It needs a queue to implement breadth-first search
  - void bfs(int v): breadth first traversal of a graph
    - starting with node v the global array visited is initialized to 0
    - the queue operations are similar to those described in Chapter 4

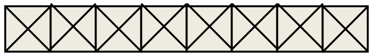
```
typedef struct queue *queue_pointer;
typedef struct queue {
        int vertex;
        queue_pointer link;
        };
void addq(queue_pointer *, queue_pointer *, int);
int deleteq(queue_pointer *);
```

• Breadth First Search : Example



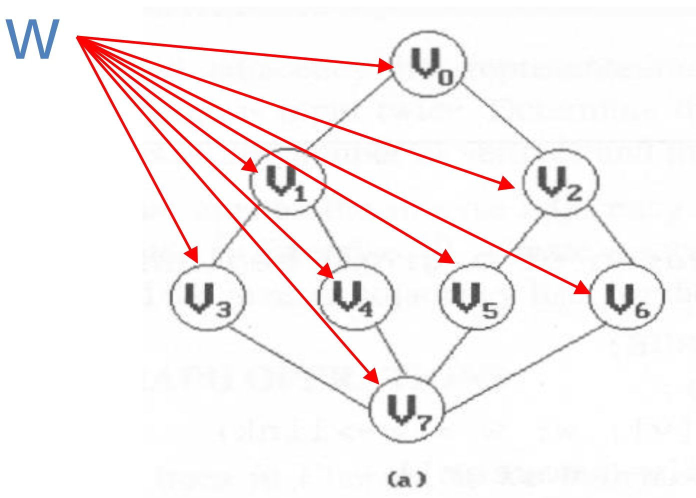
[0] [1] [2] [3] [4] [5] [6] [7]

visited:



output: 0 1 2 3 4 5 6 7

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




```
node_pointer w;
queue_pointer front,rear;
front = rear = NULL; /* initialize queue */
printf("%5d",v);
visited[v] = TRUE;
addq(&front, &rear, v);
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;
        }
}
```

Graph Operations (6)

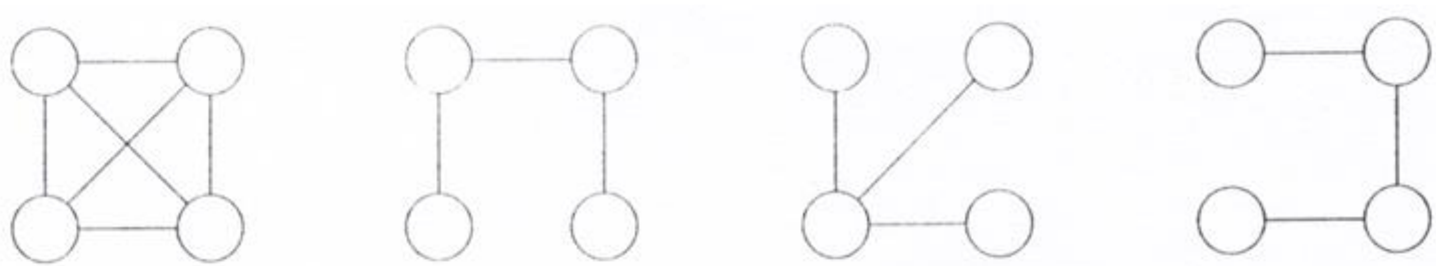
- Connected components
  - If G is an undirected graph, then one can determine whether or not it is connected:
    - simply making a call to either dfs or bfs
    - then determining if there is any unvisited vertex

```
void connected(void)
{
/* determine the connected components of a graph */
int i;
for (i = 0; i < n; i++)
   if(!visited[i]) {
      dfs(i);
      printf("\n");
   }
}
```

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

Graph Operations (7)

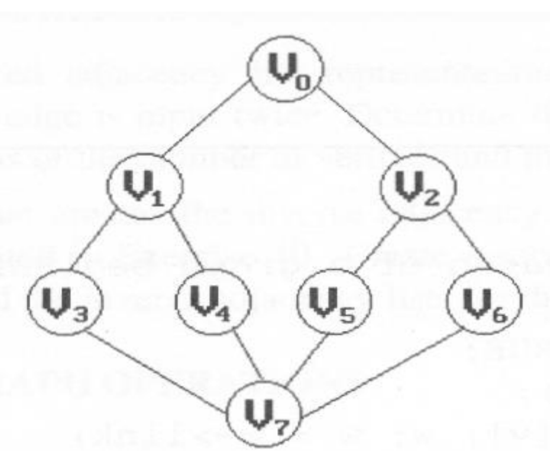
- Spanning trees
  - Definition: A tree T is said to be a spanning tree of a connected graph G if T is a subgraph of G and T contains all vertices of G.
  - E(G): T (tree edges) + N (nontree edges)
    - T: set of edges used during search
    - N: set of remaining edges



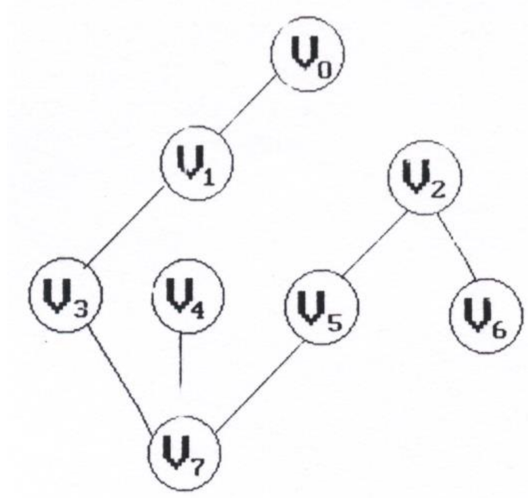
A complete graph and three of its spanning trees

Graph Operations (8)

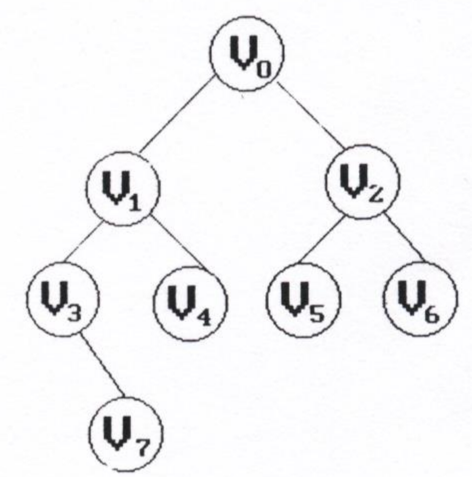
- When graph G is connected, a depth first or breadth first search starting at any vertex will visit all vertices in G
- We may use DFS or BFS to create a spanning tree
  - Depth first spanning tree when DFS is used
  - Breadth first spanning tree when BFS is used



(a)



(a) dfs(0) spanning tree



(b) bfs(0) spanning tree

Graph Operations (9)

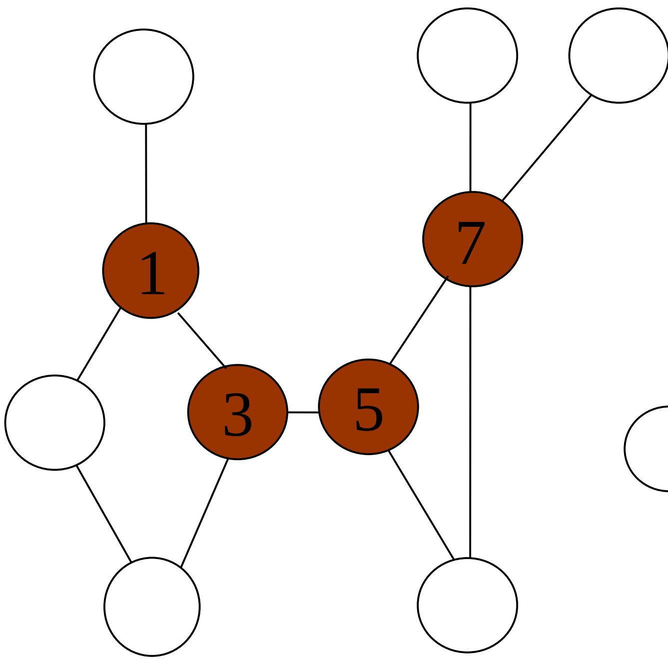
- Properties of spanning trees
  - If a nontree edge (v, w) is introduced into any spanning tree T, then a cycle is formed.
  - A spanning tree is a minimal subgraph, G’, of G such that V(G’) = V(G) and G’ is connected.
    - We define a minimal subgraph as one with the fewest number of edge
    - A spanning tree has n-1 edges

Graph Operations (10)

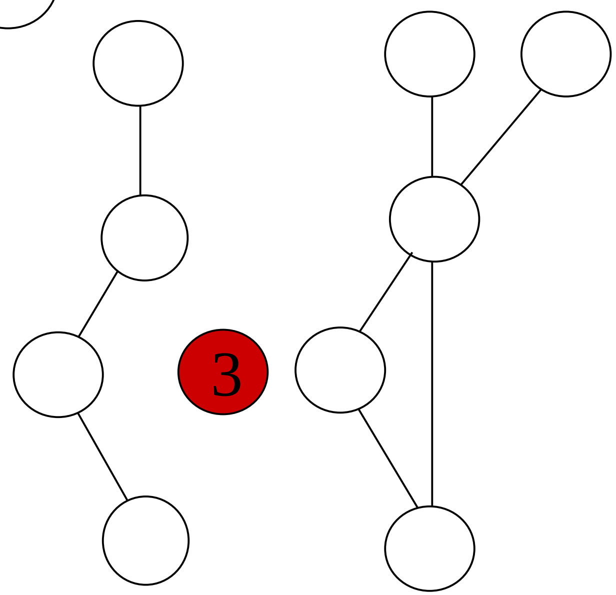
- Biconnected Graph & Articulation Points
  - Assumption: G is an undirected, connected graph
  - Definition: A vertex v of G is an articulation point iff the deletion of v, together with the deletion of all edges incident to v, leaves behind a graph that has at least two connected components
  - Definition: A biconnected graph is a connected graph that has no articulation points.
  - Definition: A biconnected component of a connected graph G is a maximal biconnected subgraph H of G.
    - By maximal, we mean that G contains no other subgraph that is both biconnected and properly contains H.

Graph Operations (11)

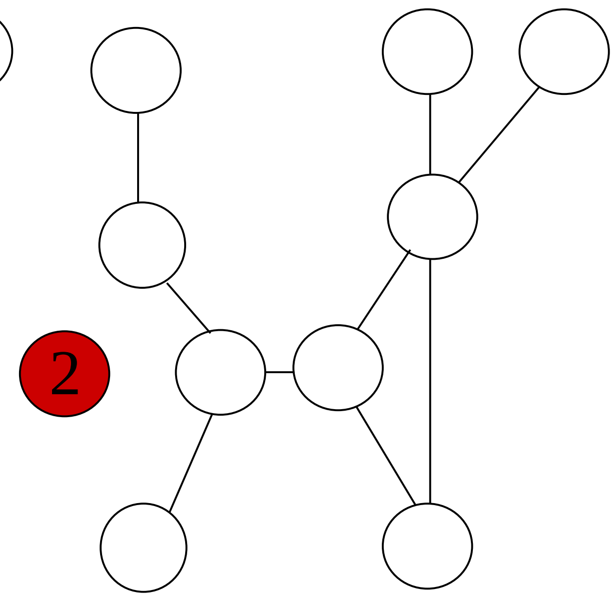
- Examples of Articulation Points (node 1, 3, 5, 7)



Connected graph



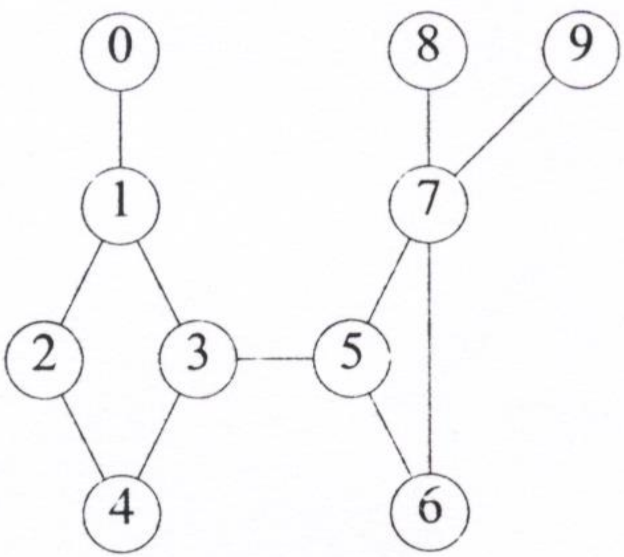
two connected components



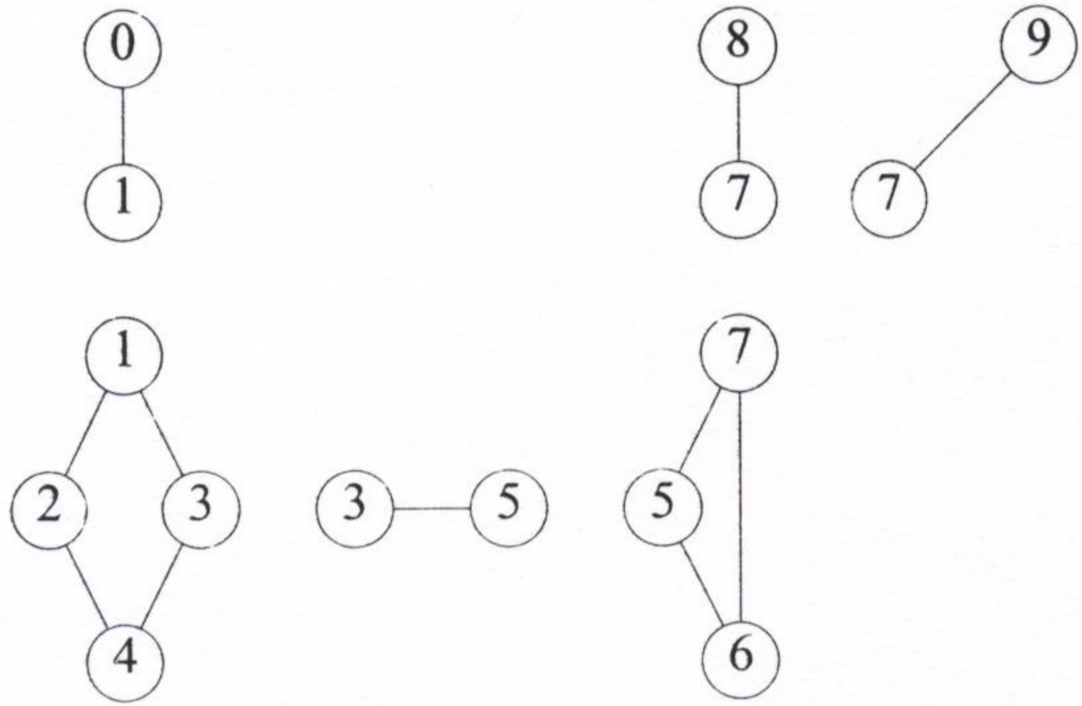
one connected graph

Graph Operations (12)

- Biconnected component
  a maximal connected subgraph H
  - no subgraph that is both biconnected and properly contains H



(a) Connected graph



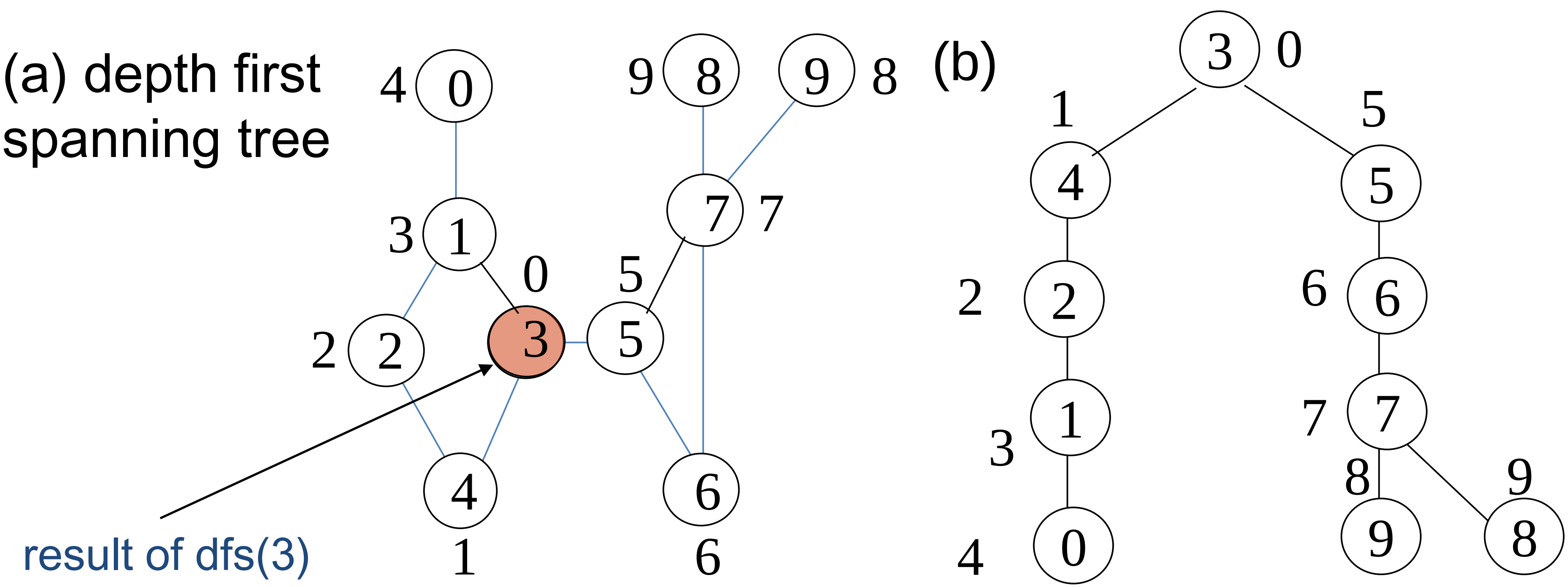
(b) Biconnected components

Graph Operations (13)

- Finding the biconnected components
  - By using depth first spanning tree of a connected undirected graph
  - The depth first number (*dfn*) outside the vertices in the figures gives the DFS visit sequence
  - If *u* is an ancestor of v then *dfn*(*u*) < *dfn*(*v*)

(a) depth first spanning tree

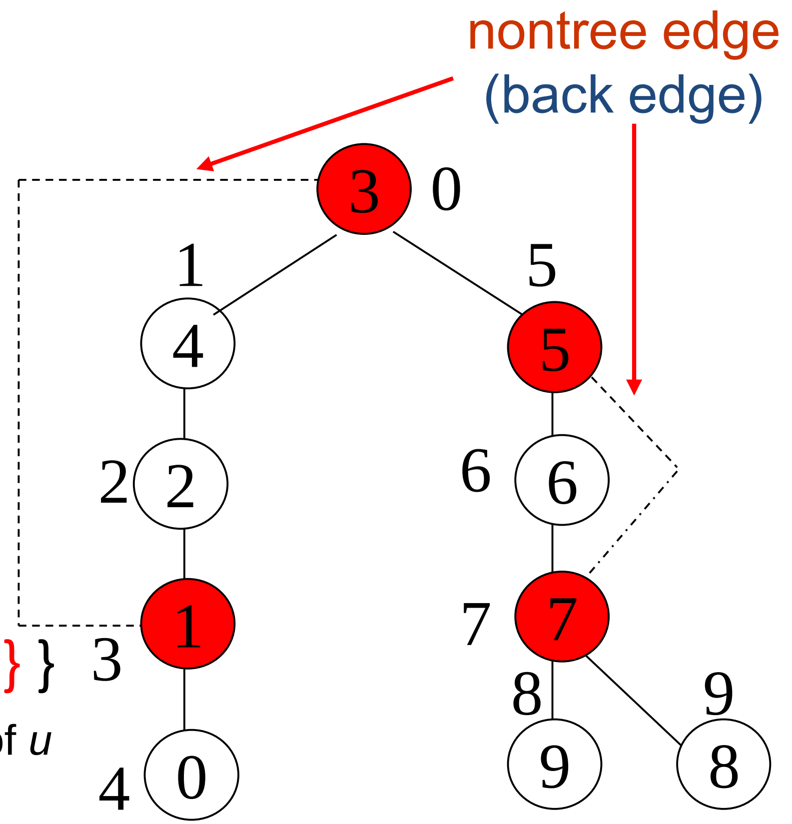
(b)



Graph Operations (14)

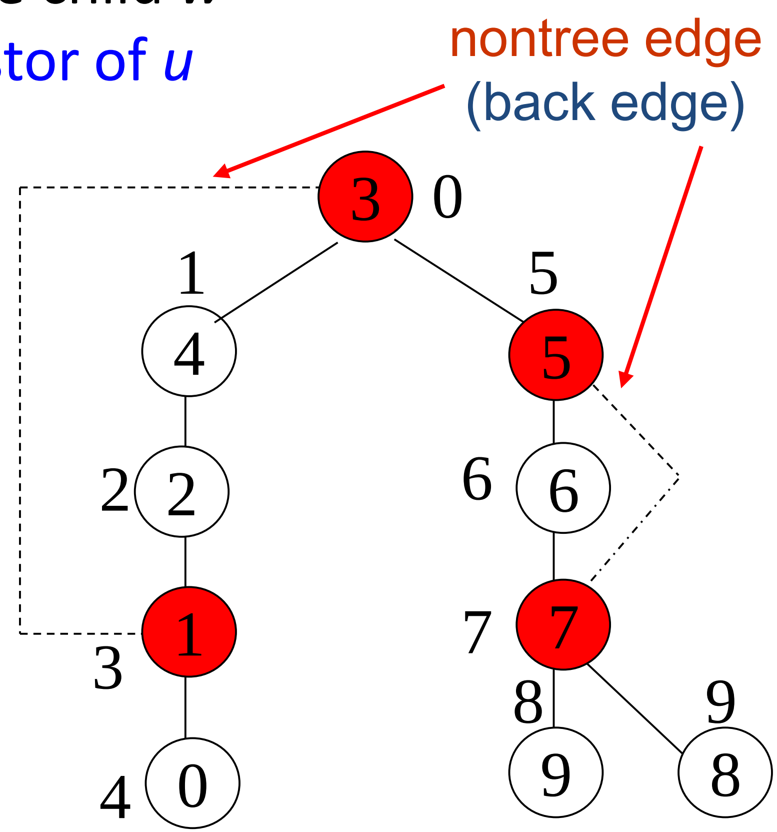
- *dfn* and *low*
  - Define *low*(*u*): the lowest *dfn* that we can reach from *u* using a path of descendants followed by at most one back edge
  - *low*(*u*) = min{ *dfn*(*u*), min{ *low*(*w*) | *w* is a child of *u* }, min{ *dfn*(*w*) | (*u*, *w*) is a back edge } }

    here, *w* is an ancestor of *u*



Graph Operations (15)

- Finding an articulation point in a graph:
  Any vertex *u* is an articulation point *iff*
  - *u* is the root of the spanning tree and has two or more children
  - *u* is not the root and has at least one child *w* such that we cannot reach an ancestor of *u* using a path that consists of only
    (1) *w*
    (2) descendants of *w*
    (3) a single back edge
    thus, $low(w) \geq dfn(u)$



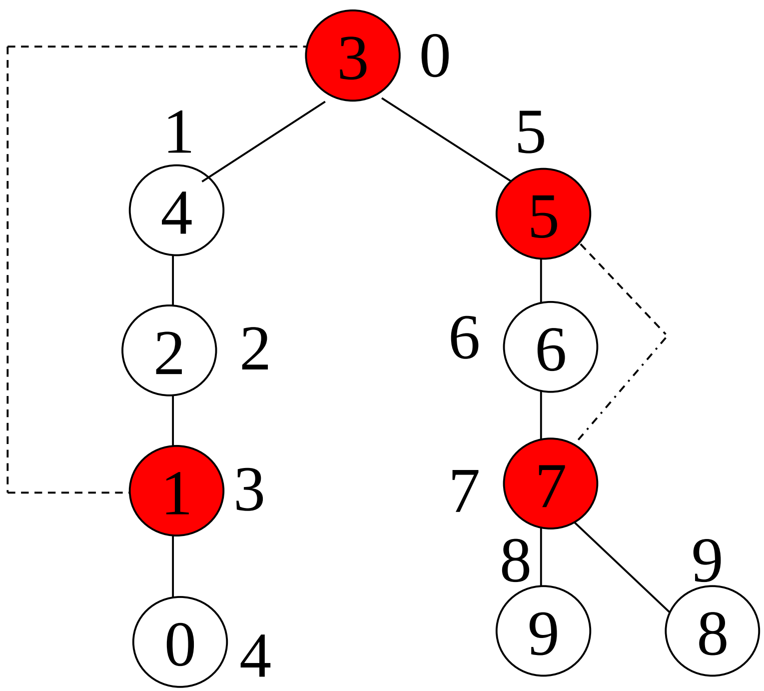
Vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8

Graph Operations (16)

- *dfn* and *low* values for *dfs* spanning tree with root = 3
  - $low(u) = \min\{\, dfn(u), \min\{\, low(w) \mid w \text{ is a child of } u \,\}, \min\{\, dfn(w) \mid (u, w) \text{ is a back edge} \,\}\,\}$

here, *w* is an ancestor of *u*

vertex	dfn	low	child	low_child	low:dfn	arti. point
0	4	4 (4,n,n)	null	null	null:4	
1	3	0 (3,4,0)	0	4	4 ≥ 3	•
2	2	0 (2,0,n)	1	0	0 < 2	
3	0	0 (0,0,n)	4,5	0,5	0,5 ≥ 0	•
4	1	0 (1,0,n)	2	0	0 < 1	
5	5	5 (5,5,n)	6	5	5 ≥ 5	•
6	6	5 (6,5,n)	7	5	5 < 6	
7	7	5 (7,8,5)	8,9	9,8	9,8 ≥ 7	•
8	9	9 (9,n,n)	null	null	null:9	
9	8	8 (8,n,n)	null	null	null:8	



Graph Operations (17)

- Determining *dfn* and *low*
  - we can easily modify *dfs*( ) to compute *dfn* and *low* for each vertex of a connected undirected graph

```
#define MIN2 (x, y) ((x) < (y) ? (x) : (y))
short int dfn[MAX_VERTICES];
short int low[MAX_VERTICES];
int num;
```

```
void init(void)
{
  int i;
  for (i = 0; i < n; i++) {
    visited[i] = FALSE;
    dfn[i] = low[i] = -1;
  }
  num = 0;
}
```

Program 6.5: Initialization of *dfn* and *low*

- Determining *dfn* and *low* (cont'd)
  - we invoke the function with call *dfnlow*(*x*, -1), where *x* is the starting vertex (root) for the depth first search

```
void dfnlow(int u, int v)
{
   node_pointer ptr;
   int w;
   dfn[u] = low[u] = num++;
   for (ptr = graph[u]; ptr; ptr = ptr->link) {
     w = ptr->vertex;
     if (dfn[w] < 0) { /* w is an unvisited vertex */
       dfnlow(w,u);
       low[u] = MIN2(low[u],low[w]);
     }
     else if (w != v)
       low[u] = MIN2(low[u],dfn[w]);
   }
}
```

v is the parent of u (if any)

w is a child of *u*

dfn(*u*)

(*u*, *w*) is a back edge

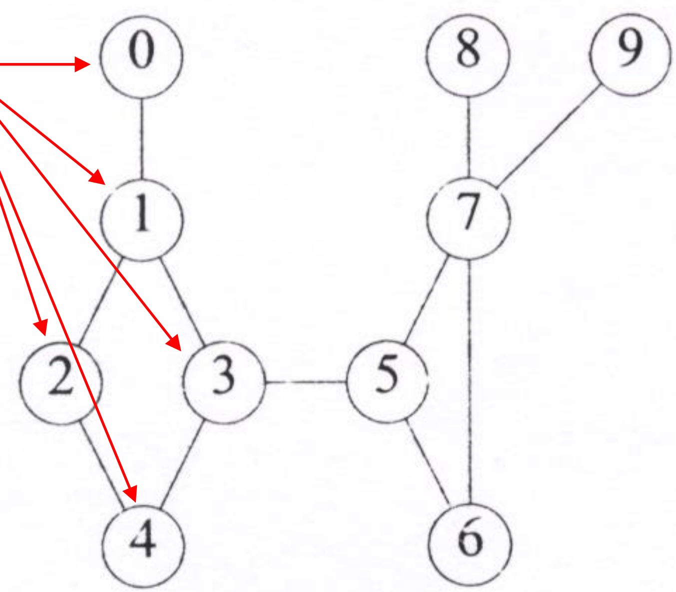
u: 0

v: -1

w: 0

num: 4

ptr



	0	1	2	3	4	5	6	7	8	9
dfn:	-4	-3	-2	-0	-1	-1	-1	-1	-1	-1
low:	-4	-0	-2	-0	-1	-1	-1	-1	-1	-1

Graph Operations (19)

- Partition the edges of the connected graph into their biconnected components
  - In *dfnlow*, we know that *low*[*w*] has been computed following the return from the function call *dfnlow*(*w*, *u*)
  - If *low*[*w*] $\geq$ *dfn*[*u*], then we have identified a new biconnected component
  - We can output all edges in a biconnected component if we use a stack to save the edges when we first encounter them
  - The function *bicon* (Program 6.6) contains the code modified from *dfnlow*, and the same initialization is used

- Find Biconnected components

	0	1	2	3	4	5	6	7	8	9
dfn:	4	3	2	0	1	5	6	7	9	8
low:	4	0	0	0	0	5	5	5	9	8

output: <1, 0><1, 3><2, 1><4, 2><3, 4>

<7, 9>

<7, 8>

<7, 5><6, 7>

<5, 6>

<3, 5>

u:

v: -1

w:

num: 10

7	6	5	3
5	7	6	5

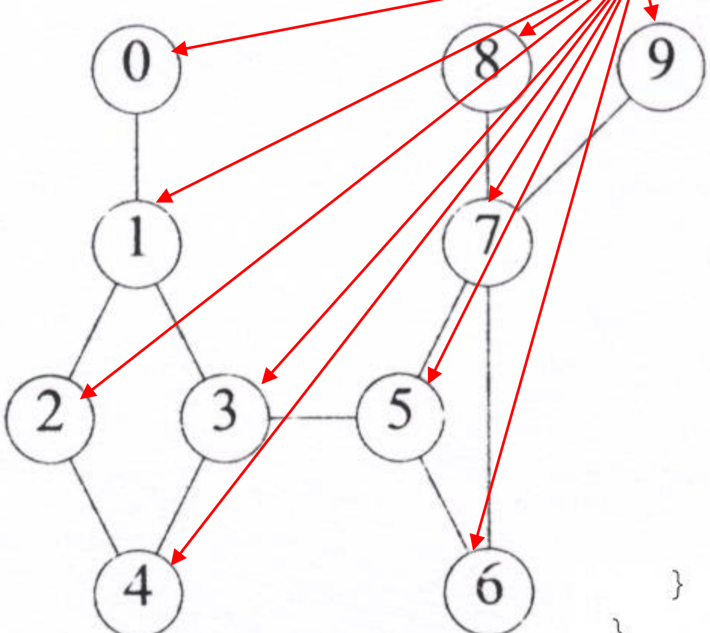
```
void bicon(int u, int v)
{
   node_pointer ptr;
   int w,x,y;
   dfn[u] = low[u] = num++;
   for (ptr = graph[u]; ptr; ptr = ptr->link) {
     w = ptr->vertex;
     if (v != w && dfn[w] < dfn[u])
       add(&top,u,w); /* add edge to stack */
     if (dfn[w] <0) { /* w has not been visited */
       bicon(w,u);
       low[u] = MIN2(low[u],low[w]);
       if (low[w] >= dfn[u]) {
         printf("New biconnected component: ");
         do { /* delete edge from stack */
           delete(&top, &x, &y);
           printf(" <%d,%d>",x,y);
         } while (!((x == u) && (y == w)));
         printf("\n");
       }
     }
     else if (w != v) low[u] = MIN2(low[u],dfn[w]);
   }
}
```

back edge or not yet visited

w is a child of *u*

(*u*, *w*) is a back edge

ptr



MINIMUM COST SPANNING TREES

Minimum Cost Spanning Trees (1)

- Introduction
  - The cost of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in the spanning tree
  - A minimum-cost spanning tree is a spanning tree of least cost
  - Three different algorithms can be used to obtain a minimum cost spanning tree
    - Kruskal's algorithm
    - Prim's algorithm
    - Sollin's algorithm
  - All the three use a design strategy called the greedy method

Minimum Cost Spanning Trees (2)

- Greedy Strategy
  - Construct an optimal solution in stages
  - At each stage, we make the best decision (selection) possible at this time.
    - using least-cost criterion for constructing minimum-cost spanning trees
  - Make sure that the decision will result in a feasible solution
  - A feasible solution works within the constraints specified by the problem
- Our solution must satisfy the following constrains
  - Must use only edges within the graph.
  - Must use exactly n - 1 edges.
  - May not use edges that produce a cycle

Minimum Cost Spanning Trees (3)

- Kruskal's Algorithm
  - Build a minimum cost spanning tree T by adding edges to T one at a time
  - Select the edges for inclusion in T in non-decreasing order of the cost
  - An edge is added to T if it does not form a cycle
  - Since G is connected and has $n > 0$ vertices, exactly n-1 edges will be selected
  - Time complexity: O(e log e)
  - Theorem
    - Let G be an undirected connected graph. Kruskal's algorithm generates a minimum cost spanning tree

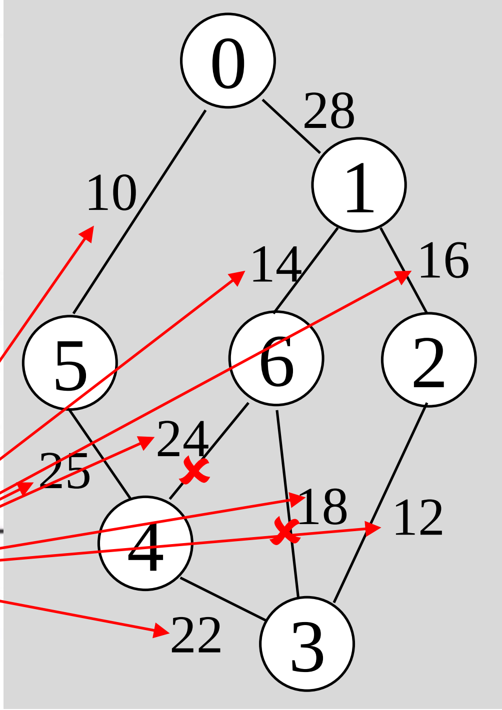
Minimum Cost Spanning Trees (4)

- Kruskal's Algorithm (cont'd)

```
T = {};
while (T contains less than n-1 edges && E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
}
if (T contains fewer than n-1 edges)
    printf("No spanning tree\n");
```

Program 6.7: Kruskal's algorithm

choose



Minimum Cost Spanning Trees (5)

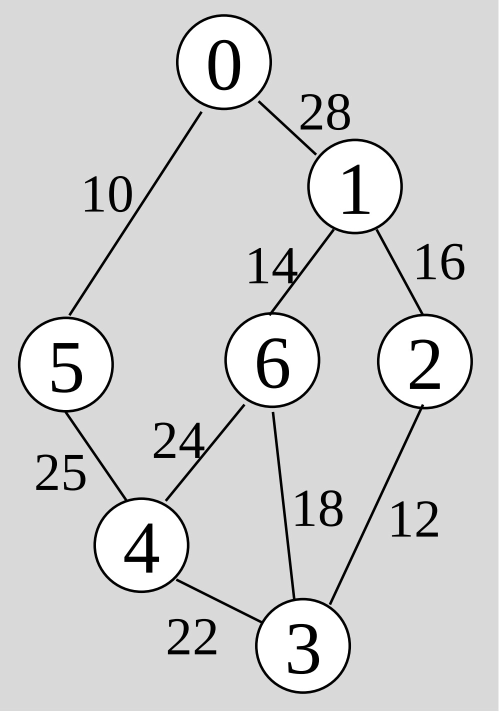
- Prim's Algorithm
  - Build a minimum cost spanning tree T by adding edges to T one at a time
  - At each stage, add a least cost edge to T such that the set of selected edges is still a tree
  - Repeat the edge addition step until T contains n-1 edges

Minimum Cost Spanning Trees (6)

- Prim's Algorithm (cont'd)

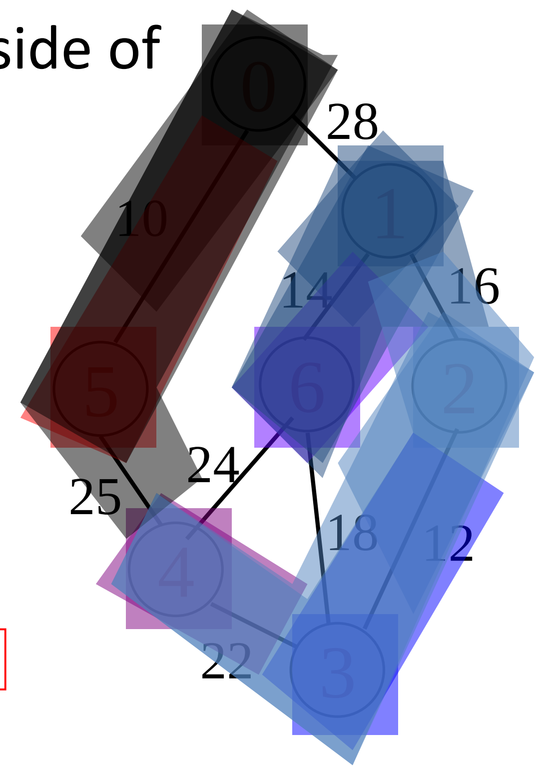
```
T = {};
TV = {0}; /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
   let (u, v) be a least cost edge such that u ∈ TV and
   v ∉ TV;
   if (there is no such edge)
      break;
   add v to TV;
   add (u, v) to T;
}
if (T contains fewer than n-1 edges)
   printf("No spanning tree\n");
```

Program 6.8: Prim's algorithm



Minimum Cost Spanning Trees (7)

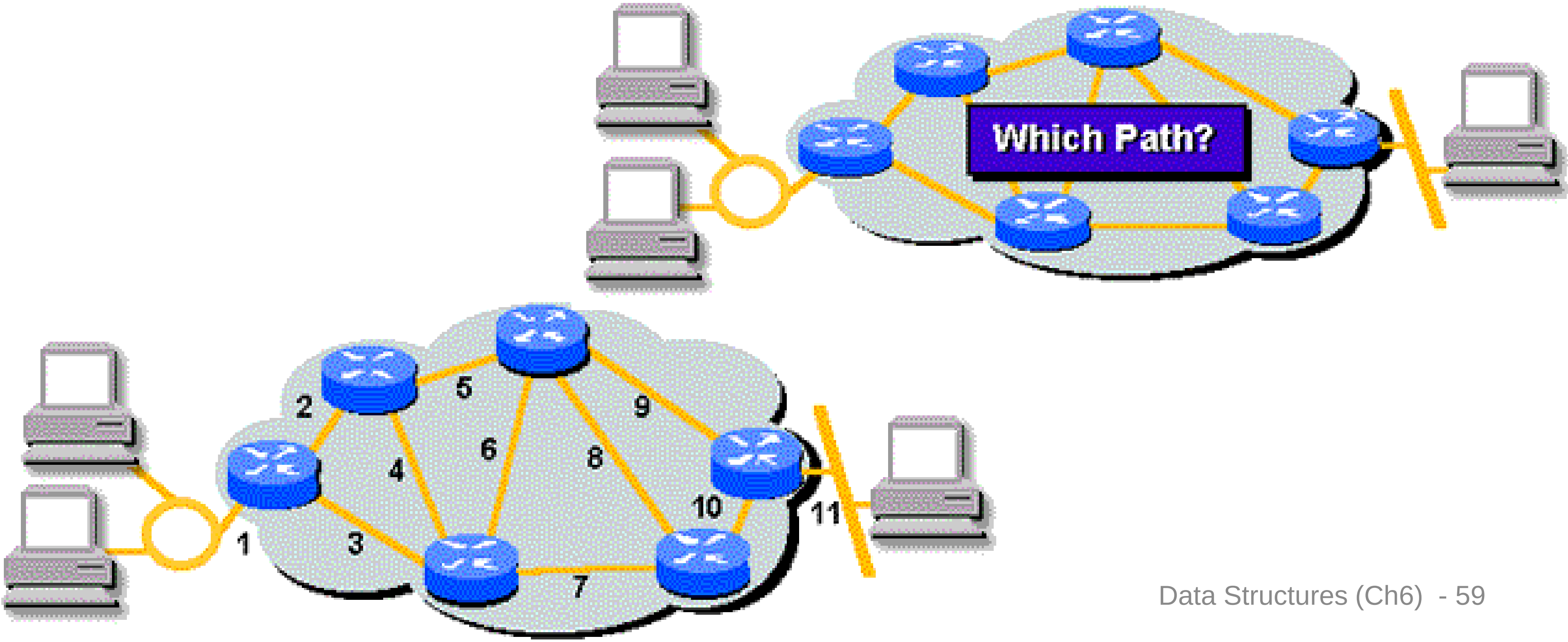
- Sollin's Algorithm
  - Selects several edges for inclusion in *T* at each stage.
  - At the start of a stage, the selected edges forms a spanning forest.
  - During a stage we select a minimum cost edge that has exactly one end in the tree and the other outside of tree (i.e., in the forest).
  - Repeat until only one tree at the end of a stage or no edges remain for selection.
    - Stage 1: (0, 5), (1, 6), (2, 3), (3, 2), (4, 3), (5, 0), (6, 1) ⇨ {(0, 5)}, {(1, 6)}, {(2, 3), (4, 3)}
    - Stage 2: {(0, 5), (5, 4)}, {(1, 6), (1, 2)}, {(2, 3), (4, 3), (1, 2)}
    - Result: {(0, 5), (5, 4), (1, 6), (1, 2), (2, 3), (4, 3)}



SHORTEST PATHS

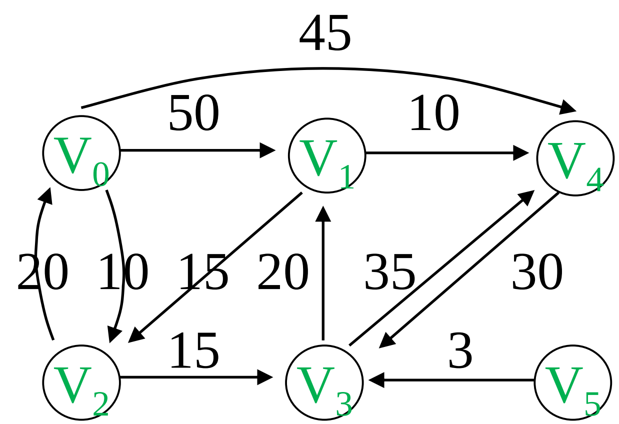
Shortest Paths (1)

- In this section, we shall study the path problems such like
  - Is there a path from city A to city B?
  - If there is more than one path from A to B, which path is the shortest?



Shortest Paths (2)

- Single source/All destinations: nonnegative edge cost
  - Problem: given a directed graph G = (V, E), a length function length(i, j), length(i, j) $\geq$ 0, for the edges of G, and a source vertex v
  - Need to solve: determine a shortest path from v to each of the remaining vertices of G
  - Let S denote the set of vertices
  - dist[w]: the length of shortest path starting from v, going through only the vertices that are in S, ending at w



path	length
1) v0 v2	10
2) v0 v2 v3	25
3) v0 v2 v3 v1	45
4) v0 v4	45

Shortest Paths (3)

- Dijkastra's Algorithm
  - Find the min cost of the path from a given source node to every other node
  - Given: the cost $e(v_i, v_j)$ of all edges; v_0 is the source node; $e(v_i, v_j) = \infty$, if v_i and v_j are not adjacent

  $S \leftarrow \{v_0\}$;

  $dist[v_0] \leftarrow 0$;

  for each v in V - $\{v_0\}$ do $dist[v] \leftarrow e(v_0, v)$;

  while $S \neq V$ do

    choose a vertex w in V-S such that dist[w] is a minimum;

    add w to S;

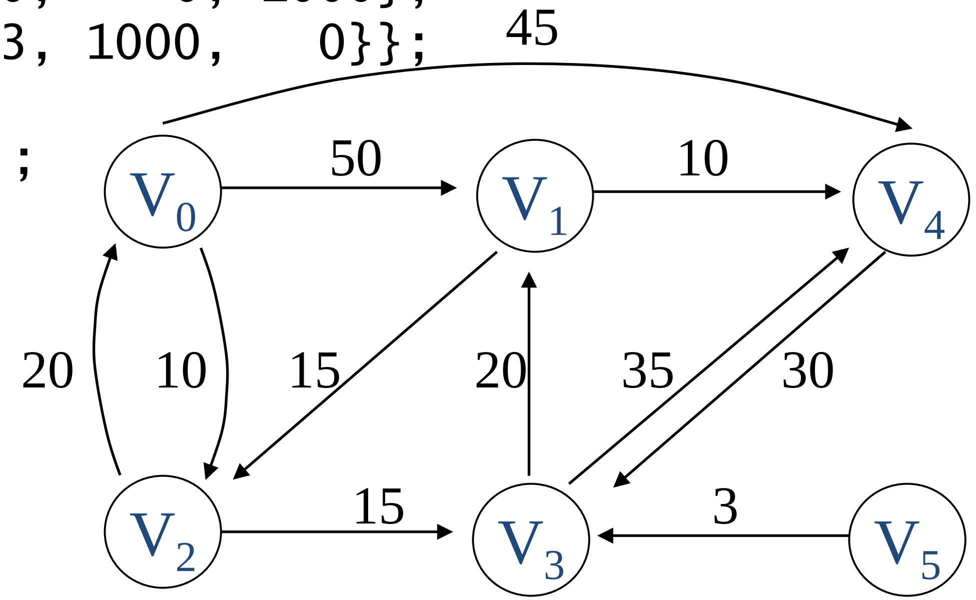
    for each v in V-S do

      $dist[v] \leftarrow min(dist[v], dist[w]+e(w, v))$;

Shortest Paths (4)

- Declarations for the Shortest Path Algorithm

```
#define MAX_VERTICES 6
int cost[ ][MAX_VERTICES]=
{      {   0,   50,   10, 1000,   45, 1000},
       {1000,    0,   15, 1000,   10, 1000},
       {  20, 1000,    0,   15, 1000, 1000},
       {1000,   20, 1000,    0,   35, 1000},
       {1000, 1000,   30, 1000,    0, 1000},
       {1000, 1000, 1000,    3, 1000,    0}};
int distance[MAX_VERTICES];
short int found{MAX_VERTICES];
int n = MAX_VERTICES;
```



Shortest Paths (5)

- Choosing the least cost edge

```
int choose(int distance[], int n, short int found[])
{
/* find the smallest distance not yet checked */
   int i, min, minpos;
   min = INT_MAX;
   minpos = -1;
   for (i = 0; i < n; i++)
      if (distance[i] < min && !found[i]) {
         min = distance[i];
         minpos = i;
      }
   return minpos;
}
```

Program 6.11: Choosing the least cost edge

• Single Source Shortest Paths Program (v=0)

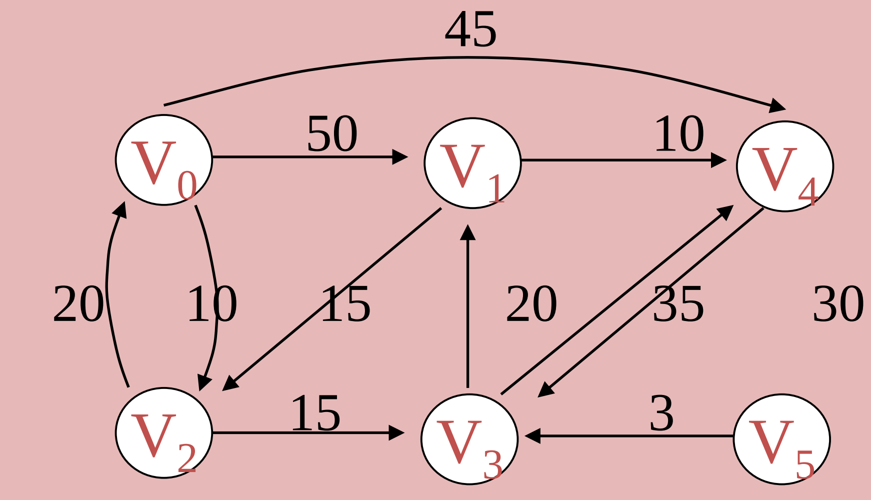
```
void shortestpath(int v, int cost[][MAX_VERTICES],
int distance[], int n, short int found[])
{
    int i,u,w;
    for (i = 0; i < n; i++) {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;
    distance[v] = 0;
    for (i = 0; i < n-2; i++) {
        u = choose(distance,n,found);
        found[u] = TRUE;
        for (w = 0; w < n; w++)
            if (!found[w])
                if (distance[u] + cost[u][w] < distance[w])
                    distance[w] = distance[u] + cost[u][w];
    }
}
```

Program 6.10: Single source shortest paths

found:

[0]	[1]	[2]	[3]	[4]	[5]
X	X	X	X	X	X

cost:



distance:

0	45	10	25	45	1000

i: [illegible]

u: [illegible]

w: [illegible]

Shortest Paths (7)

- All Pairs Shortest Paths
  - We could solve this problem using *shortestpath* with each of the vertices in V(G) as the source. ($O(n^3)$)
  - We can obtain a conceptually simpler algorithm that works correctly even if some edges in G have negative weights, require G has no cycles with a negative length (still $O(n^3)$)

Shortest Paths (8)

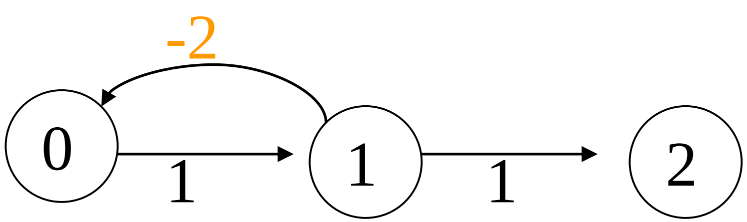
- Another solution
  - Use dynamic programming method
  - Represent the graph G by its cost adjacency matrix with cost[i][j]
    - If $i = j$, cost[i][j] = 0.
    - If <i, j> is not in G, cost[i][j] is set to some sufficiently large number
  - Let $A^k[i][j]$ be the cost of shortest path from i to j, using only those intermediate vertices with an index $\leq k$
    - The shortest path from i to j is $A^{n-1}[i][j]$ as no vertex in G has an index greater than $n - 1$

Shortest Paths (9)

- Algorithm concept
  - The basic idea in the all pairs algorithm is begin with the matrix A^{-1} and successively generated the matrices $A^{-1}, A^{0}, A^{1}, \ldots, A^{n-1}$
  - $A^{-1}[i][j]=\text{cost}[i][j]$
  - Calculate the A^0, A^1, A^2, ..., A^{n-1} from A^{-1} iteratively
  - $A^k[i][j]=\min\{A^{k-1}[i][j], A^{k-1}[i][k]+A^{k-1}[k][j]\}$, $k>=0$

Shortest Paths (10)

- Graph with Negative Cycle
  - The length of the shortest path from vertex 0 to vertex 2 in A^1 is $-\infty$



(a) Directed graph

$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

(b) A^{-1}

0, 1, 0, 1, 0, 1, …, 0, 1, 2

Shortest Paths (11)

- All pairs shortest paths program

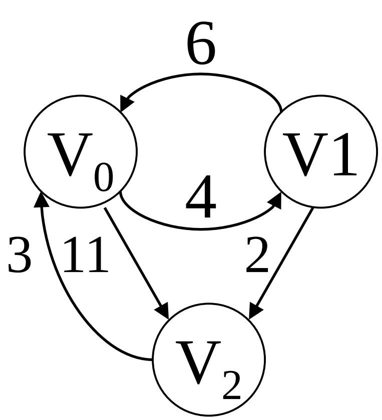
final distance:

0	4	11
6	0	2
3	7	0

cost:

0	4	11
6	0	2
3	1000	0

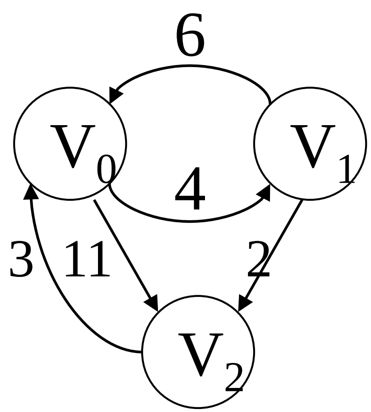
K 0
I 2
J 1




```
void allcosts(int cost[][MAX_VERTICES],
                int distance[][MAX_VERTICES], int n)
{
  int i,j,k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      distance[i][j] = cost[i][j];
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        if (distance[i][k] + distance[k][j] <
                              distance[i][j])
          distance[i][j] =
          distance[i][k] + distance[k][j];
}
```

Program 6.12: All pairs, shortest paths function

Shortest Paths (12)



$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

Cost Matrix for G

$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

A^{-1} $\quad$ A^0 $\quad$ A^1 $\quad$ A^2

TOPOLOGICAL SORTS

Topological Sorts (1)

- Activity on vertex (AOV) networks
  - All but the simplest of projects can be divided into several subprojects called activities
  - The successful completion of these activities results in the completion of the entire project
  - Example:
    A student working toward a degree in computer science must complete several courses successful

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

(a) Courses needed for a computer science degree at a hypothetical university

Topological Sorts (2)

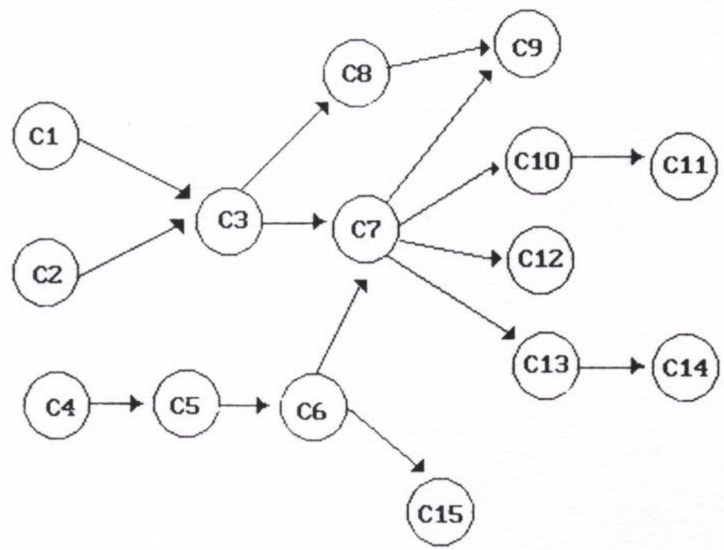
- Definition
  - Activity On Vertex (AOV) Network
    - a directed graph in which the vertices represent tasks or activities and the edges represent precedence relations between tasks
  - Predecessor (successor)
    - Vertex *i* is a predecessor of vertex *j iff* there is a directed path from *i* to *j*. *j* is a successor of *i*
  - Partial order
    - A precedence relation which is both transitive ($\forall$i, j, k, i · j & j · k $\rightarrow$ i · k) and irreflexive ($\forall$x, $\neg$x · x)
  - Acyclic graph
    - A directed graph with no directed cycles

Topological Sorts (3)

- Definition: Topological order
  - Linear ordering of vertices of a graph
  - $\forall i, j$ if i is a predecessor of j, then i precedes j in the linear ordering

C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9

C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C12, C13, C14

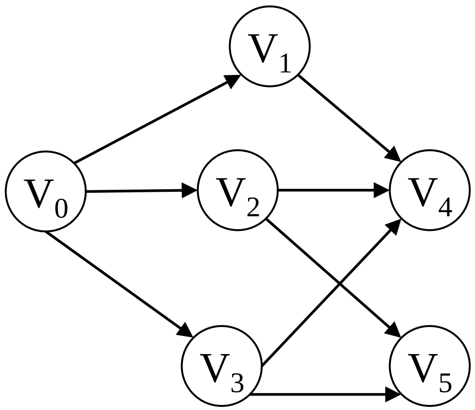


(b) AOV network representing courses as vertices and edges as prerequisites

Topological Sorts (4)

- Topological sort Algorithm

```
for (i = 0; i <n; i++) {
    if every vertex has a predecessor {
      fprintf (stderr, "Network has a cycle. \n " );
      exit(1);
    }
    pick a vertex v that has no predecessors;
    output v;
    delete v and all edges leading out of v from the network;
}
```



Topological order generated:

V_0 V_3 V_2 V_5 V_1 V_4

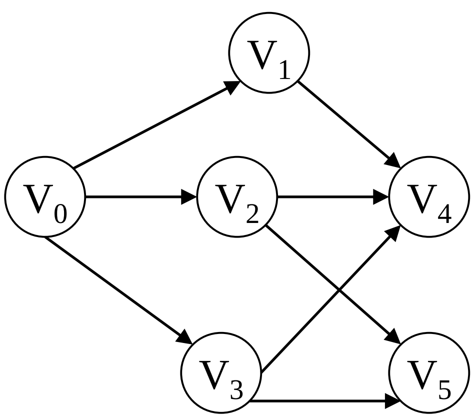
Topological Sorts (5)

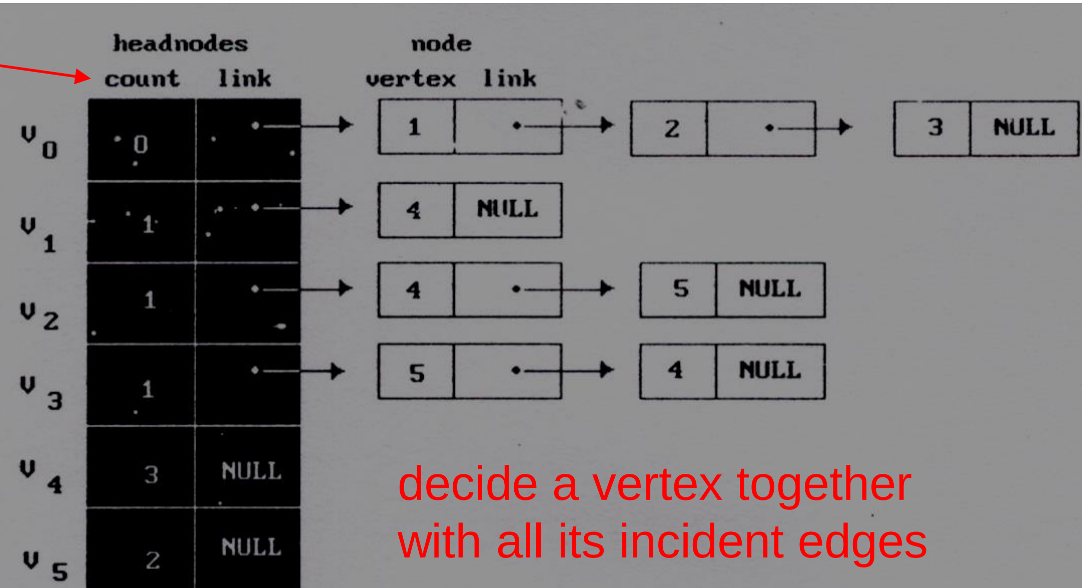
- Declarations used in *topsort*
- data representation

```
typedef struct node *node_pointer;
typedef struct node {
        int vertex;
        node_pointer link;
        };
typedef struct {
        int count;
        node_pointer link;
        } hdnodes;
hdnodes graph[MAX_VERTICES];
```

decide whether a vertex has any predecessors

the in-degree of that vertex

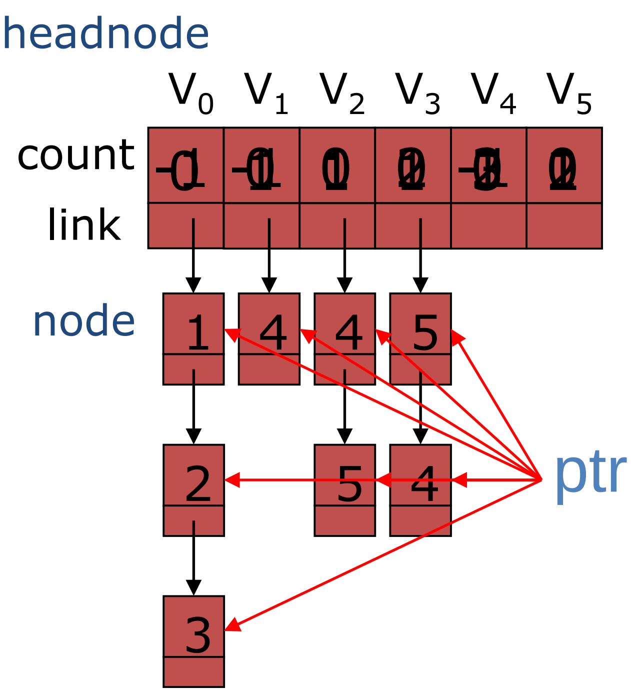




decide a vertex together with all its incident edges

Figure 6.40: Adjacency list representation of Figure 6.39(a)

- Topological sort Program

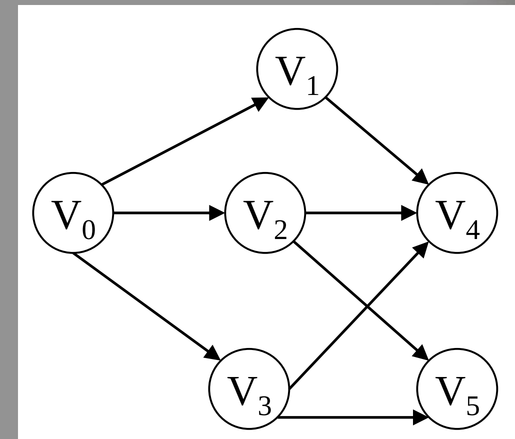


top: j: k:

output:

v0 v3 v2 v5 v1 v4

```
void topsort(hdnodes graph[], int n)
{
   int i,j,k,top;
   node_pointer ptr;
   /* create a stack of vertices with no predecessors */
   top = -1;
   for (i = 0; i < n; i++)
      if (!graph[i].count) {
         graph[i].count = top;
         top = i;
      }
   for (i = 0; i < n; i++)
      if (top == -1) {
         fprintf(stderr,"\nNetwork
        terminated. \n");
         exit(1);
      }
      else {
         j = top;    /* unstack a vertex */
         top = graph[top].count;
         printf("v%d, ",j);
         for (ptr = graph[j].link; ptr; ptr = ptr->link) {
         /* decrease the count of the successor vertices
         of j */
           k = ptr->vertex;
           graph[k].count--;
           if (!graph[k].count) {
           /* add vertex k to the stack */
              graph[k].count = top;
              top = k;
           }
         }
      }
}
```



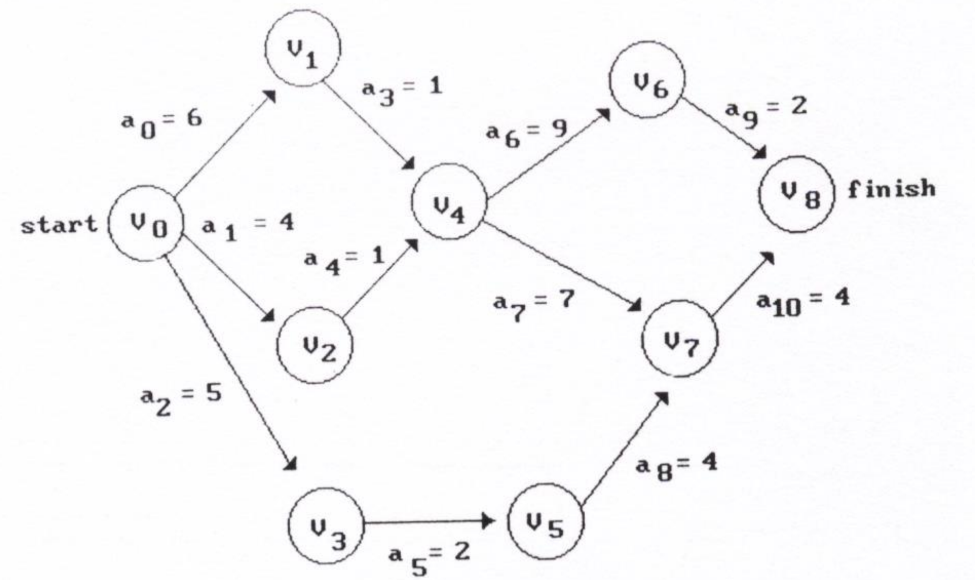
Time complexity: O(n+e)

Topological Sorts (7)

- Activity on Edge (AOE) Networks
  - AOE is related to the AOV network
- AOE networks have proved very useful for evaluating the performance of many types of projects
  - What is the least amount of time in which the project may be complete (assuming there are no cycle in the network)?
  - Which activities should be speeded to reduce project length?

Topological Sorts (8)

- An AOE network
  - directed edge: tasks or activities to be performed
  - Vertex: events which signal the completion of certain activities
  - Number: associated with each edge (activity) is the time required to perform the activity



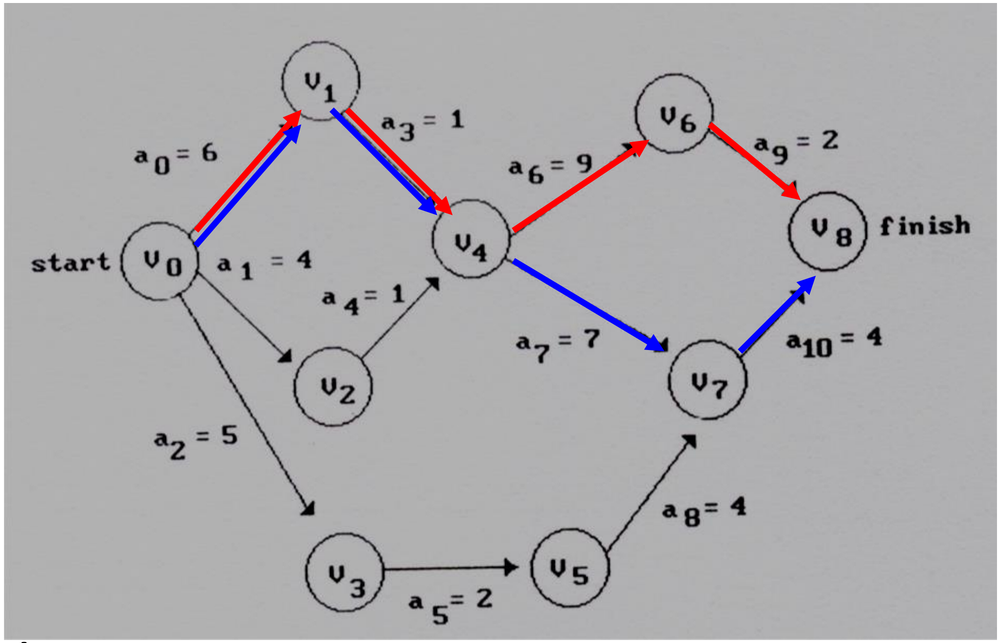
(a) AOE network. Activity graph of a hypothetical project

event	interpretation
v_0	start of project
v_1	completion of activity a_0
v_4	completion of activities a_3 and a_4
v_7	completion of activities a_7 and a_8
v_8	completion of project

(b) Interpretation of some of the events in the activity graph of (a)

Topological Sorts (9)

- Application of AOE Network
  - Evaluate performance
    - minimum amount of time
    - activity whose duration time should be shortened
  - Critical path
    - a path that has the longest length
    - minimum time required to complete the project



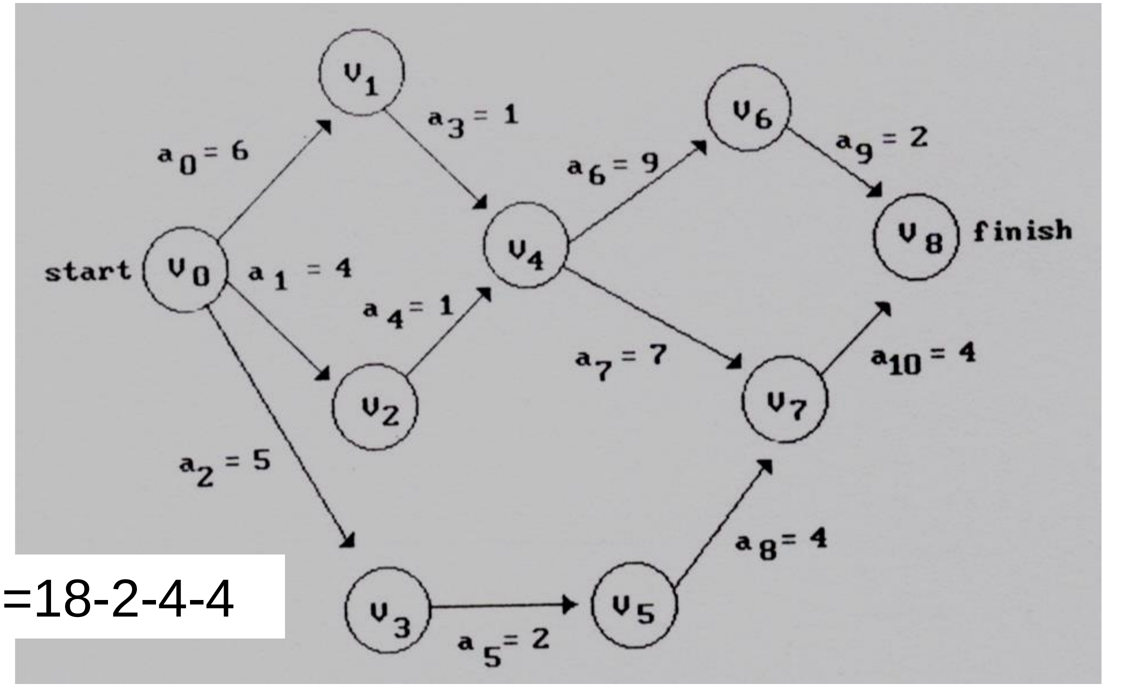
v_0, v_1, v_4, v_6, v_8 or v_0, v_1, v_4, v_7, v_8 (18)

Topological Sorts (10)

- Critical-path analysis
  - The purpose of critical-path analysis is to identify critical activities so that resource may be concentrated on these activities in an attempt to reduce a project finish time.
  - Critical-path analysis can also be carried out with AOV network
- Determine Critical Paths
  - Delete all noncritical activities
  - Generate all the paths from the start to finish vertex

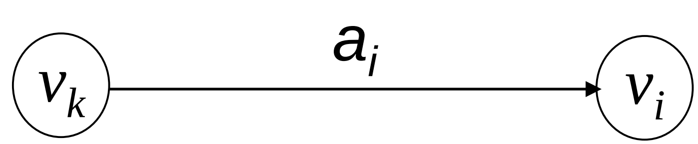
Topological Sorts (11)

- Various Factors
  - The *earliest time* of an event v_i
    - the length of the longest path from v_0 to v_i (Ex. 7 for v_4)
  - *early*(i): earliest event occurrence (Earliest activity) time of i
    - the earliest start time for all activities responded by edges leaving v_i
    - *early*(6) = *early*(7)=7
  - *late*(i): latest event occurrence (latest activity) time of i
    - the latest time the activity may start without increasing the project duration
    - *early*(5)=5, *late*(5)=8 =18-2-4-4
    - *early*(7)=7, *late*(7)=7



Topological Sorts (12)

- Various Factors (cont'd)
  - *late*(*i*)-*early*(*i*)
    - measure of how critical an activity is (ex. *late*(5)-*early*(5)=8-5=3)
  - Critical activity
    - an activity for which *early*(*i*)=*late*(*i*) (ex. *early*(7)=*late*(7))
  - *earliest*[*j*]:
    earliest event occurrence time for all event *j* in the network
  - *latest*[*j*]:
    latest event occurrence time for all event *j* in the network
  - If activity a_i is represented by edge <*k*, *i*>
    - *early*(*i*) = *earliest*[*k*]
    - *late*(*i*) = *latest*[*i*] - *duration of activity* a_i



  - We compute the times *earliest*[*j*] and *latest*[*j*] in two stages: a forward stage and a backward stage

Topological Sorts (13)

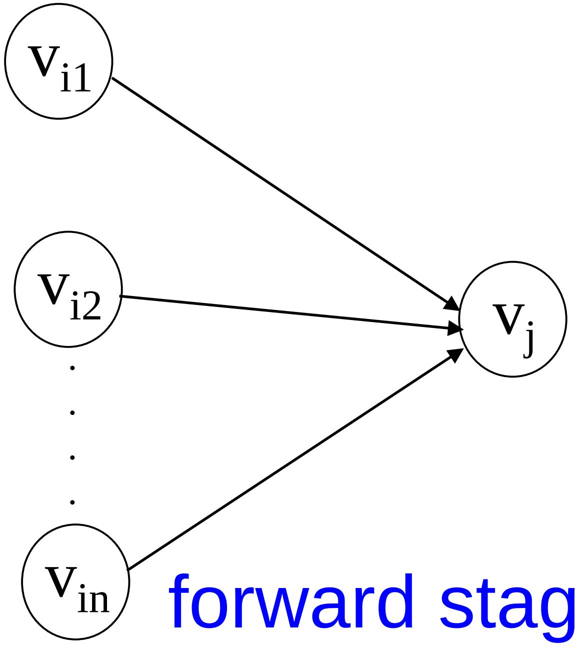
- Calculation of Earliest Times
  - During the forwarding stage, we start with *earliest*[0] = 0 and compute the remaining start times using the formula:

$$earliest[j] = \max_{i \in p(j)} \{ earliest[i] + \text{duration of} < i,\ j > \}$$

  Where *P*(*j*) is the set of immediate predecessors of *j*

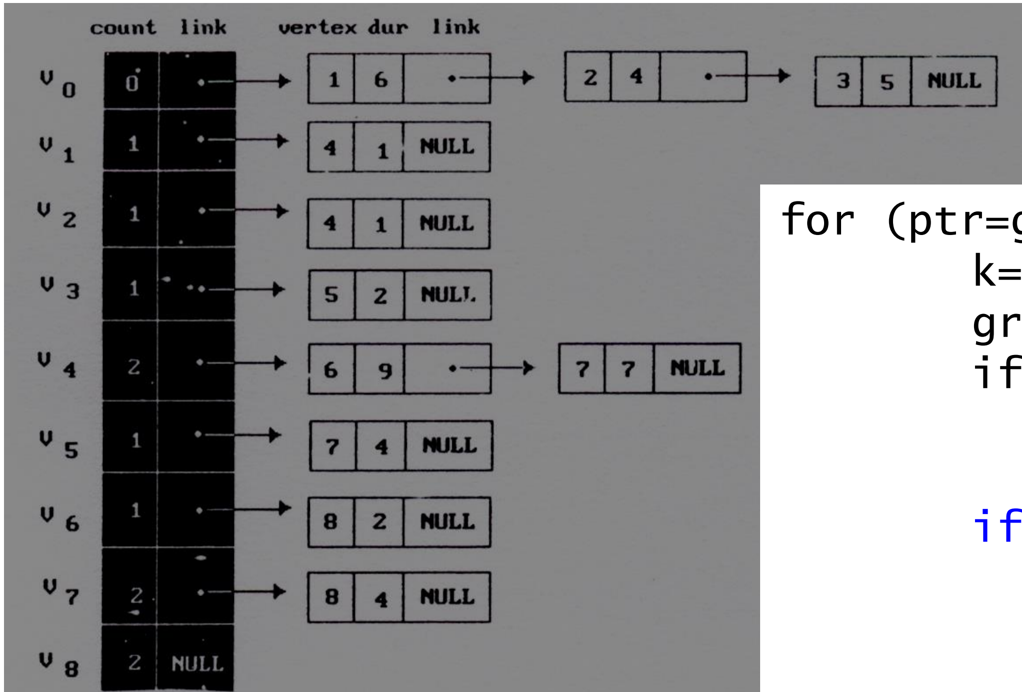
  - We can easily obtain an algorithm that does this by inserting the following statement at the end of the **else** clause in *topsort*

```
if (earliest[k] < earliest[j] + ptr->duration)
    earliest[k] = earliest[j] + ptr->duration;
```

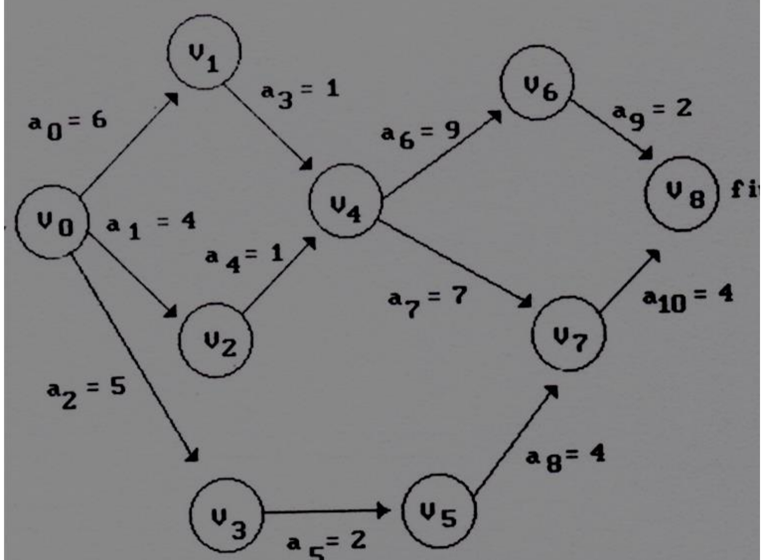


forward stage

- Calculation of Earliest Times (cont'd)




```
for (ptr=graph[j].link;ptr;ptr=ptr->link){
        k=ptr->vertex;
        graph[k].count--;
        if(!graph[k].count){
                graph[k].count=top;
                top=k;}
        if(earliest[k]<
                earliest[j]+ptr->duration)
                earliest[k] =
                earliest[j]+ptr->duration;
}
```



Earliest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	0	0	0	0	0	0	0	0	0	[0]
output v_0	0	6	4	5	0	0	0	0	0	[3, 2, 1]
output v_3	0	6	4	5	0	7	0	0	0	[5, 2, 1]
output v_5	0	6	4	5	0	7	0	11	0	[2, 1]
output v_2	0	6	4	5	5	7	0	11	0	[1]
output v_1	0	6	4	5	7	7	0	11	0	[4]
output v_4	0	6	4	5	7	7	16	14	0	[7, 6]
output v_7	0	6	4	5	7	7	16	14	16	[6]
output v_6	0	6	4	5	7	7	16	14	18	[8]
output v_8										

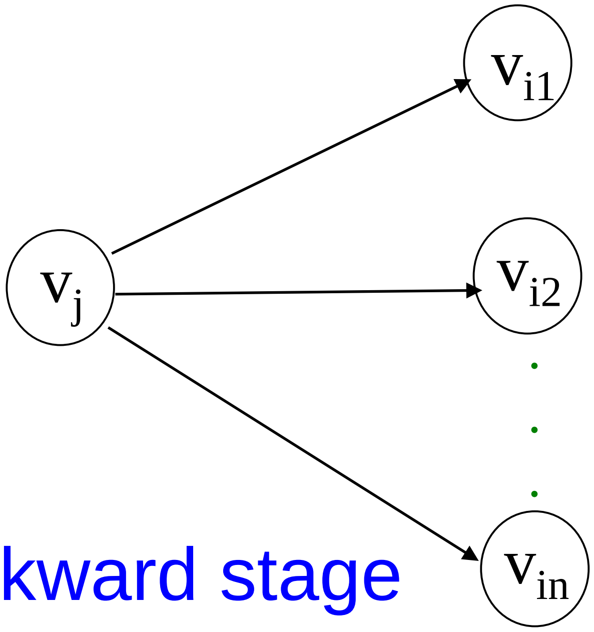
Topological Sorts (15)

- Calculation of latest times
  - In the backward stage, we compute the values of *latest*[*i*] using a procedure analogous to that used in the forward stage.
  - We start with *latest*[*n*-1] = *earliest*[*n*-1] and use the formula:

$$latest[j] = \min_{i \in S(j)} \{latest[i] - \text{duration of} < j, i >\}$$

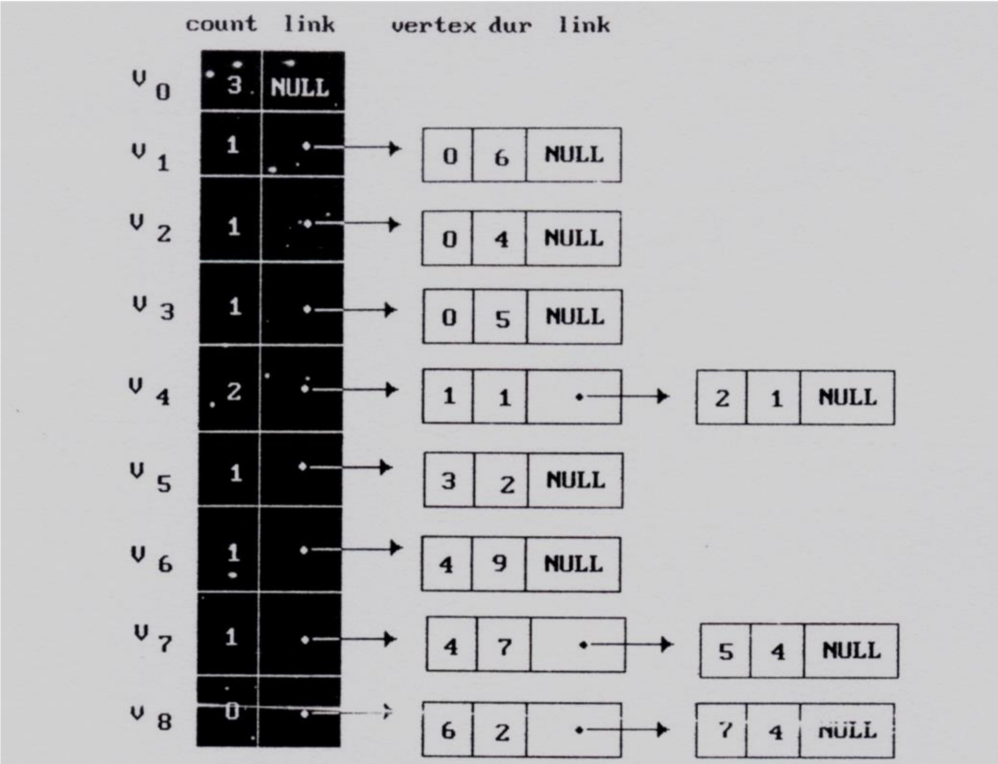
    Where *S*(*j*) is the set of vertices adjacent from vertex *j*
  - Use inverse adjacency list
  - Insert the following statement at the end of the **else** clause in *topsort*

```
if (latest[k] > latest[j] - ptr->duration)
    latest[k] = latest[j] - ptr->duration;
```

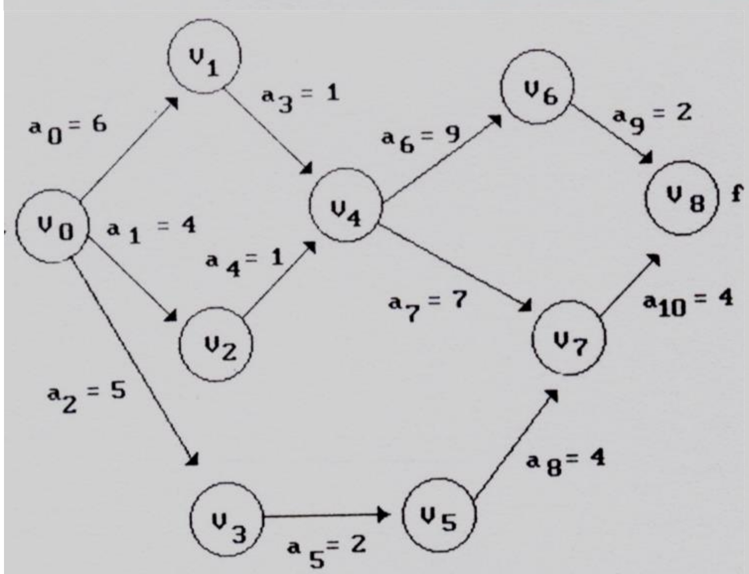


backward stage

Calculation of latest Times (cont'd)




```
for (ptr=graph[j].link;ptr;
    ptr=ptr->link){
    k=ptr->vertex;
    graph[k].count--;
    if(!graph[k].count){
        graph[k].count=top;
        top=k;}
    if(latest[k]>
        latest[j]-ptr->duration)
        latest[k] =
        latest[j]-ptr->duration;
}
```

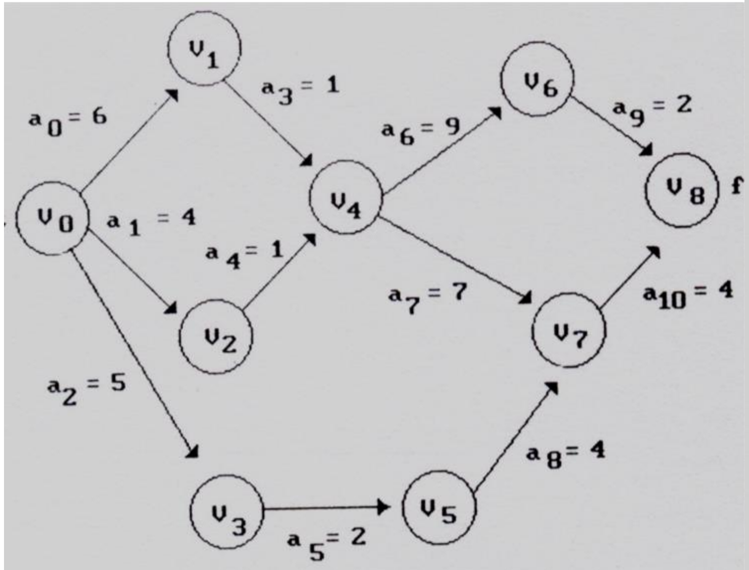


Latest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	18	18	18	18	18	18	18	18	18	[8]
output v_8	18	18	18	18	18	18	16	14	18	[7, 6]
output v_7	18	18	18	18	7	10	16	14	18	[5, 6]
output v_5	18	18	18	18	7	10	16	14	18	[3, 6]
output v_3	3	18	18	8	7	10	16	14	18	[6]
output v_6	3	18	18	8	7	10	16	14	18	[4]
output v_4	3	6	6	8	7	10	16	14	18	[2, 1]
output v_2	2	6	6	8	7	10	16	14	18	[1]
output v_1	0	6	6	8	7	10	16	14	18	[0]

- Calculation of latest Times (cont'd)

output v_6	0	6	4	5	7	7	16	14	18	[8]
output v_8										

(b) Computation of *earliest*



Latest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	18	18	18	18	18	18	18	18	18	[8]
output v_8	18	18	18	18	18	18	16	14	18	[7, 6]
output v_7	18	18	18	18	7	10	16	14	18	[5, 6]
output v_5	18	18	18	18	7	10	16	14	18	[3, 6]
output v_3	3	18	18	8	7	10	16	14	18	[6]
output v_6	3	18	18	8	7	10	16	14	18	[4]
output v_4	3	6	6	8	7	10	16	14	18	[2, 1]
output v_2	2	6	6	8	7	10	16	14	18	[1]
output v_1	0	6	6	8	7	10	16	14	18	[0]

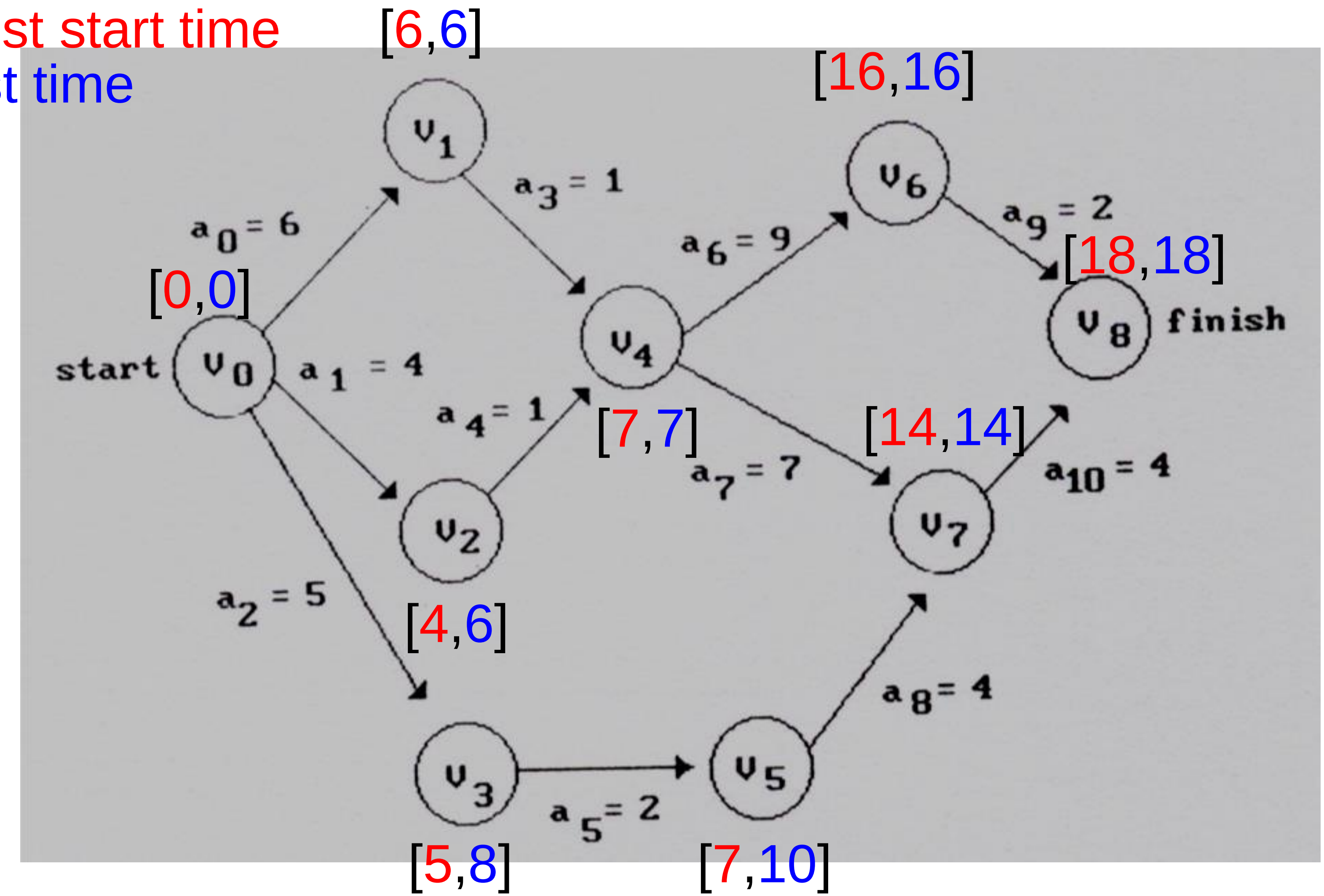
(b) Computation of *latest*

$$\begin{aligned}
latest[8] &= earliest[8] = 18 \\
latest[6] &= \min\{earliest[8] - 2\} = 16 \\
latest[7] &= \min\{earliest[8] - 4\} = 14 \\
latest[4] &= \min\{earliest[6] - 9;\ earliest[7] - 7\} = 7 \\
latest[1] &= \min\{earliest[4] - 1\} = 6 \\
latest[2] &= \min\{earliest[4] - 1\} = 6 \\
latest[5] &= \min\{earliest[7] - 4\} = 10 \\
latest[3] &= \min\{earliest[5] - 2\} = 8 \\
latest[0] &= \min\{earliest[1] - 6;\ earliest[2] - 4;\ earliest[3] - 5\} = 0
\end{aligned}$$

(c) Computation of *latest* from Equation (6.4) using a reverse topological order

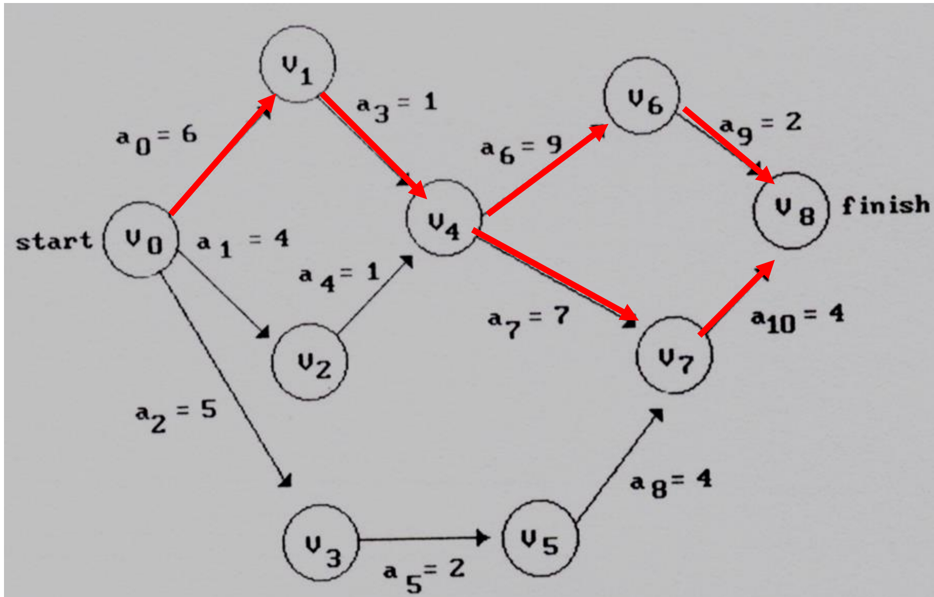
Topological Sorts (19)

Earliest start time
Latest time



Topological Sorts (18)

- Non-critical activities deleted
  - earliest and latest values ⇒ *early*(*i*) and *late*(*i*)
    ⇒ the degree of criticality for each task

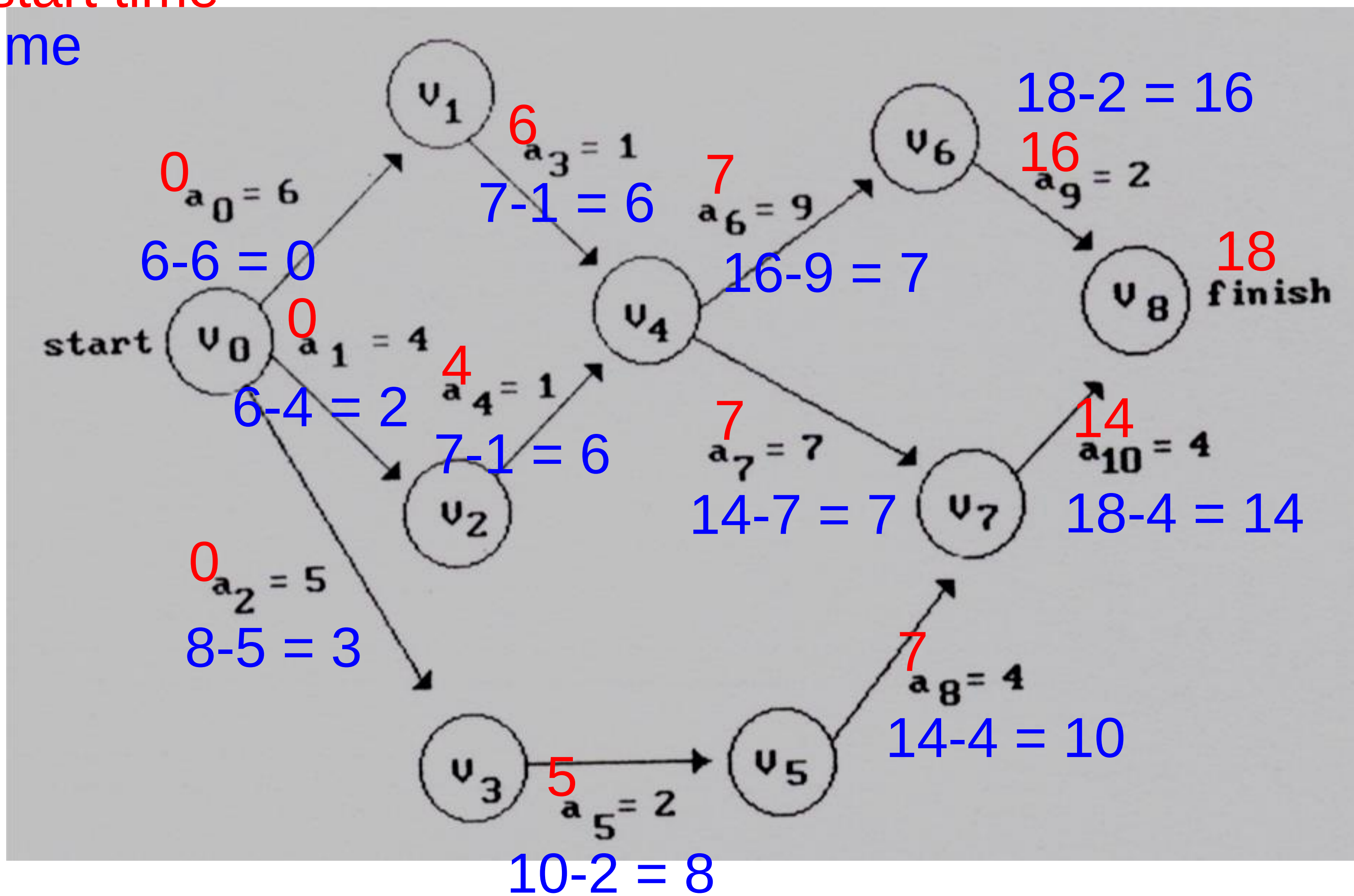


Activity	Early	Late	Late – Early	Critical
a_0	0	0	0	yes
a_1	0	2	2	no
a_2	0	3	3	no
a_3	6	6	0	yes
a_4	4	6	2	no
a_5	5	8	3	no
a_6	7	7	0	yes
a_7	7	7	0	yes
a_8	7	10	3	no
a_9	16	16	0	yes
a_{10}	14	14	0	yes

Figure 6.44: Early, late, and critical values

Topological Sorts (19)

Early start time
Late time



Topological Sorts (20)

- We note that the topsort detects only directed cycles in the network.
- There may be other flaws in the network, including
  - vertices that are not reachable from the start vertex
  - We can also use critical path analysis to detect this kind of fault in project planning