CS235102
Data Structures

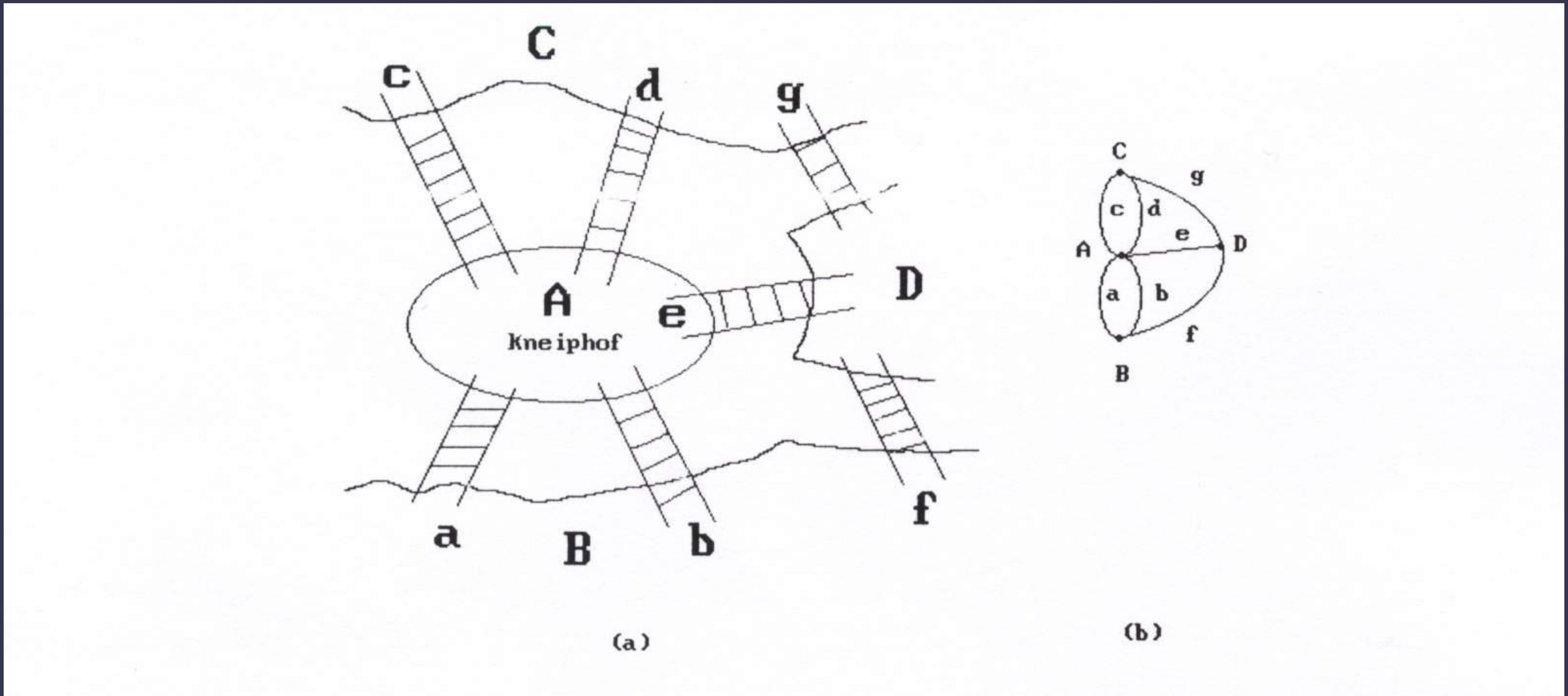
Chapter 6 Graphs

Chapter 6 Graphs: Outline

- The Graph Abstract Data Type
  - Graph Representations
- Elementary Graph Operations
  - Death First and Breadth First Search
  - Spanning Tree
- Minimum Cost Spanning Trees
  - Kruskal's, Prim's and Sollin's Algorithm
- Shortest Paths
  - Transitive Closure
- Topological Sorts
  - Activity Networks
  - Critical Paths

The Graph ADT (1/13)

- Introduction
  - A graph problem example: Köenigsberg bridge problem



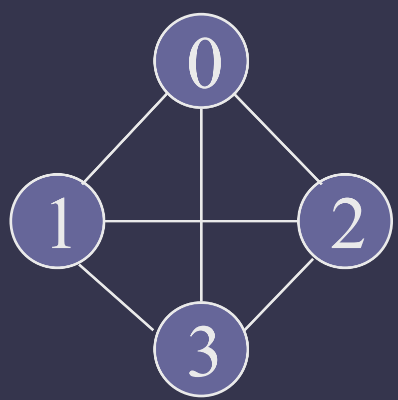
The Graph ADT (2/13)

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

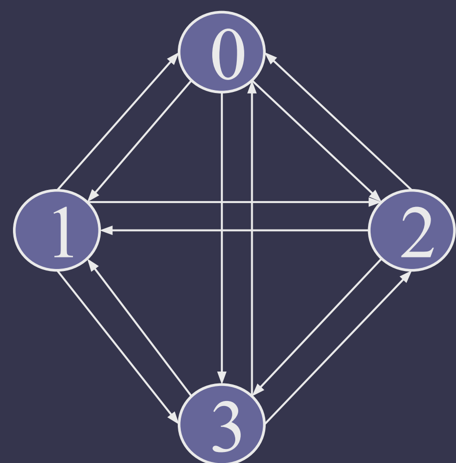
tail $\longrightarrow$ head

The Graph ADT (3/13)

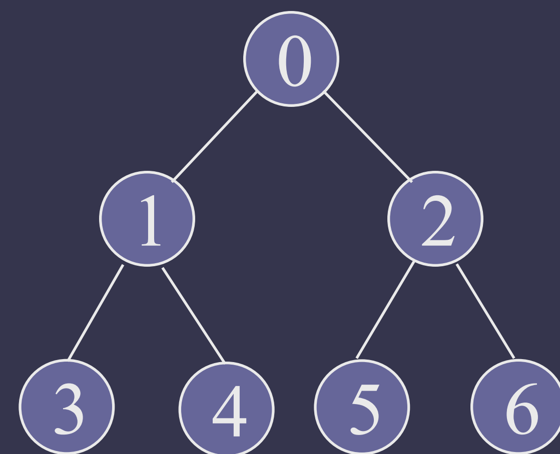
- Examples for Graph
  - complete undirected graph: $n(n-1)/2$ edges
  - complete directed graph: $n(n-1)$ edges



G1



complete graph



G2

G3

incomplete graph

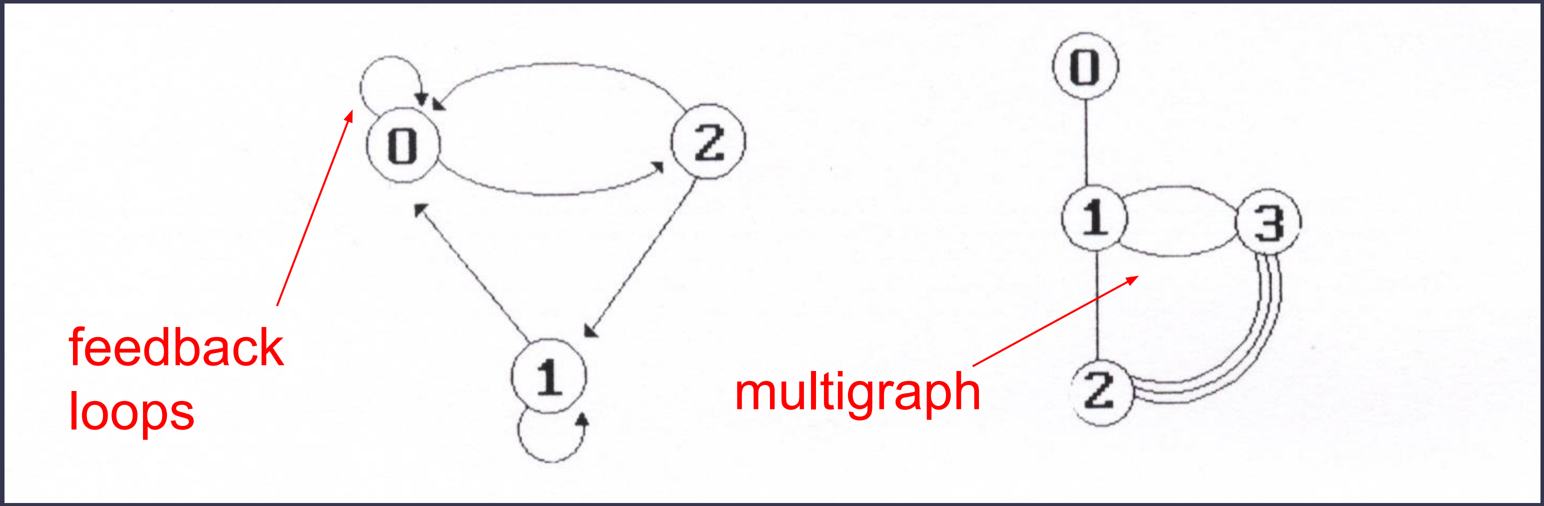
$V(G_1)=\{0,1,2,3\}$ $E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

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

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

The Graph ADT (4/13)

- 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



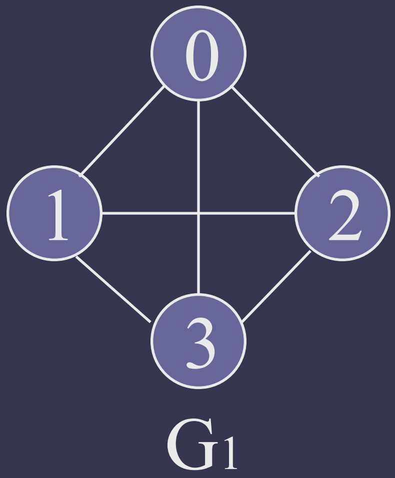
The Graph ADT (5/13)

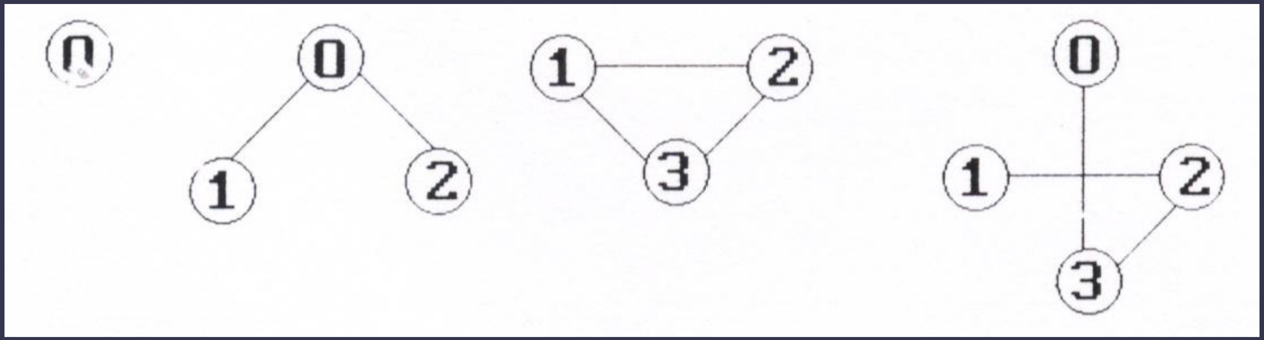
- Adjacent and Incident
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1

- 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 (6/13)

- A subgraph of *G* is a graph *G*' such that V(*G*') ⊆ V(*G*) and E(*G*') ⊆ E(*G*).

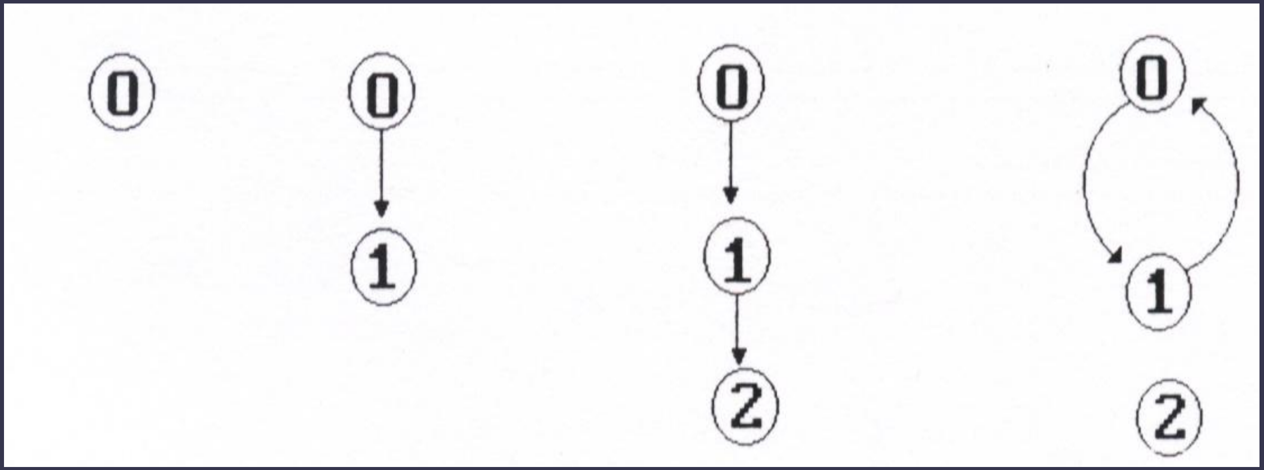




The Graph ADT (6/13)

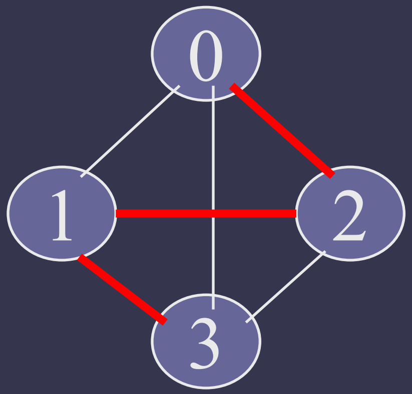
- A subgraph of *G* is a graph *G'* such that V(*G'*) ⊆ V(*G*) and E(*G'*) ⊆ E(*G*).

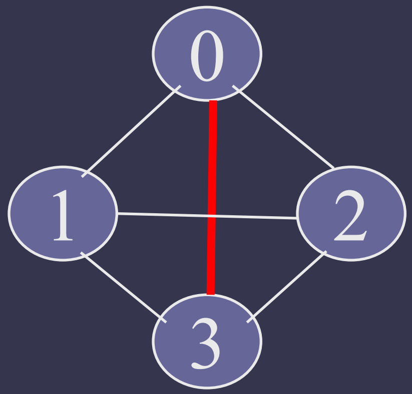
G_3

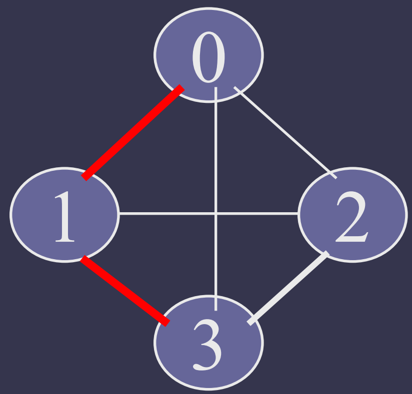


The Graph ADT (7/13)

- Path
  - 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}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, 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

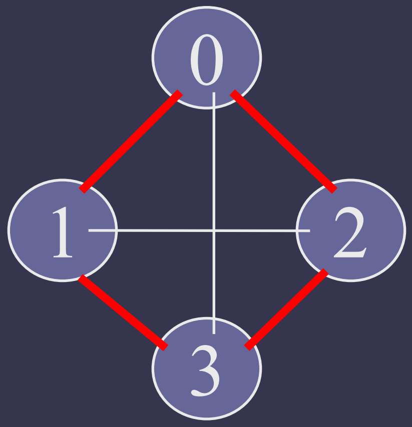






The Graph ADT (8/13)

- 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.

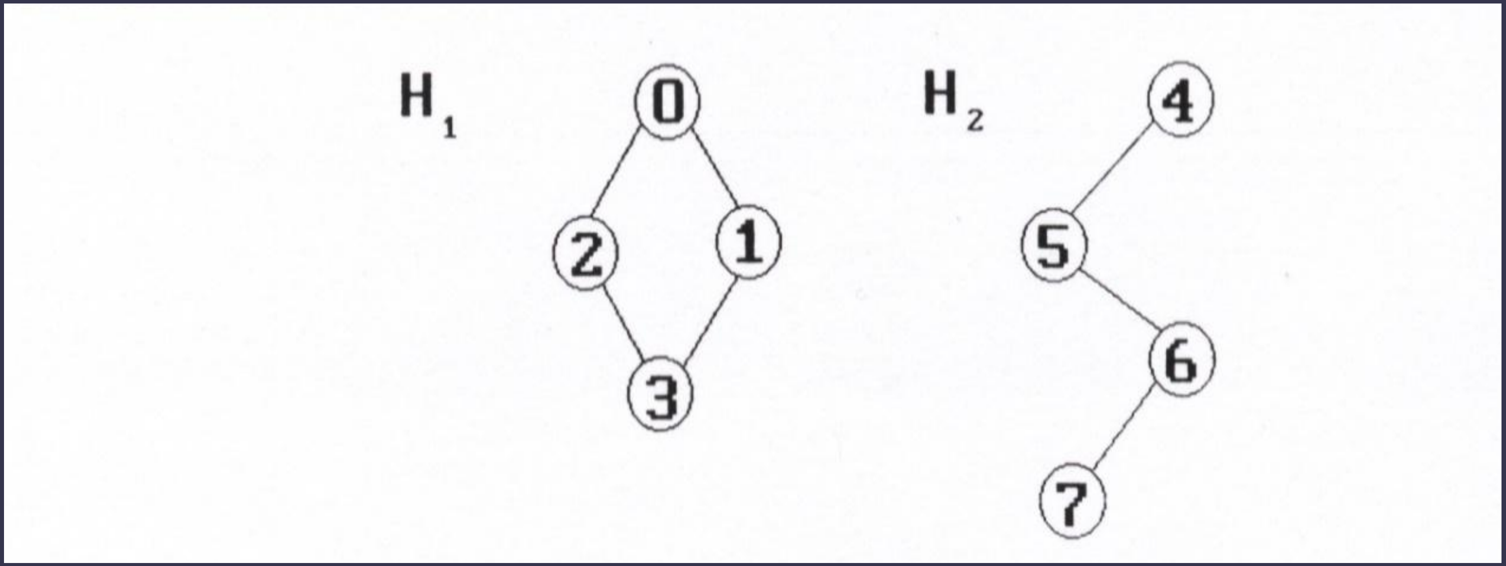


The Graph ADT (9/13)

- **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

The Graph ADT (9/13)

- 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)*.

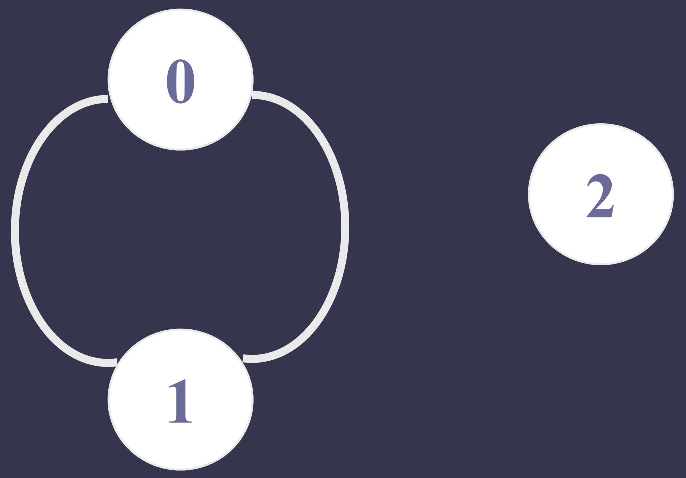


The Graph ADT (10/13)

- **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

G_3

strongly connected component



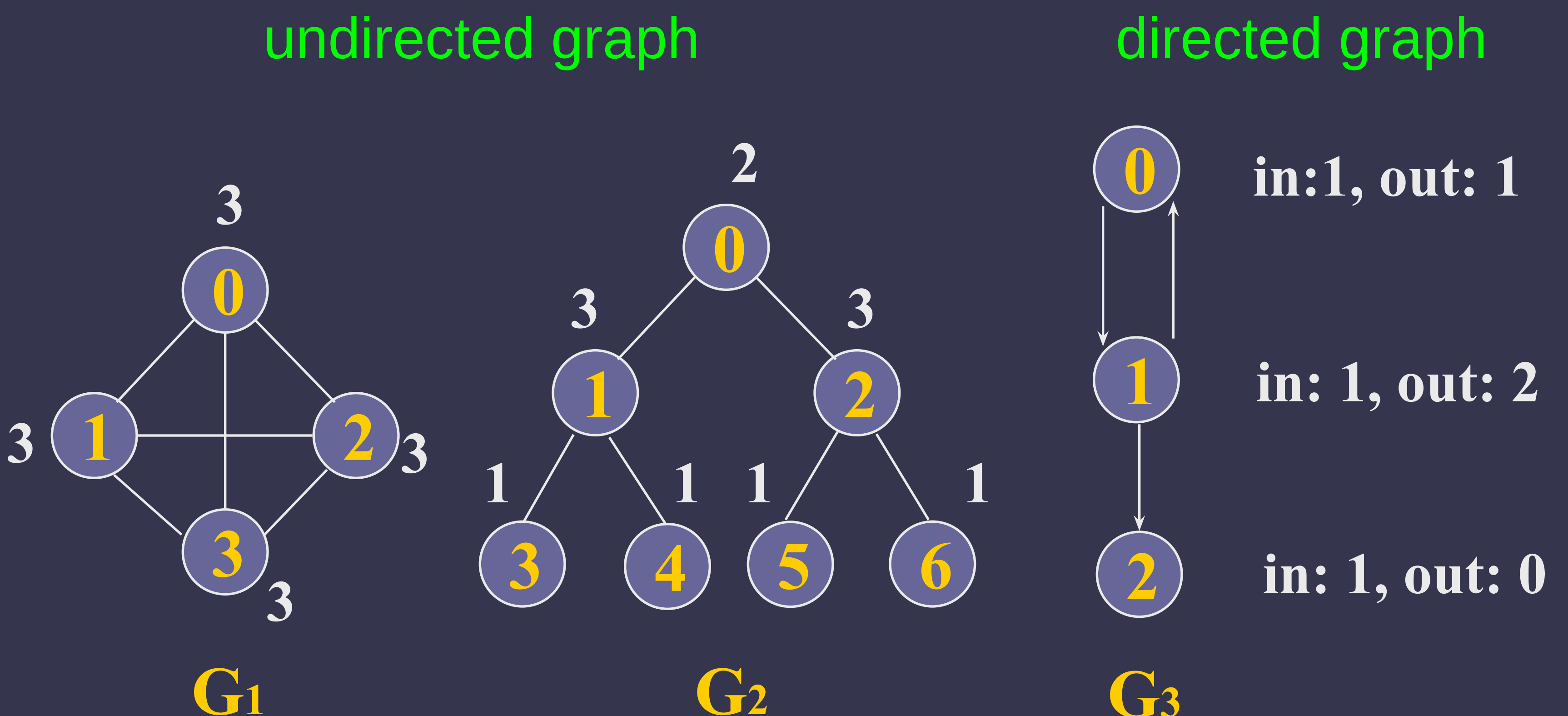
The Graph ADT (11/13)

- 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

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

The Graph ADT (12/13)

- Degree (cont'd)
- We shall refer to a directed graph as a *digraph*. When we us the term *graph*, we assume that it is an undirected graph

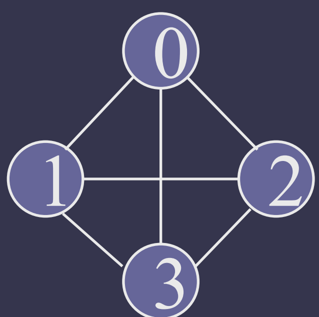


Graph Representations (1/13)

- Adjacency Matrix
- Adjacency Lists
- Adjacency Multilists

Graph Representations (2/13)

- 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)



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

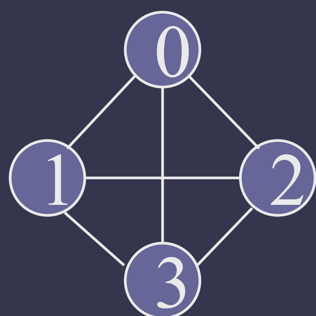
G_1

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

G_3

Graph Representations (2/13)

- Adjacency Matrix
  - The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric



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

G_1

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

G_3

Graph Representations (3/13)

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

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

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

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

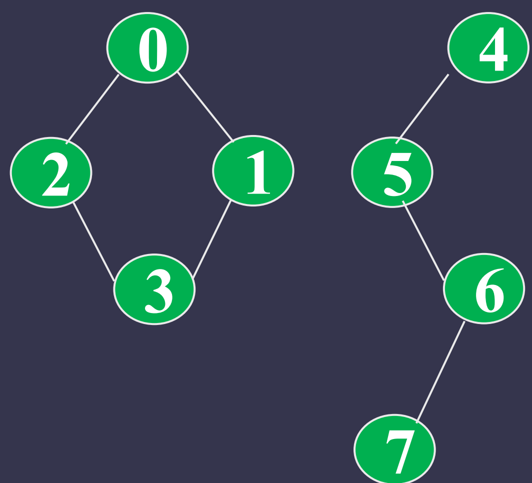
Graph Representations (3/13)

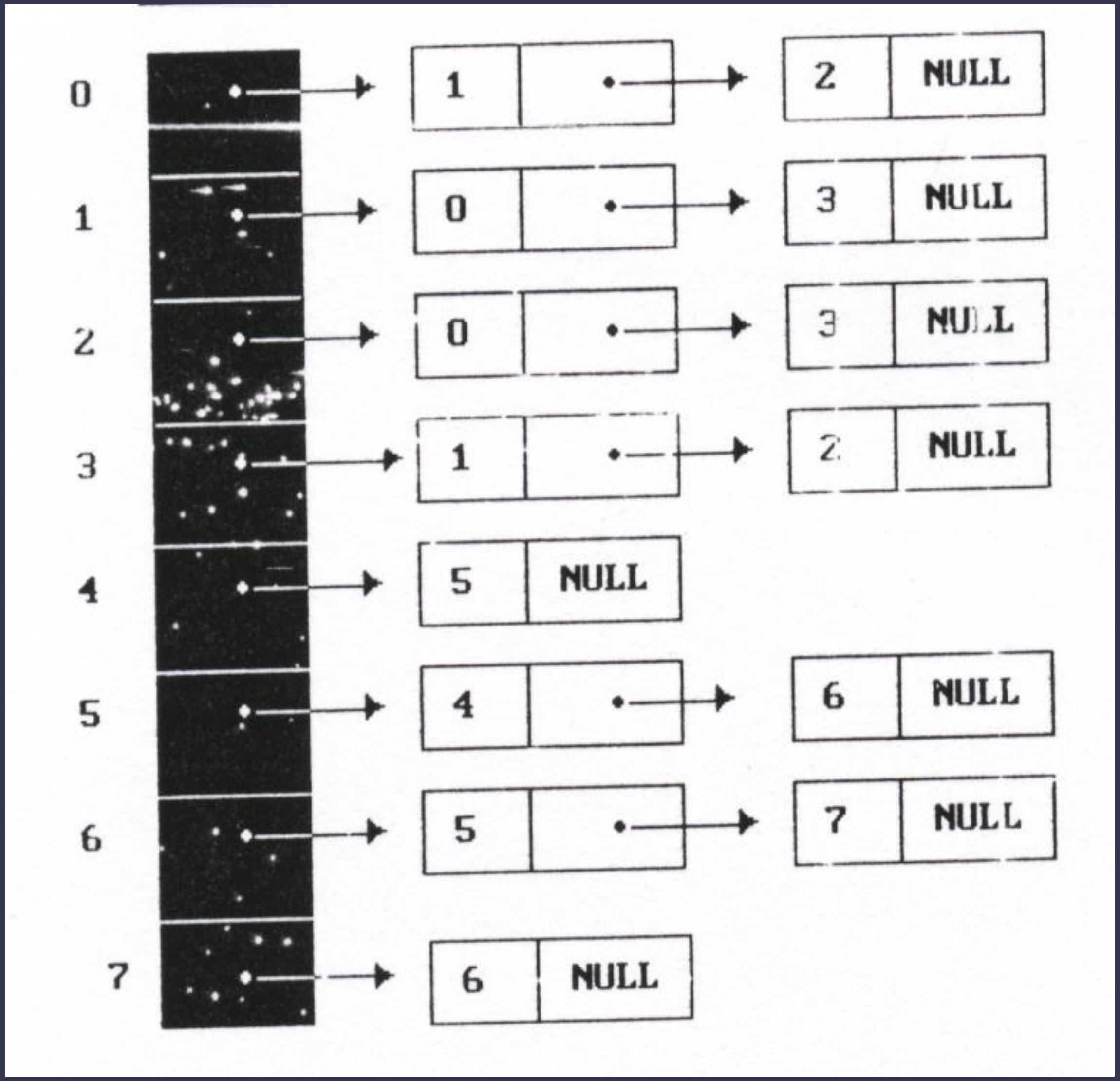
- Merits of Adjacency Matrix
  - The complexity of checking edge number or examining if *G* is connect
    - *G* is undirected: $O(n^2/2)$
    - *G* is directed: $O(n^2)$

Graph Representations (5/13)

- Adjacency lists

G4





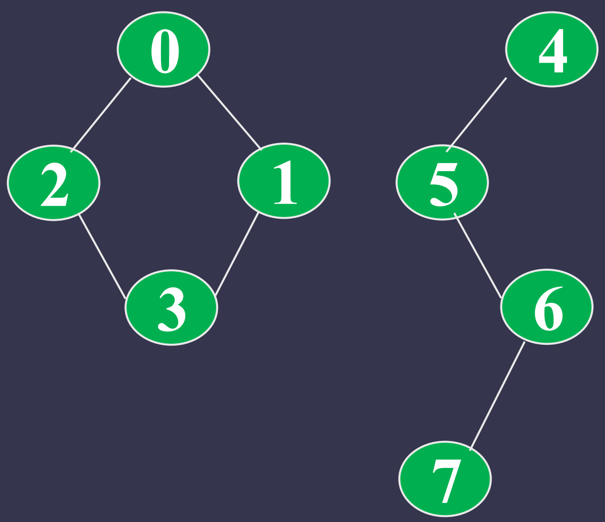
Graph Representations (4/13)

- 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 2*e* 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 (6/13)

- Sequential Representation of Graph



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

Graph Representations (6/13)

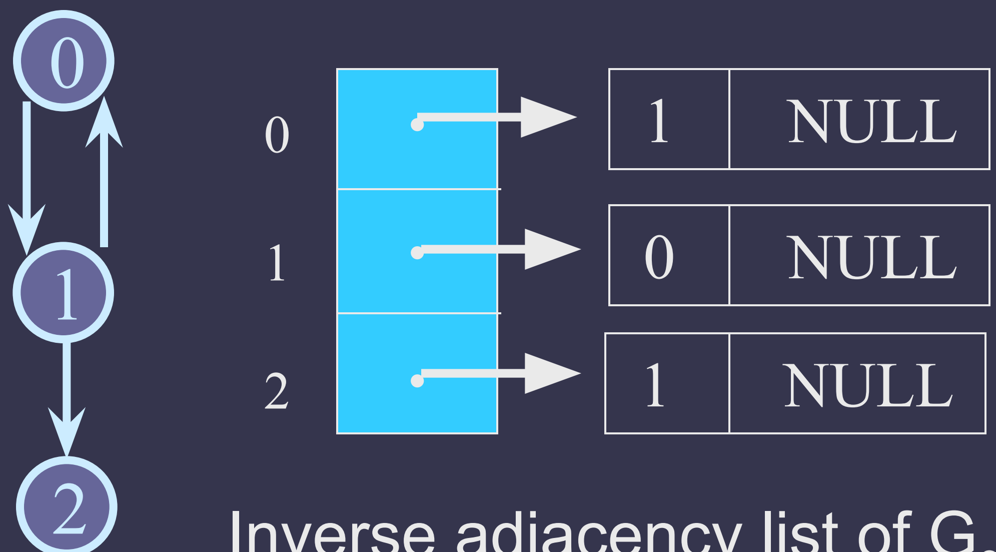
- Sequential Representation of Graph
  - Sequentially pack the nodes on the adjacency lists
  - *node*[0] ~ *node*[n-1] gives the starting point of the list for vertex i , $0 \leqq i < n$
  - *node*[n] stores "$n+2e+1$"
  - The vertices adjacent from vertex i are stored in *node*[*node*[i]], … , *node*[*node*[i+1]-1], $0 \leqq i < n$

Graph Representations (7/13)

- 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

Graph Representations (8/13)

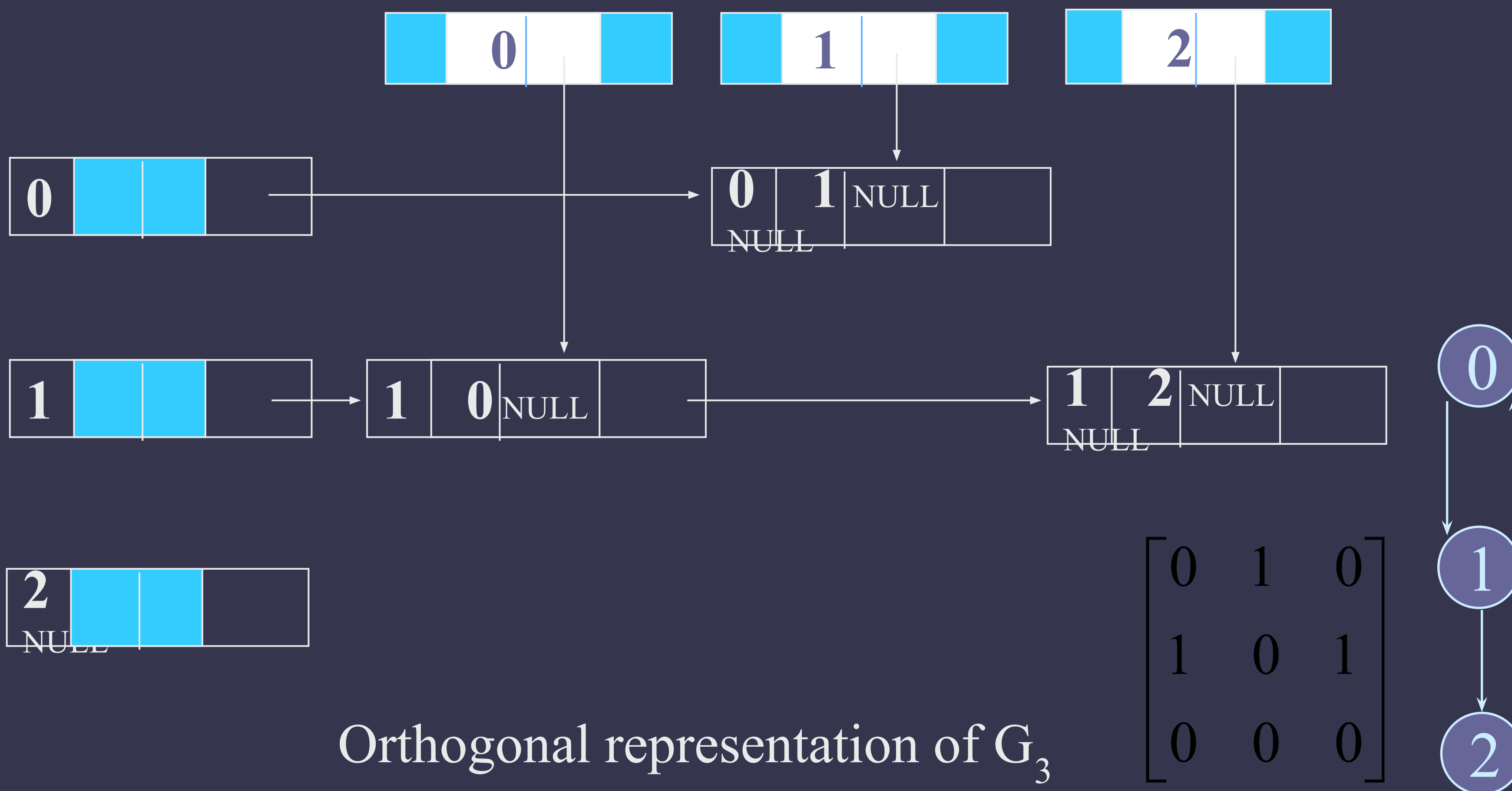
- Finding In-degree of Vertices



Inverse adjacency list of G_3

Graph Representations (9/13)

- Example of Changing Node Structure

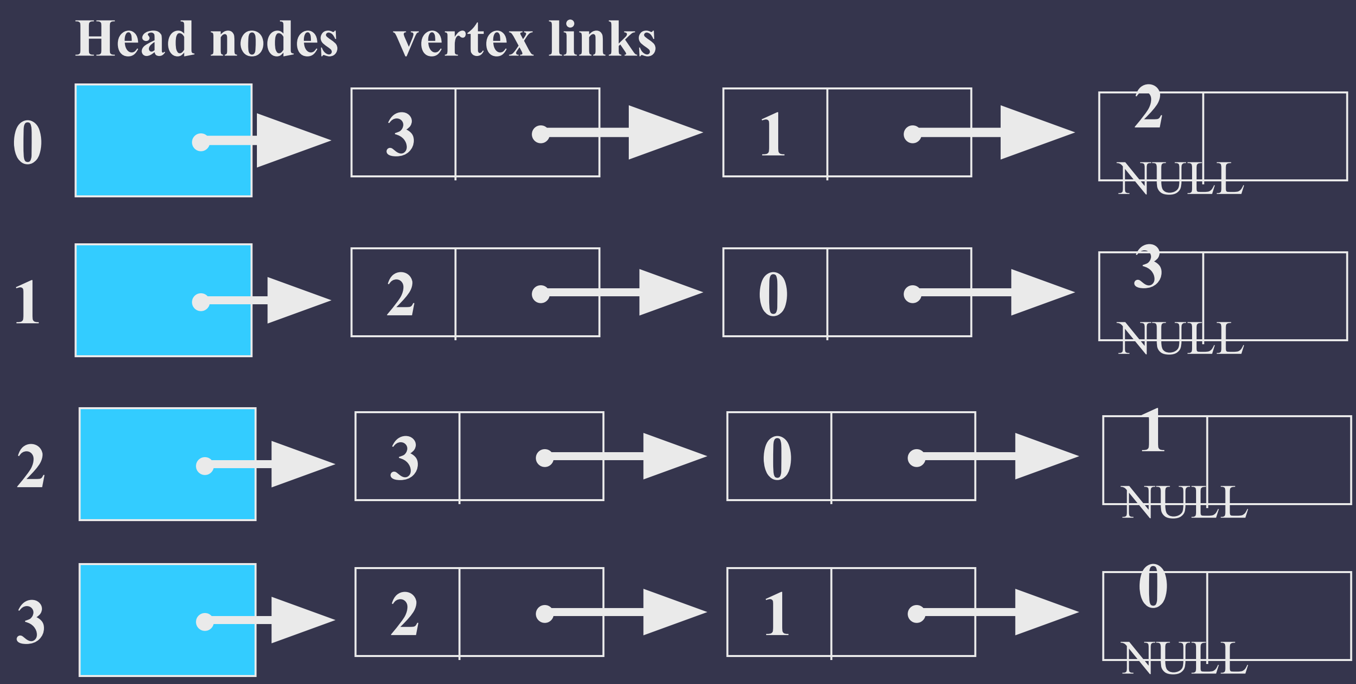


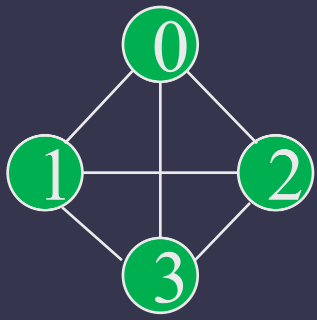
Orthogonal representation of G_3

Graph Representations (10/13)

- Vertices in Any Order

Order is of no significance

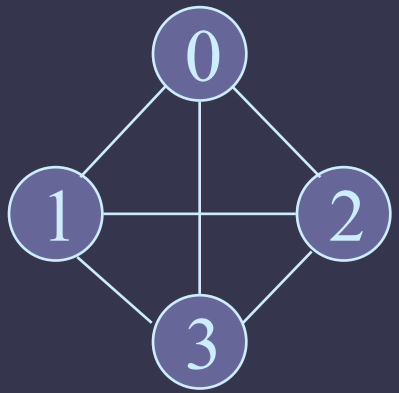


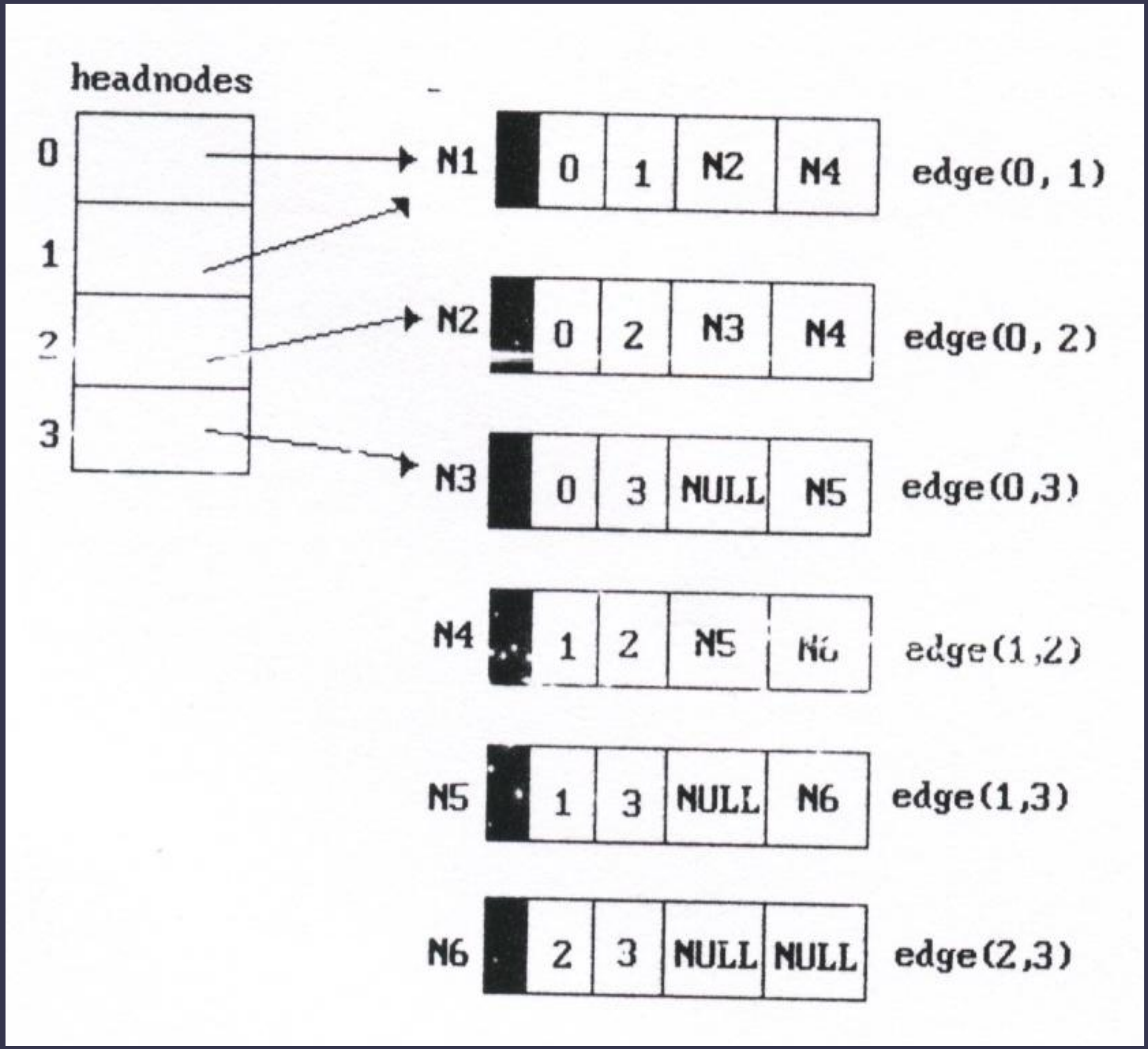


G_1

Graph Representations (12/13)

- Adjacency Multlists





Graph Representations (11/13)

- **Adjacency Multilists**
  - 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

marked	vertex1	vertex2	path1	path2

```
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 (13/13)

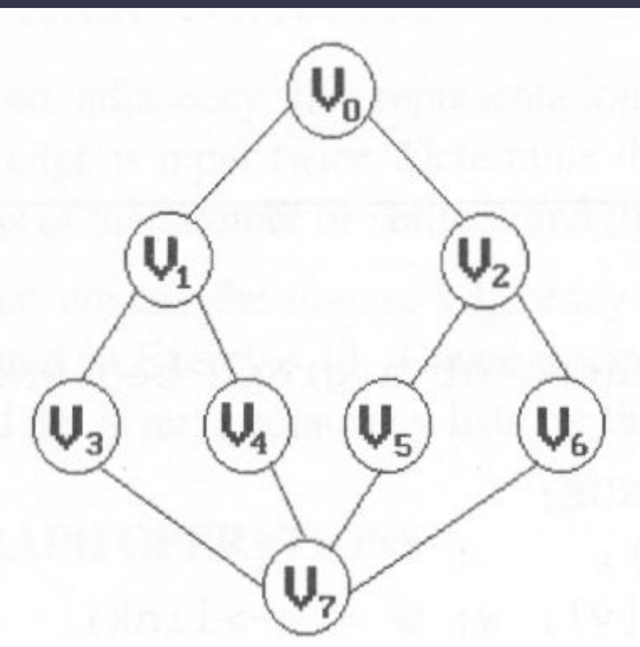
- Weighted edges
  - The edges of a graph have weights assigned to them.
  - These weights may represent as
    - the distance from one vertex to another
    - cost of going from one vertex to an adjacent vertex.
  - adjacency matrix: *adj_mat*[*i*][*j*] would keep the weights.
  - adjacency lists: add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

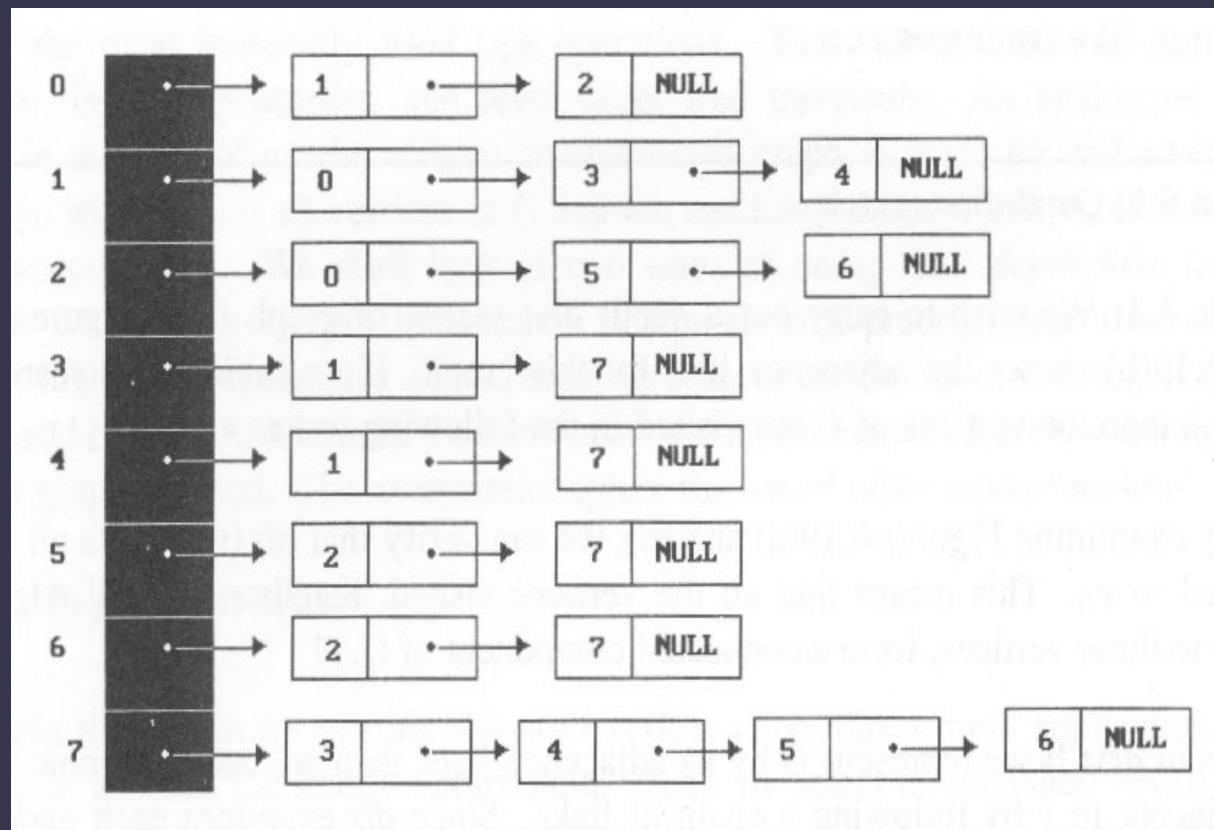
Graph Operations (1/20)

- 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 (2/20)

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

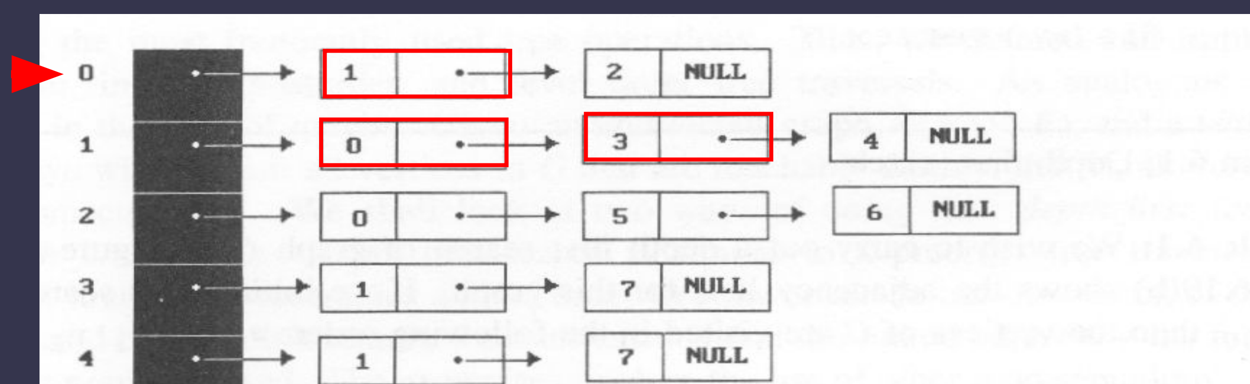




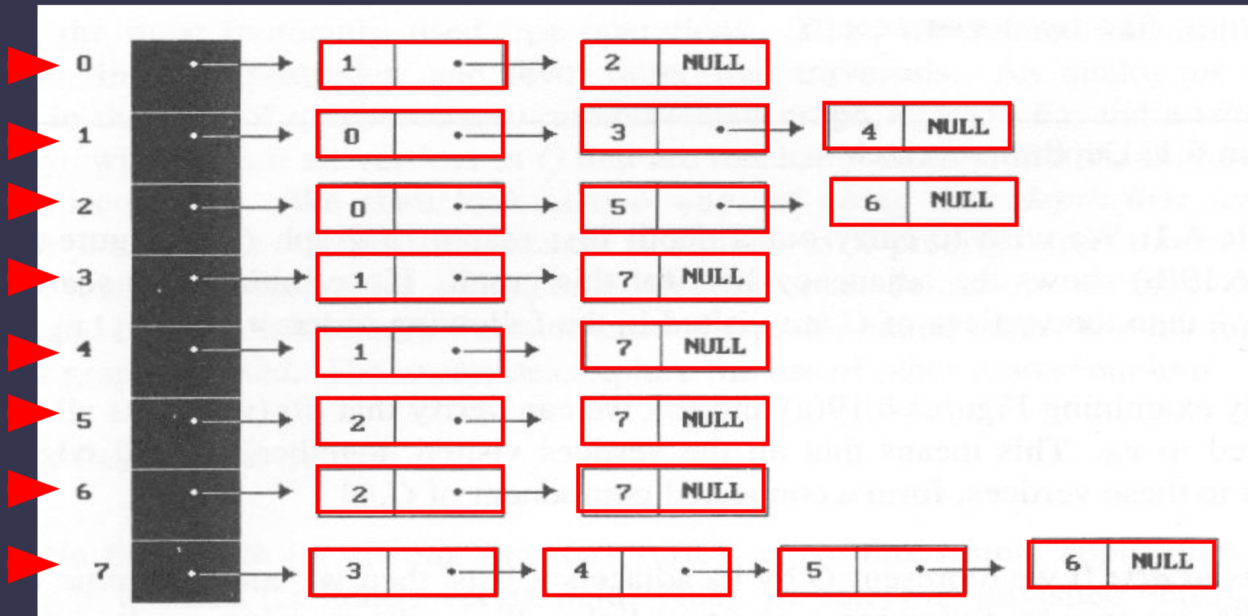
using Adjacency List

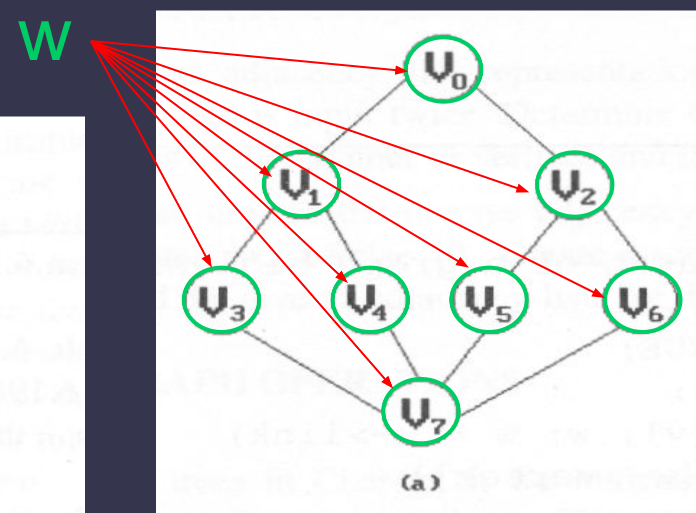
- Depth First Search

```
void dfs(int v)
{
/* depth first search of a graph beginning with vertex 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);
}
```



- 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 (4/20)

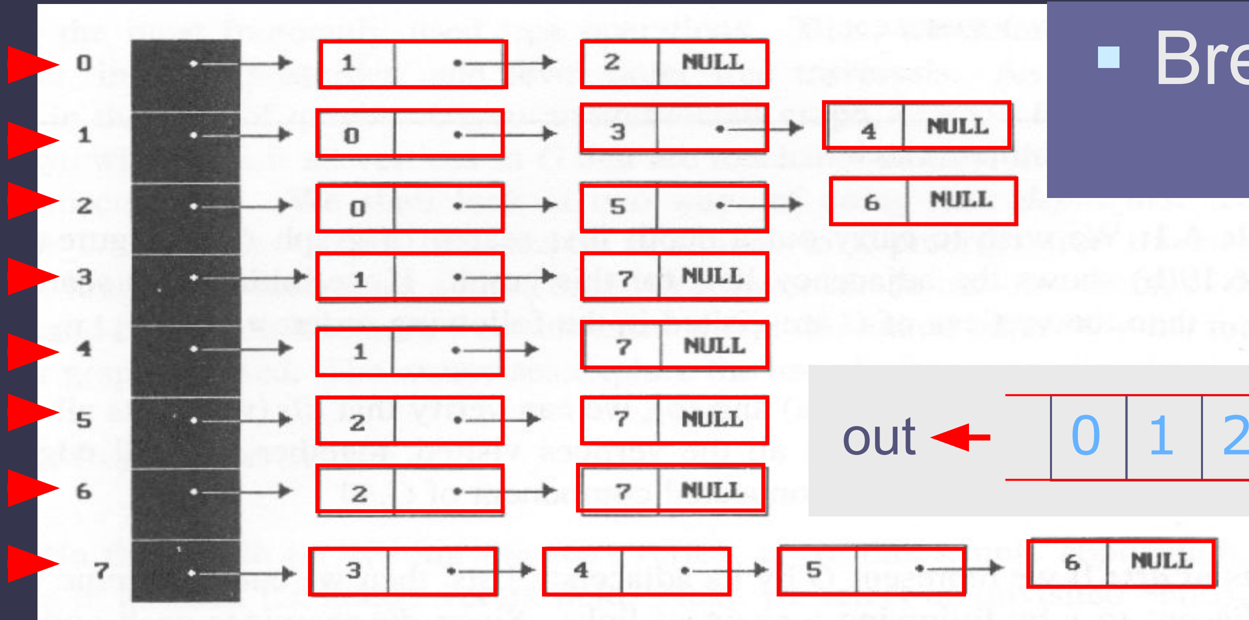
- Breadth First Search
  - It needs a queue to implement breadth-first search
  - void bfs(int v): breadth first traversal of a graph
    - starting with node v the global array visited is initialized to 0
    - the queue operations are similar to those described in Chapter 4

```
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

```
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. */
   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;
         }
      }
}
```

Breadth First Search : Example



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

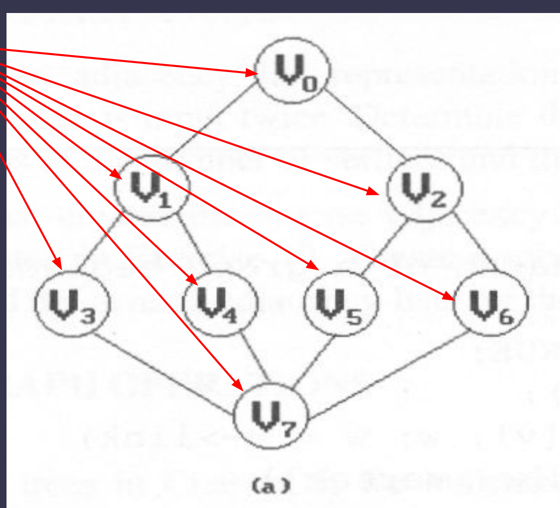
out ← | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ← in

output 0 1 2 3 4 5 6 7

:

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

w




```
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 (6/20)

- 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");
   }
}
```

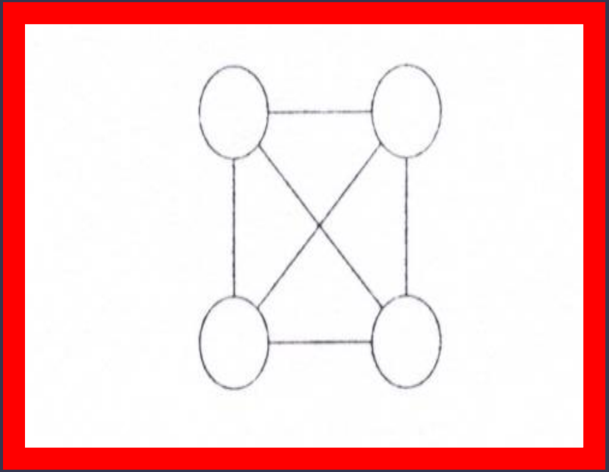
adjacency list: O(n+e)
adjacency matrix: $O(n^2)$

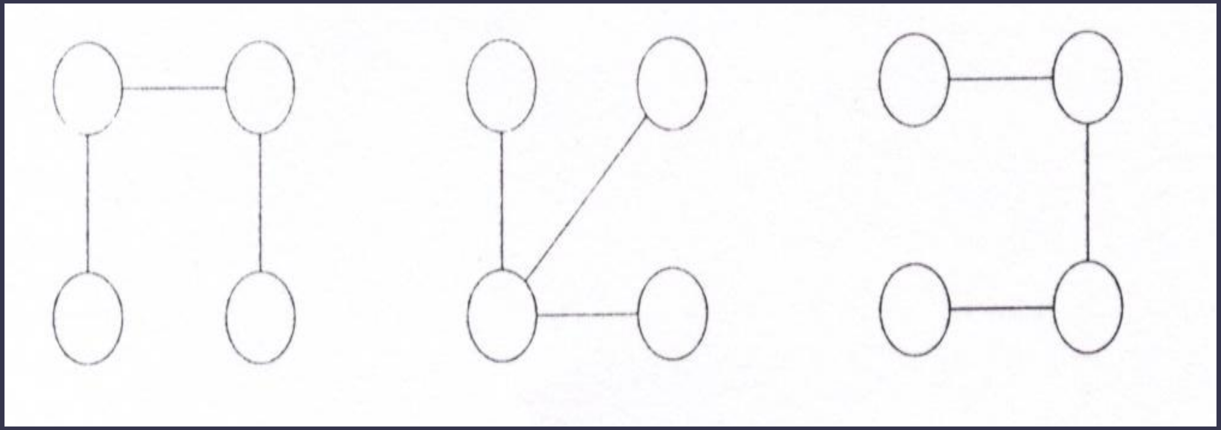
Graph Operations (7/20)

- 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

Graph Operations (7/20)

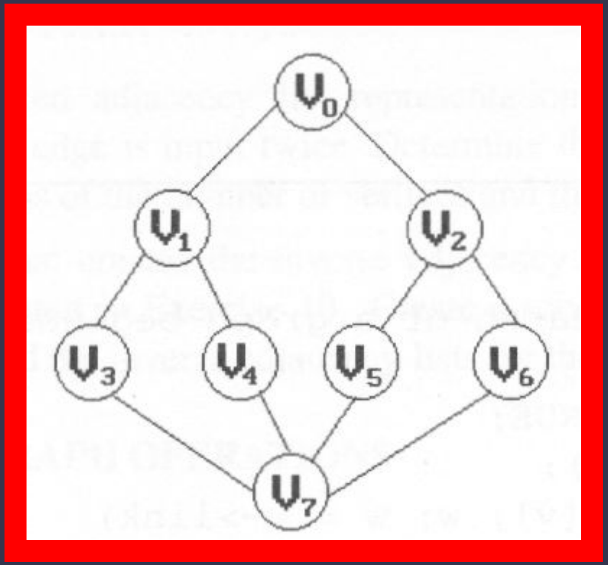
- Spanning trees

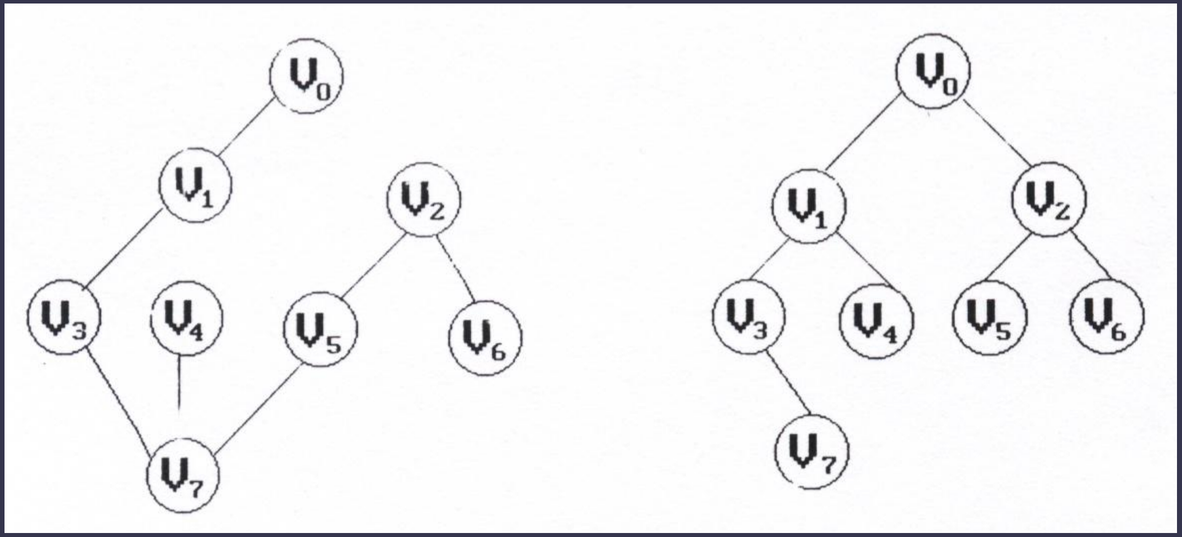




Graph Operations (8/20)

- 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





Graph Operations (9/20)

- 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 (10/20)

- 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.

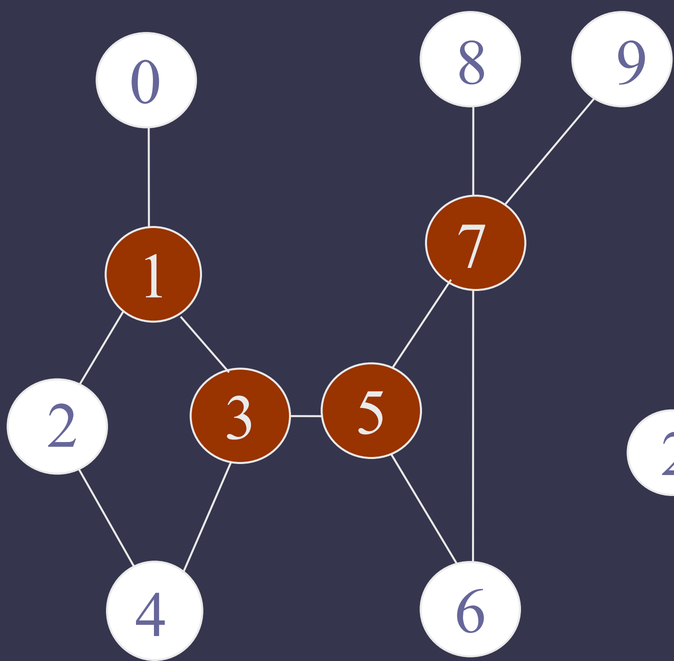
Graph Operations (10/20)

- **Definition:** A *biconnected graph* is a connected graph that has no articulation points.

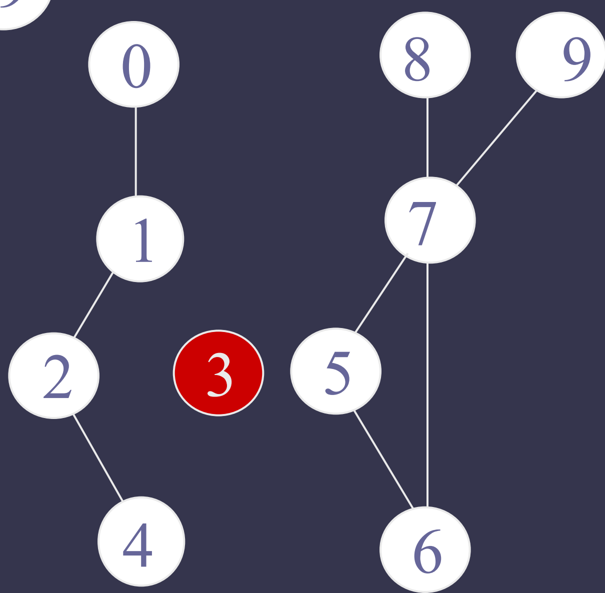
- **Definition:** A *biconnected component* of a connected graph *G* is a maximal biconnected subgraph *H* of *G*.
  - By maximal, we mean that *G* contains no other subgraph that is both biconnected and properly contains *H*.

Graph Operations (11/20)

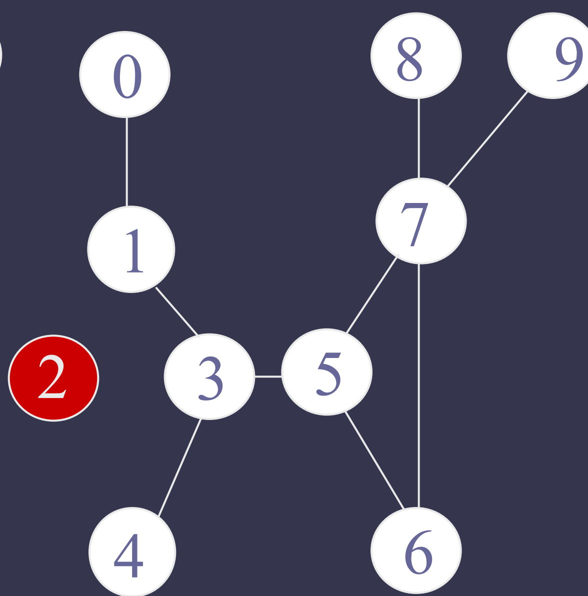
- Examples of Articulation Points (node 1, 3, 5, 7)



Connected graph



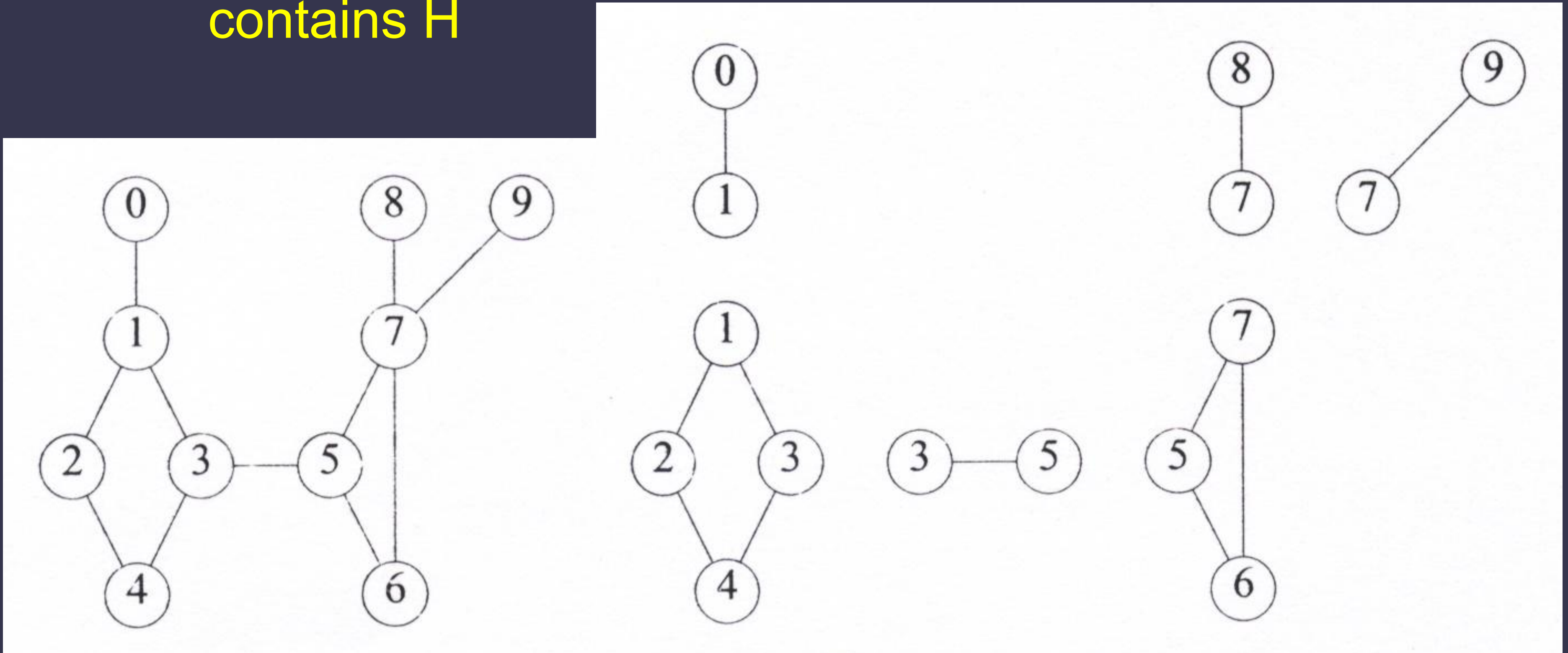
two connected components



one connected graph

Graph Operations (12/20)

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



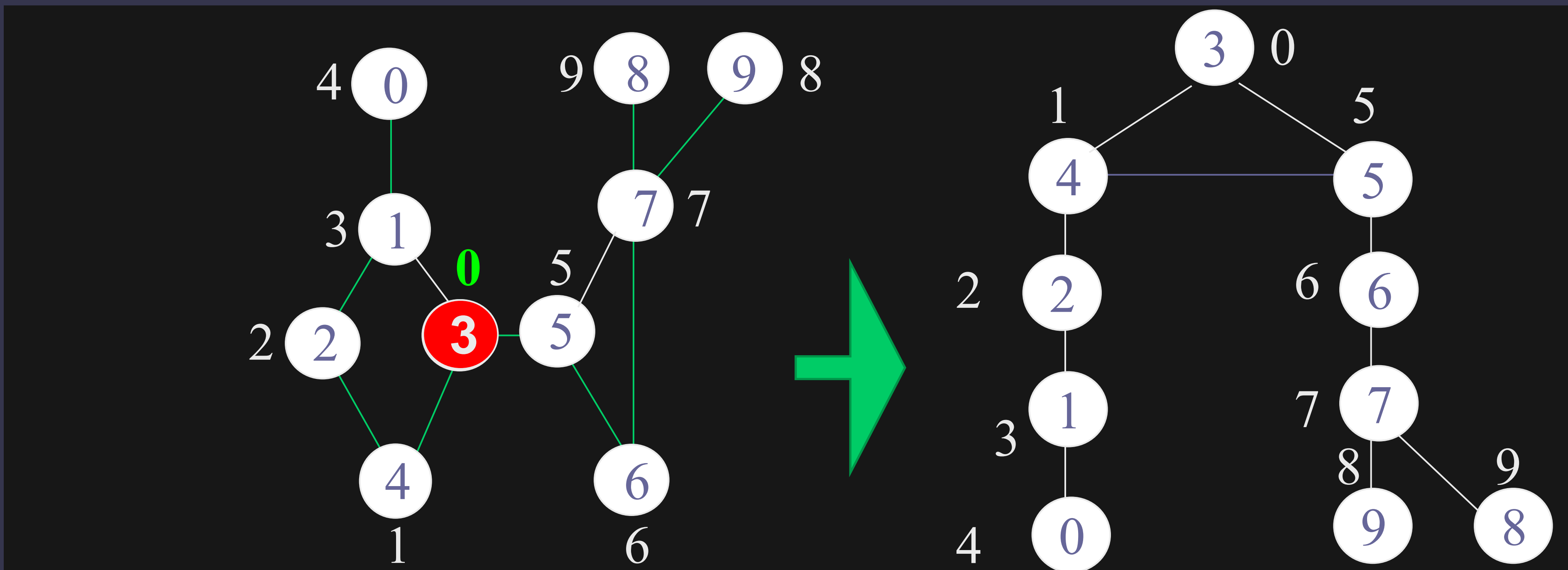
(a) Connected graph (b) Biconnected components

Graph Operations (13/20)

- 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 $dfn(u) < dfn(v)$

Graph Operations (13/20)

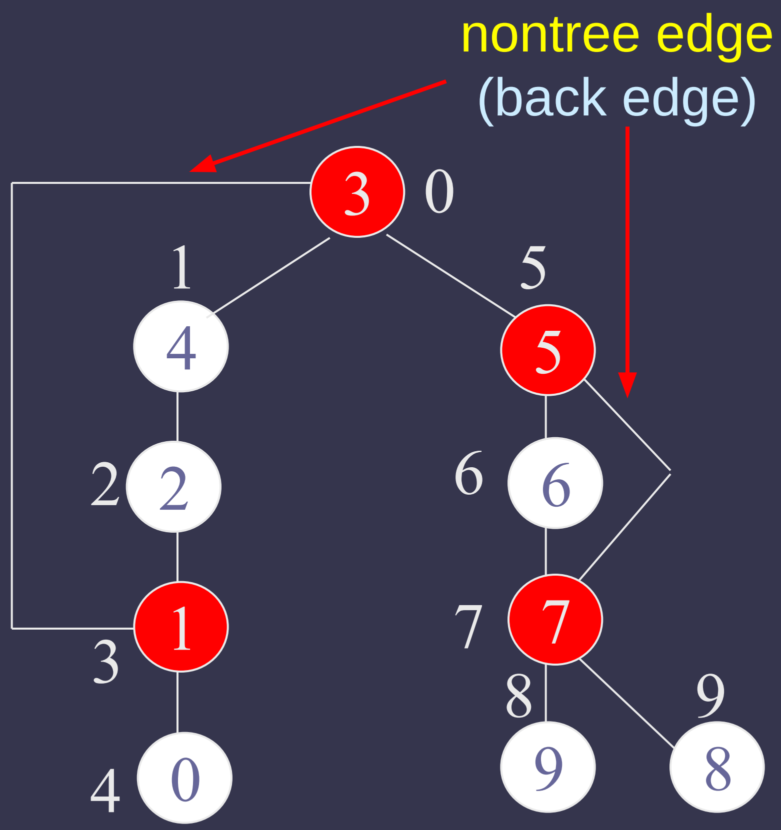
- Finding the biconnected components



result of dfs(3)

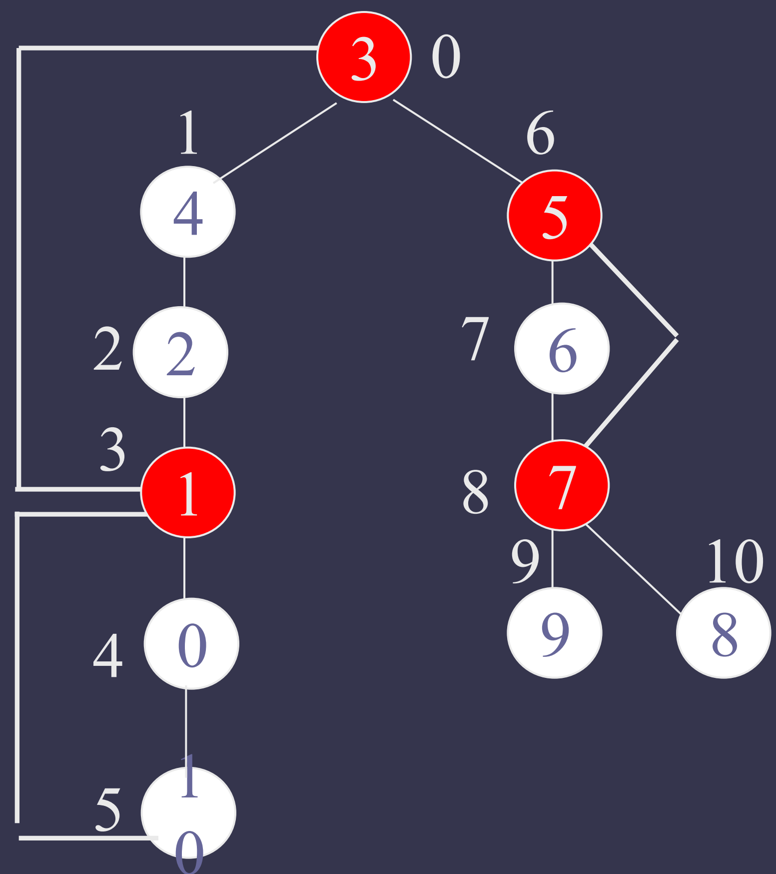
Graph Operations (14/20)

- *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



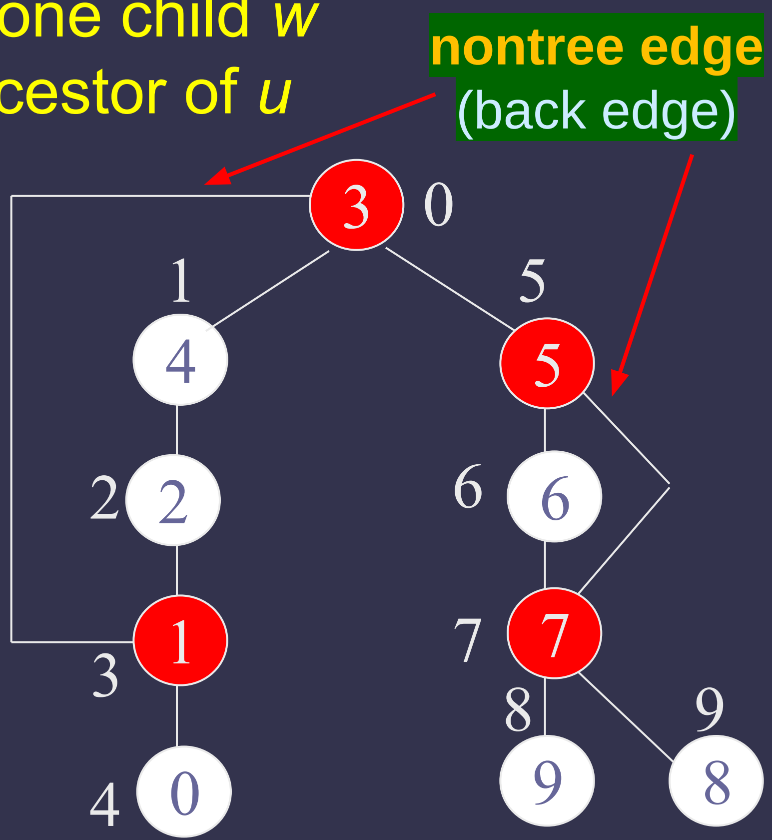
  - *low*(*u*) = min{ *dfn*(*u*), min{ *low*(*w*) | *w* is a child of *u* }, min{ *dfn*(*w*) | (*u*, *w*) is a back edge } }

Graph Operations (14/20)



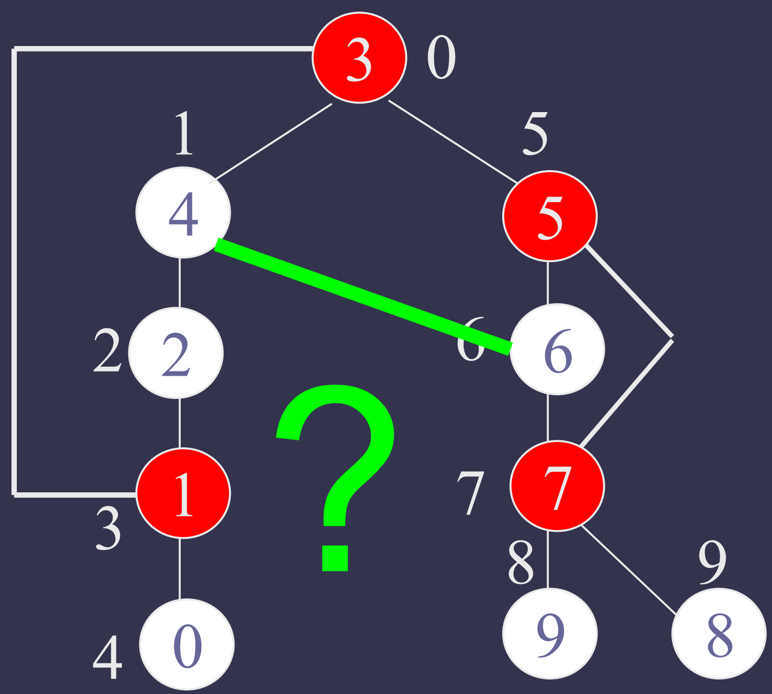
Graph Operations (15/20)

- 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, $\boxed{low(w) \geq dfn(u)}$



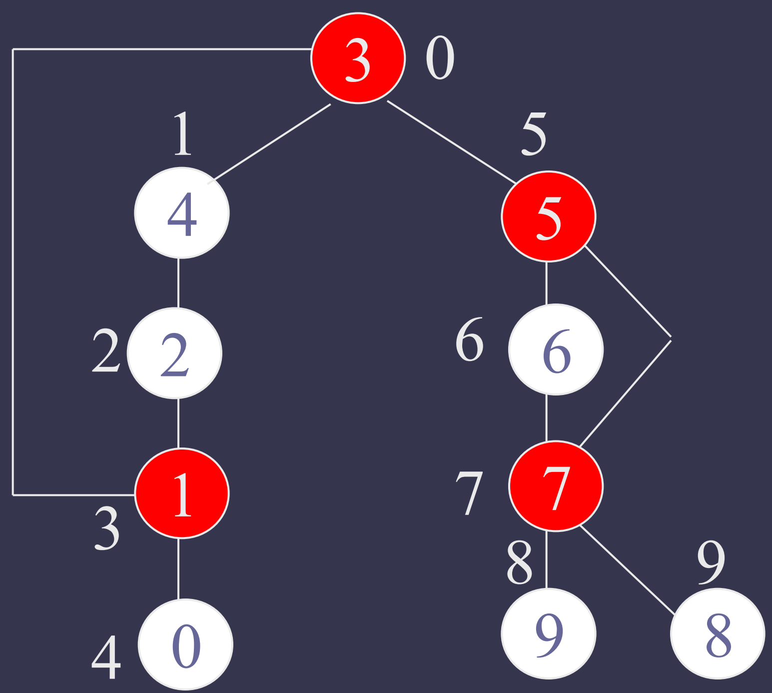
Graph Operations (15/20)

- articulation point in a graph:



Graph Operations (15/20)

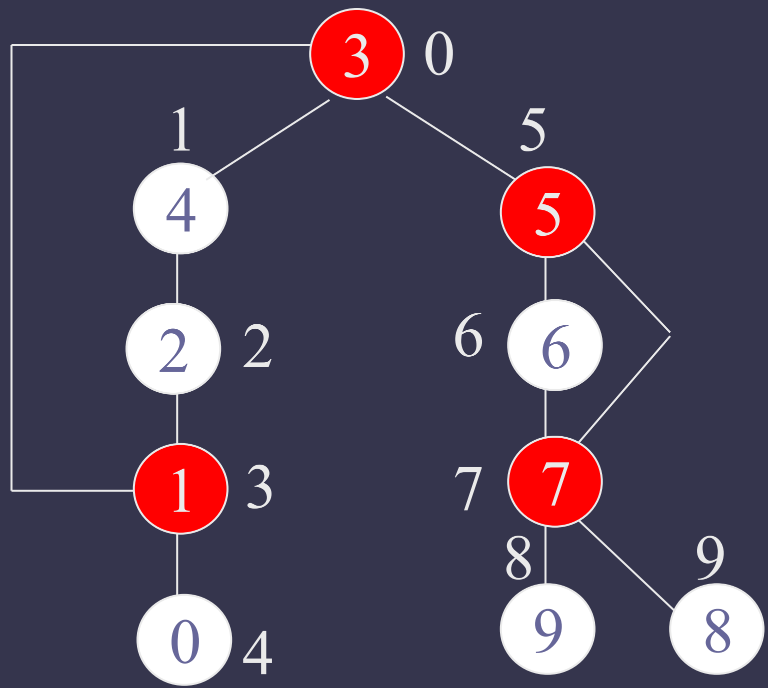
Vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8



Graph Operations (16/20)

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

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	$4 \geq 3$	•
2	2	0 (2,0,n)	1	0	0 < 2	
3	0	0 (0,0,n)	4,5	0,5	$0,5 \geq 0$	•
4	1	0 (1,0,n)	2	0	0 < 1	
5	5	5 (5,5,n)	6	5	$5 \geq 5$	•
6	6	5 (6,5,n)	7	5	5 < 6	
7	7	5 (7,8,5)	8,9	9,8	$9,8 \geq 7$	•
8	9	9 (9,n,n)	null	null	null:9	
9	8	8 (8,n,n)	null	null	null:8	



Graph Operations (17/20)

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

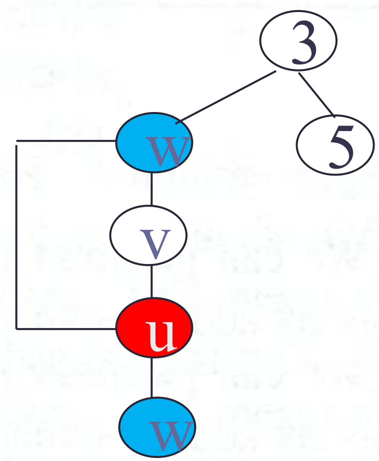
```
void init(void)
{
  int i;
  for (i = 0; i < n; i++) {
    visited[i] = FALSE;
    dfn[i] = low[i] = -1;
  }
  num = 0;
}
```

Determining *dfn* and *low* (cont'd)

```
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)

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



Graph Operations (19/20)

- Partition the edges of the connected graph into their biconnected components
  - If $low[w] \geq dfn[u]$, then we have identified a new biconnected component

Graph Operations (19/20)

- 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

```
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]);
  }
}
```

- 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 [illegible]

v [illegible]

w [illegible]

num [illegible]

7	6	5	3
5	7	6	5

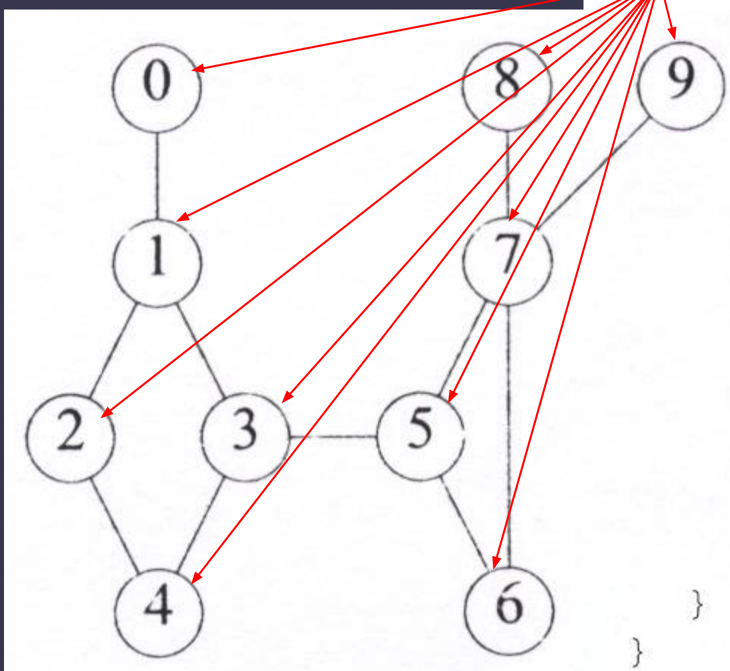
```
void bicon(int u, int v)
{
   node_pointer ptr;
   int w,x,y;
   dfn[u] = low[u] = num++;
   for (ptr = graph[u]; ptr; ptr = ptr->link) {
      w = ptr->vertex;
      if (v != w && dfn[w] < dfn[u])
         add(&top,u,w); /* add edge to stack */
      if (dfn[w] <0) { /* w has not been visited */
         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 (1/7)

- 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.

Minimum Cost Spanning Trees (1/7)

- Introduction
  - 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 (2/7)

- 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

Minimum Cost Spanning Trees (2/7)

- Greedy Strategy
- 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 (3/7)

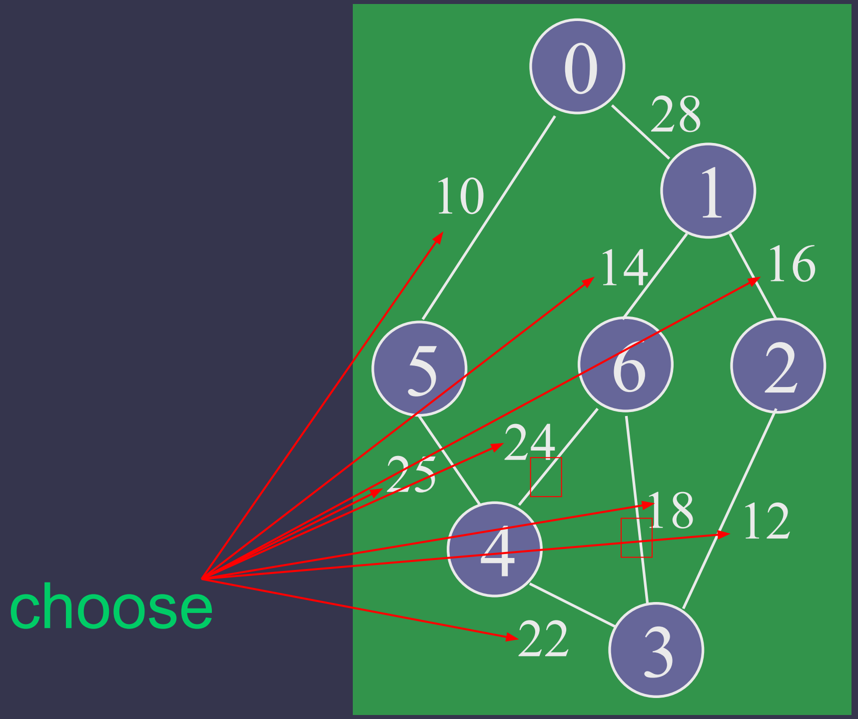
- Kruskal's Algorithm
  - Build a minimum cost spanning tree T by adding edges to T one at a time
  - Select the edges for inclusion in T in non-decreasing order of the cost
  - An edge is added to T if it does not form a cycle
  - Since G is connected and has $n > 0$ vertices, exactly n-1 edges will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

Minimum Cost Spanning Trees (3/7)

- **Theorem 6.1:**
  Let *G* be an undirected connected graph. Kruskal's algorithm generates a minimum cost spanning tree

Minimum Cost Spanning Trees (4/7)

- Kruskal's Algorithm (cont'd)

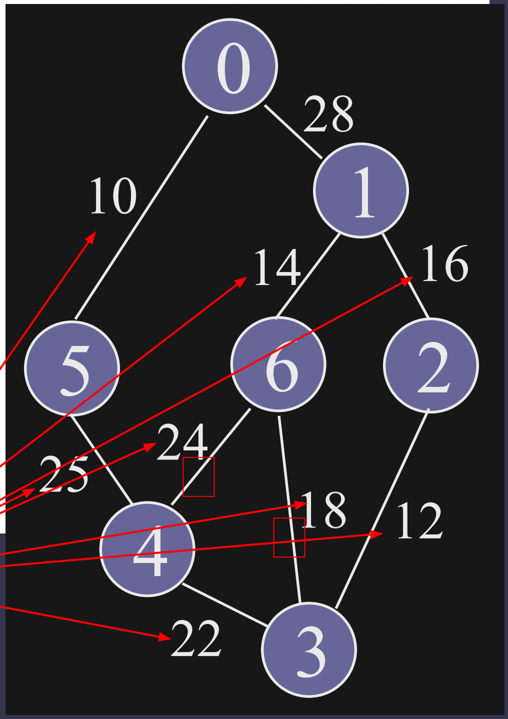


Minimum Cost Spanning Trees (4/7)

- 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");
```

choose

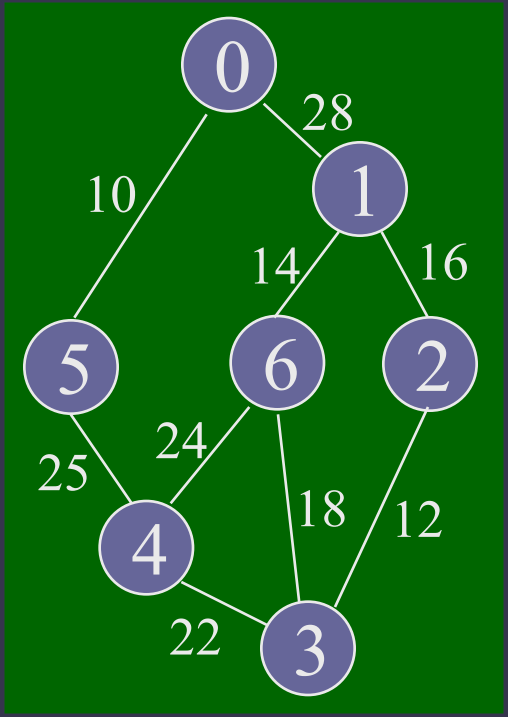


Minimum Cost Spanning Trees (5/7)

- 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 (6/7)

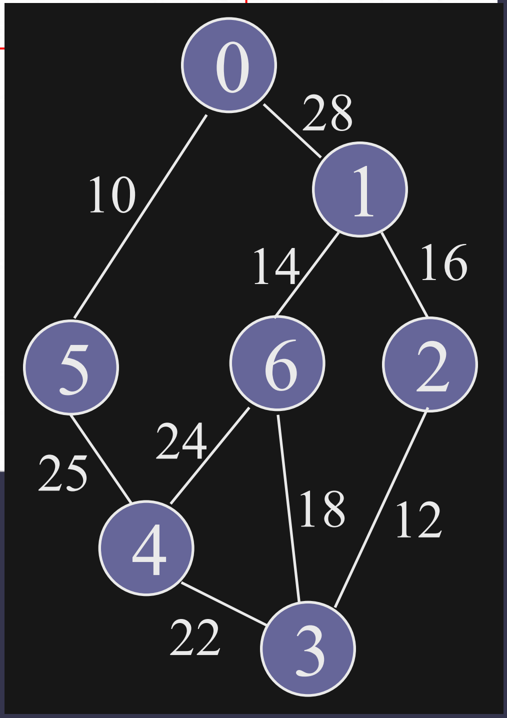
- Prim's Algorithm (cont'd)



Minimum Cost Spanning Trees (6/7)

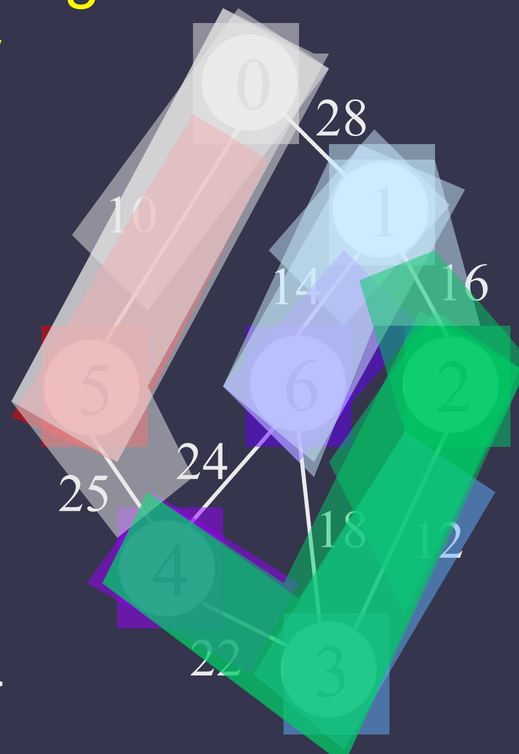
- Prim's Algorithm (cont'd)

```
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");
```



Minimum Cost Spanning Trees (7/7)

- 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 vertex in the tree edge for each tree 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)}



Minimum Cost Spanning Trees (7/7)

- Sollin's Algorithm