Course Name: Data Structures and Applications

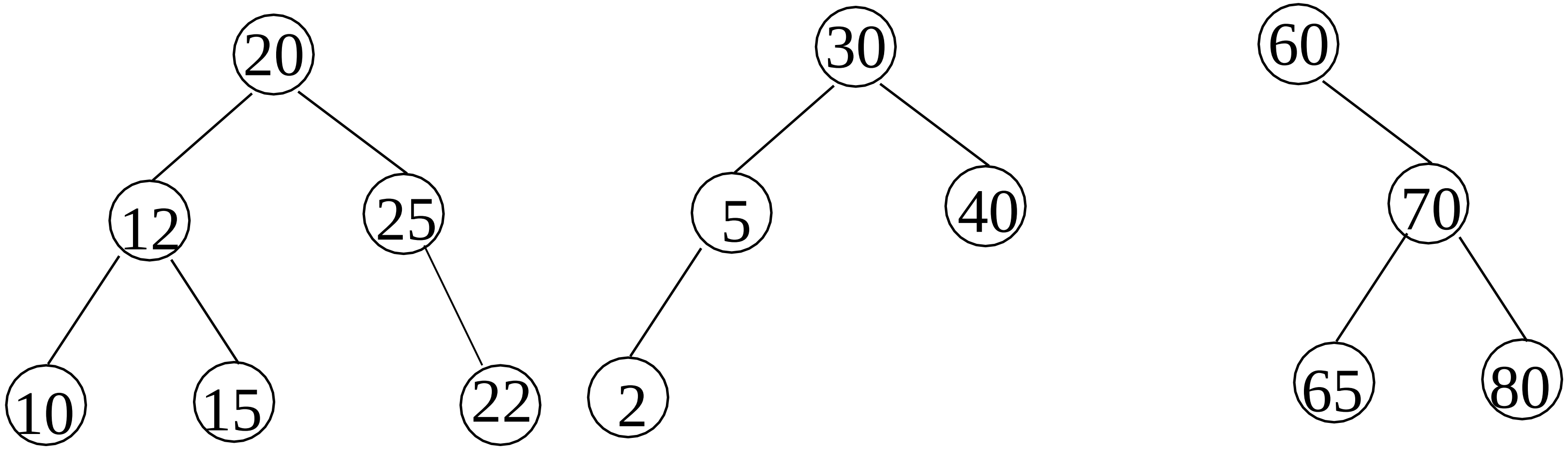
Course Code: BCS304

Module 4
Trees

Binary Search Tree

- Heap
  - a min (max) element is deleted. $O(\log_2 n)$
  - deletion of an arbitrary element $O(n)$
  - search for an arbitrary element $O(n)$
- Binary search tree
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.

Examples of Binary Search Trees



Searching a Binary Search Tree

```
tree_pointer search(tree_pointer root,
                    int key)
{
/* return a pointer to the node that
  contains key. If there is no such  node,
  return NULL */

   if (!root) return NULL;
   if (key == root->data) return root; if
   (key < root->data)
        return search(root->left_child,
                      key);
   return search(root->right_child,key);
}
```

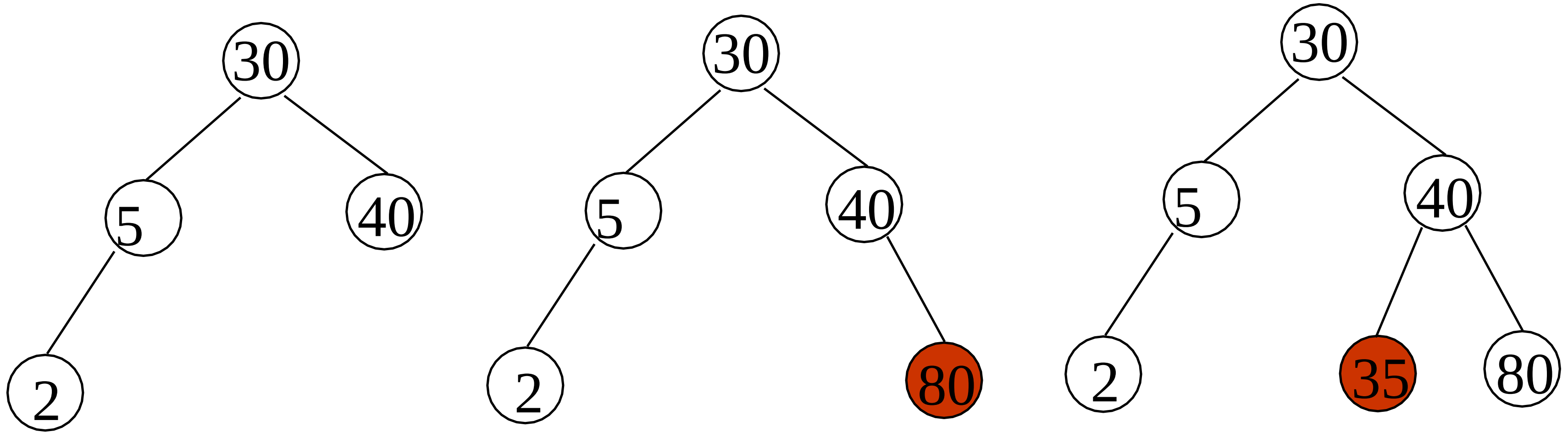
Another Searching Algorithm

```
tree_pointer search2(tree_pointer tree, int
  key)
{
  while (tree) {
    if (key == tree->data) return tree; if
    (key < tree->data)
         tree = tree->left_child;  else
    tree = tree->right_child;


  }
  return NULL;
}
```

O(h)

Insert Node in Binary Search Tree



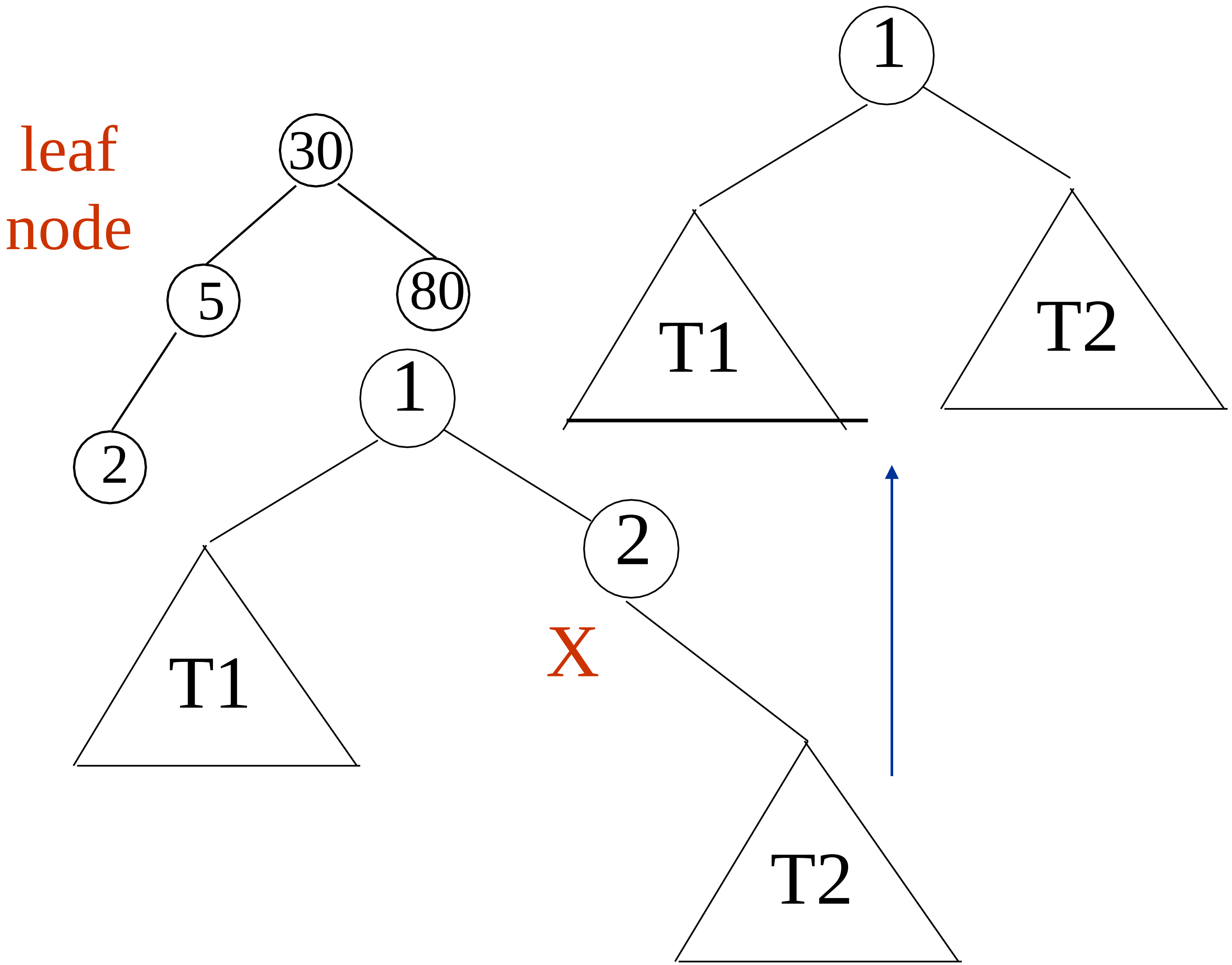
Insert 80

Insert 35

Insertion into A Binary Search Tree

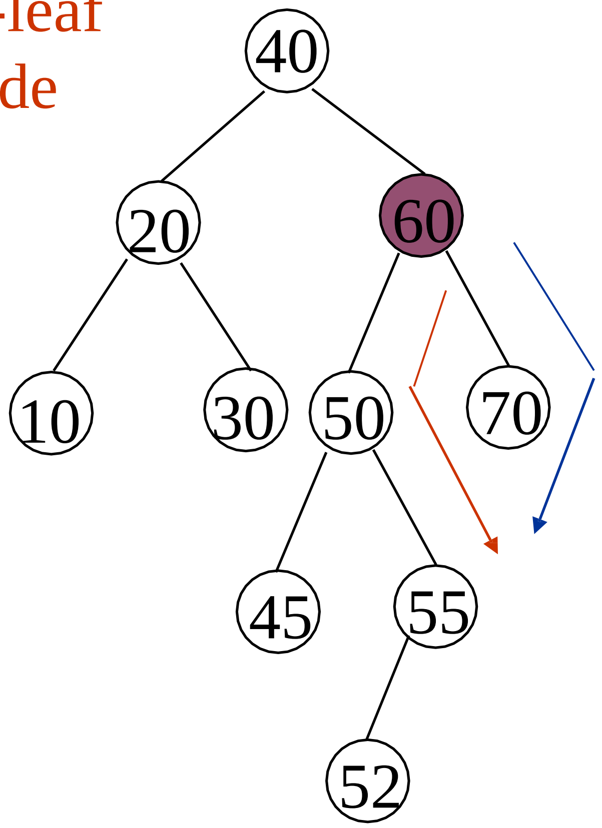
```
void insert_node(tree_pointer *node, int num)
{tree_pointer ptr,
       temp = modified_search(*node, num); if (temp ||
  !(*node)) {
   ptr = (tree_pointer) malloc(sizeof(node)); if (IS_FULL(ptr)) {
     fprintf(stderr, "The memory is full\n");
     exit(1);
   }
   ptr->data = num;
   ptr->left_child = ptr->right_child = NULL;
   if (*node)
     if (num<temp->data) temp->left_child=ptr; else temp-
        >right_child = ptr;
   else *node = ptr;
  }
}
```

Deletion for A Binary Search Tree

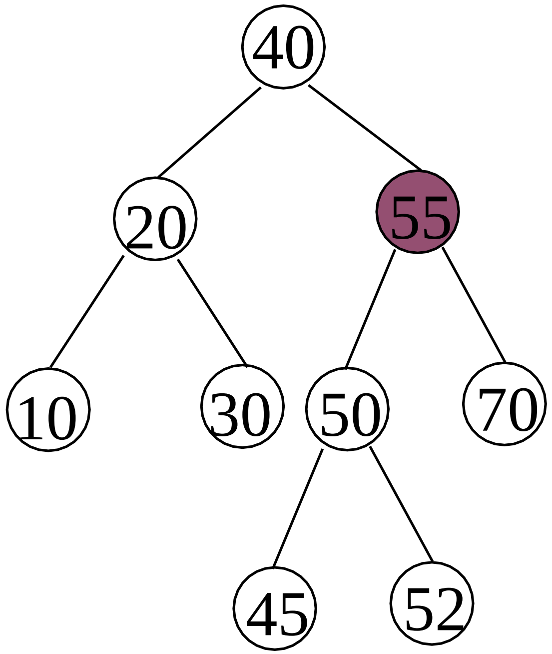


Deletion for A Binary Search Tree

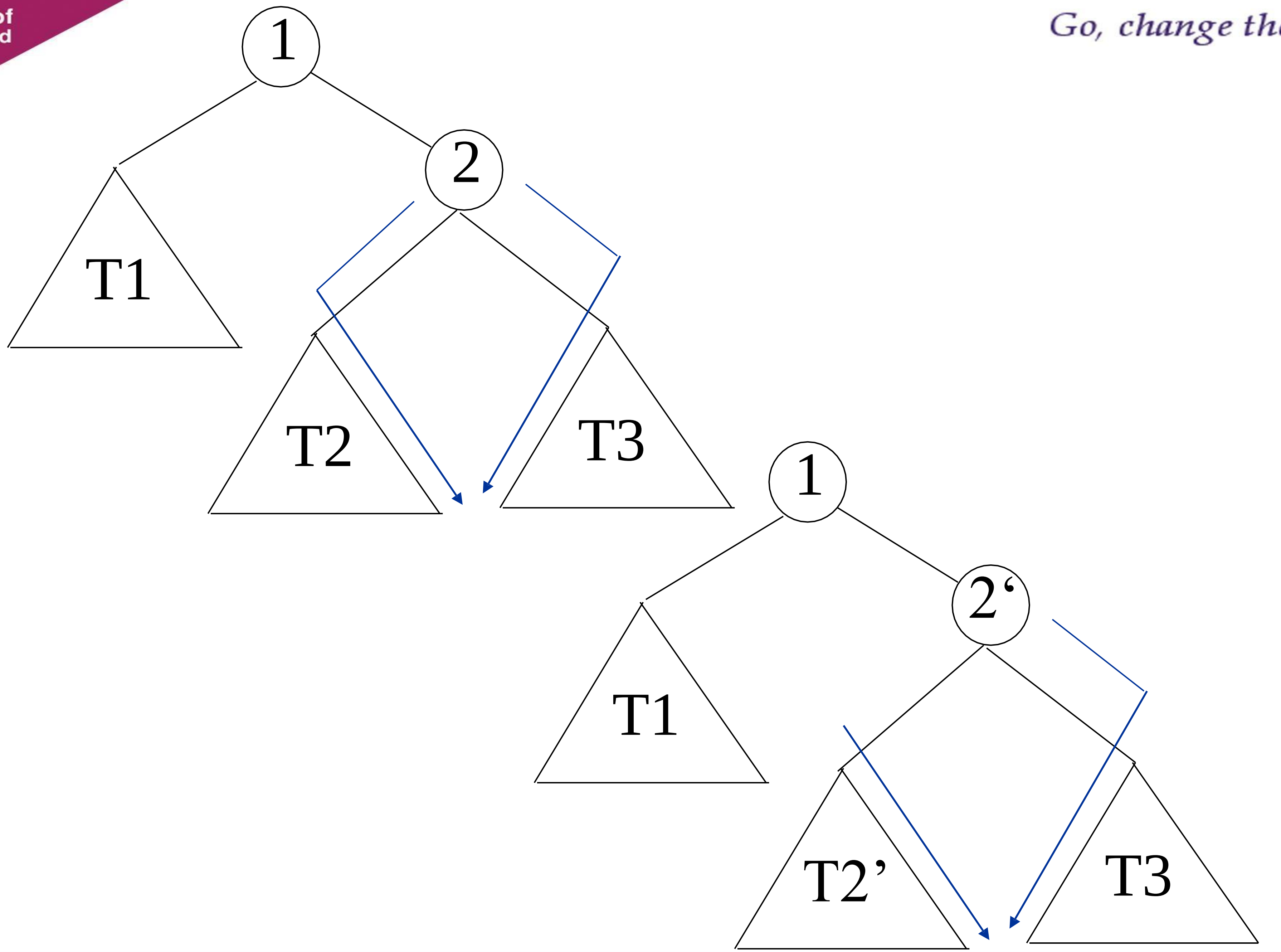
non-leaf node



Before deleting 60



After deleting 60

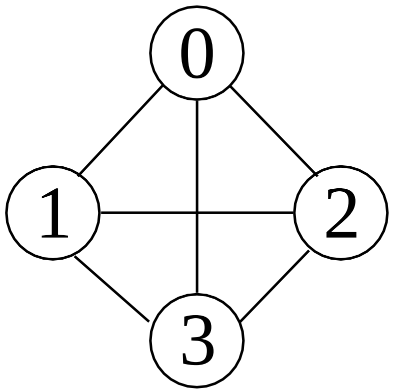

1
2
T1
T2
T3
1
2‘
T1
T2’
T3

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, $<v_0, v_1> \; != \; <v_1, v_0>$

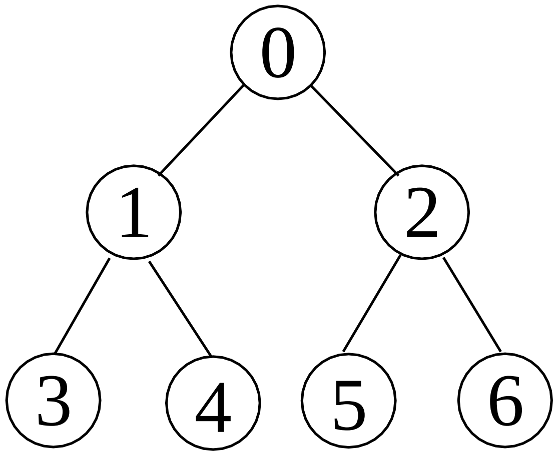
tail → head

Examples for Graph



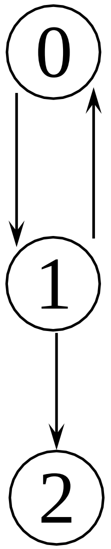
G_1

complete graph



G_2

incomplete graph



G_3

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

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

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

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

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

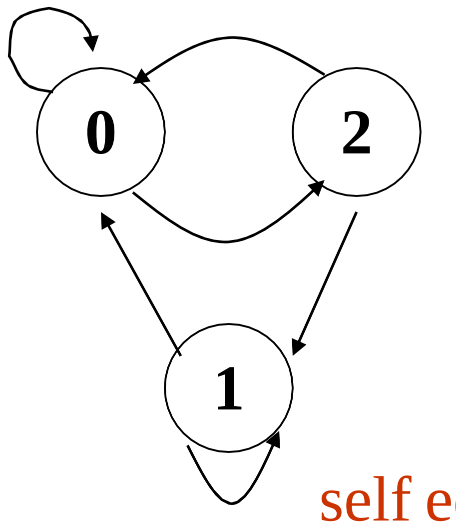
Complete Graph

- 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: G1 is a complete graph

Adjacent and Incident

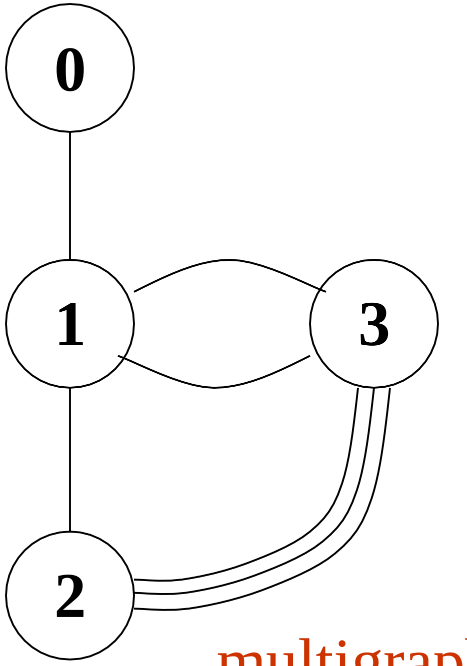
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

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



self edge

(a)



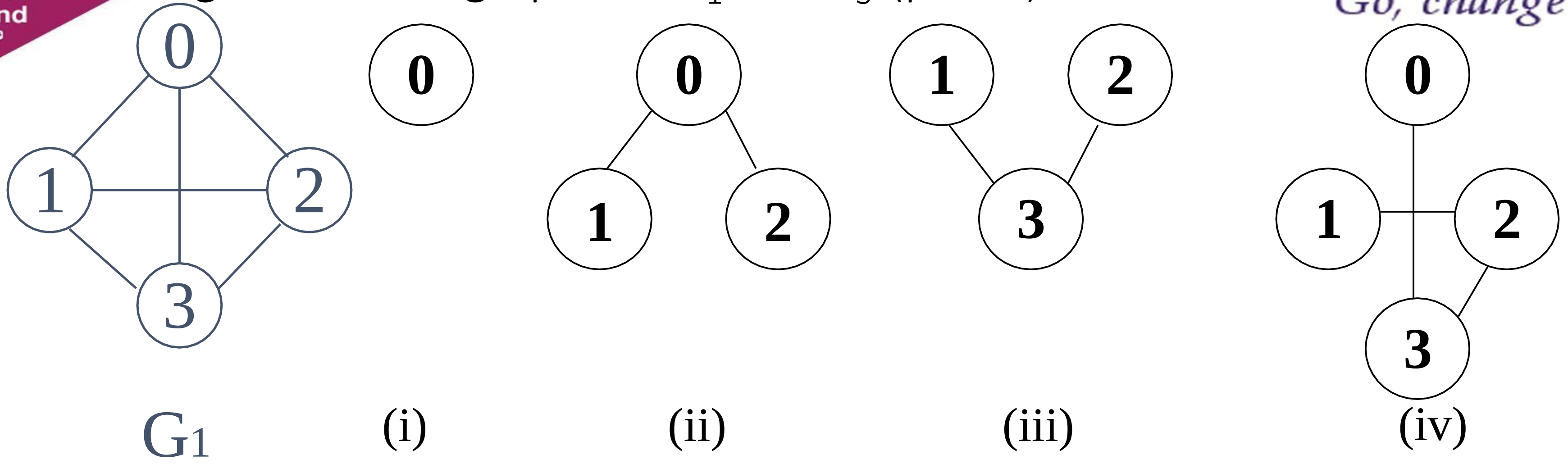
(b)

multigraph:
multiple occurrences of the same edge

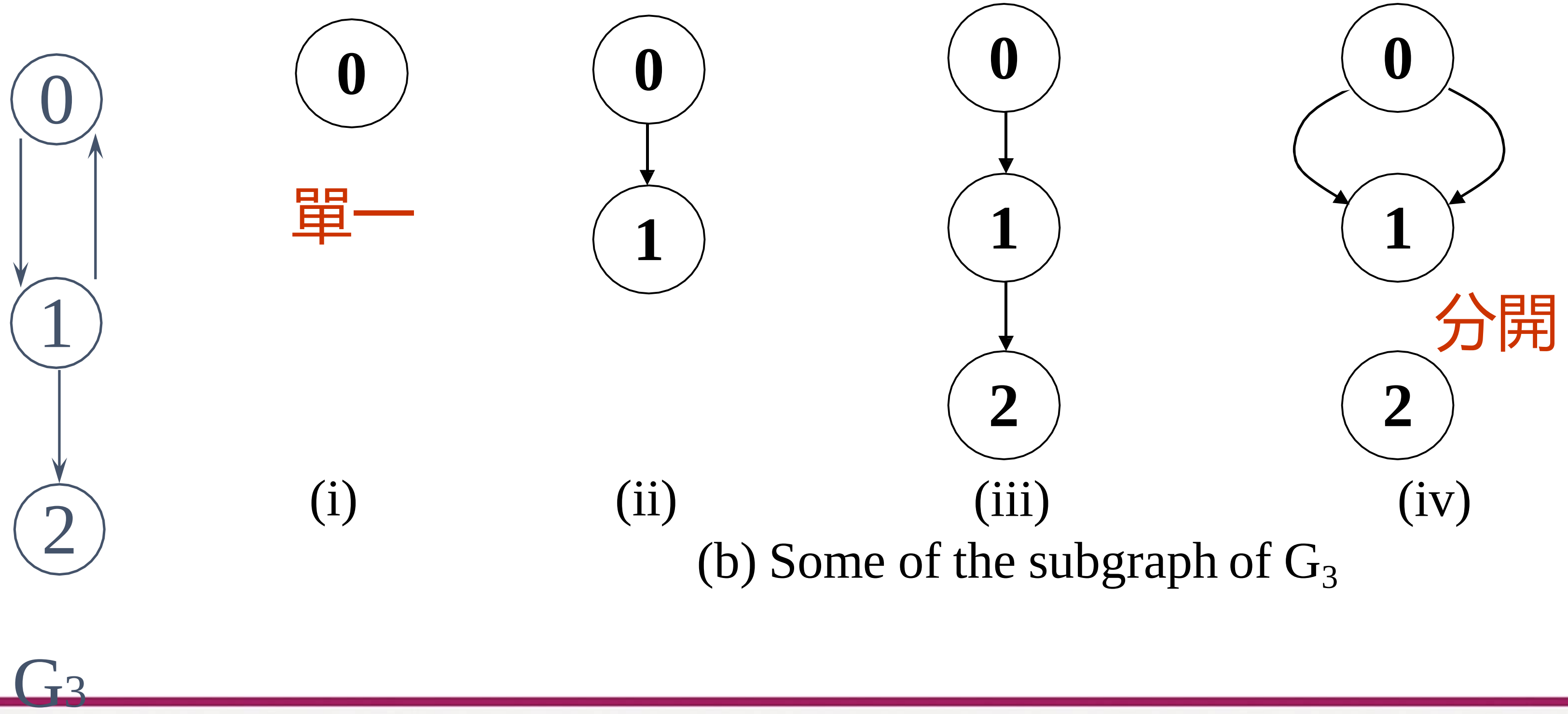
Subgraph and Path

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



(a) Some of the subgraph of G_1

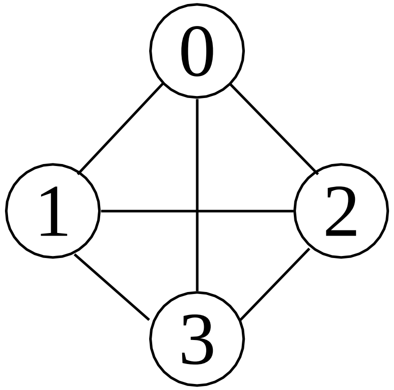


(b) Some of the subgraph of G_3

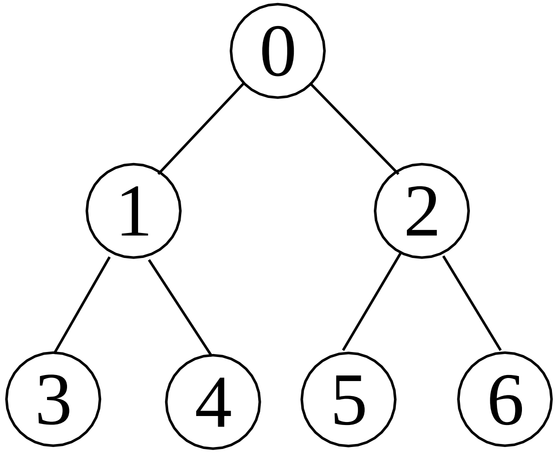
Simple Path and Style

- 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

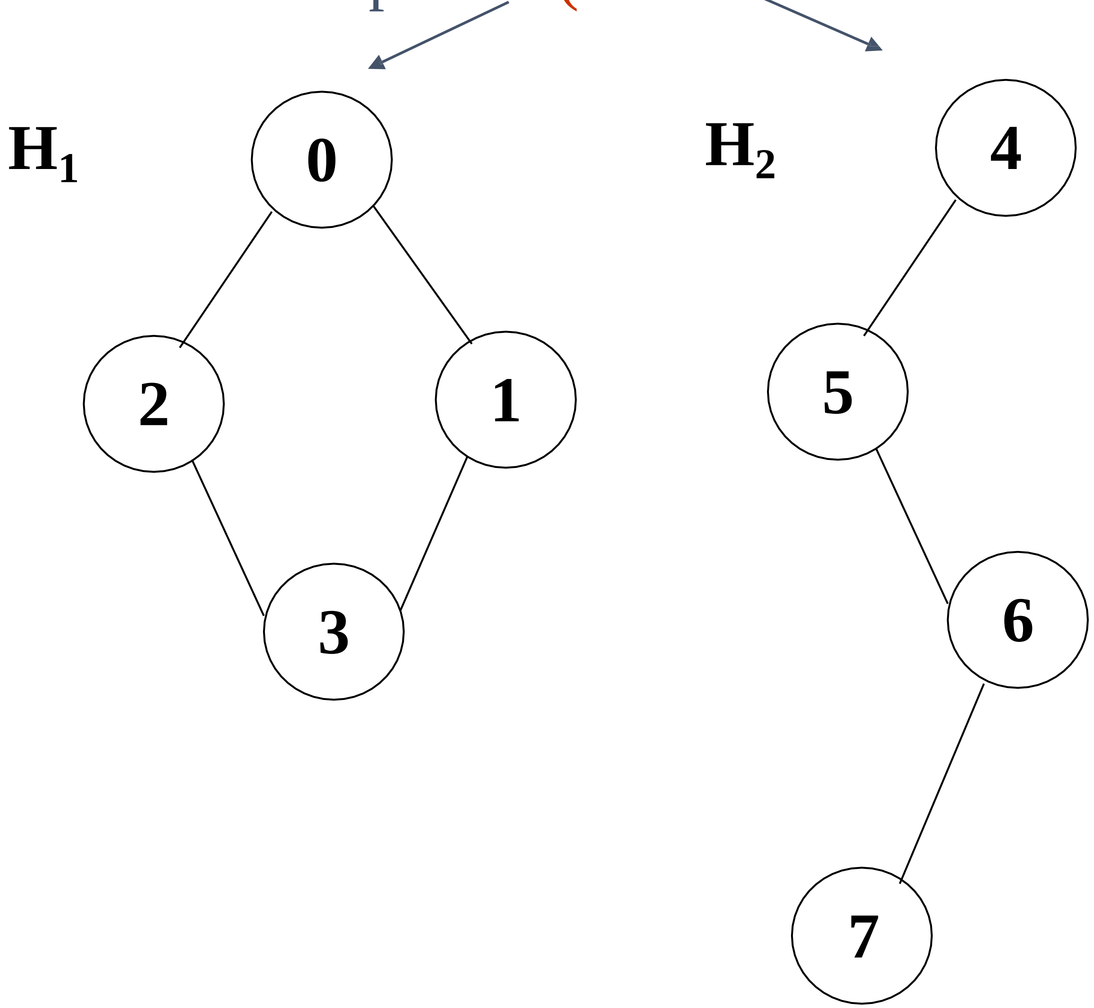
tree (acyclic graph)

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)

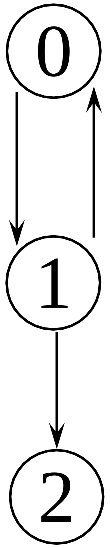
connected component (maximal connected subgraph)



G_4 (not connected)

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

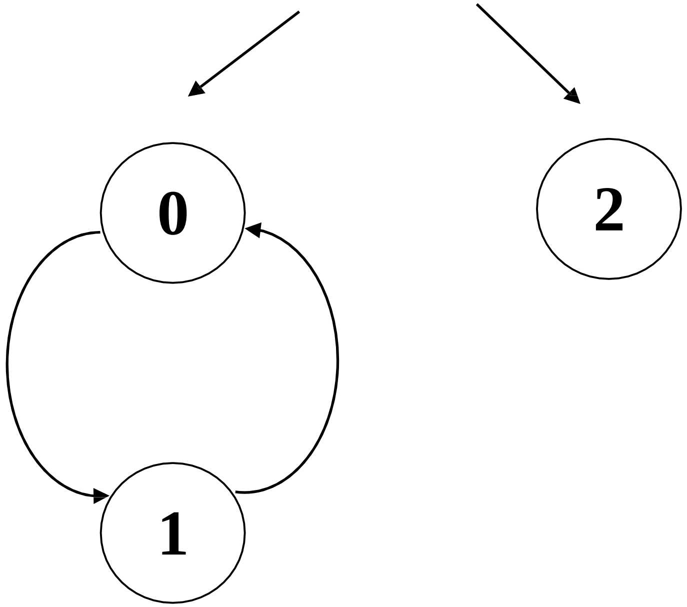
not strongly connected



G_3

strongly connected component

(maximal strongly connected subgraph)



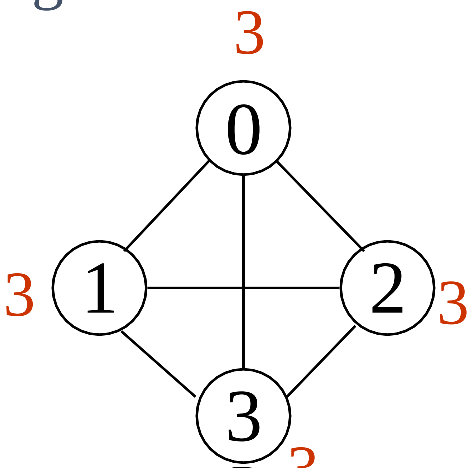
Degree

- 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

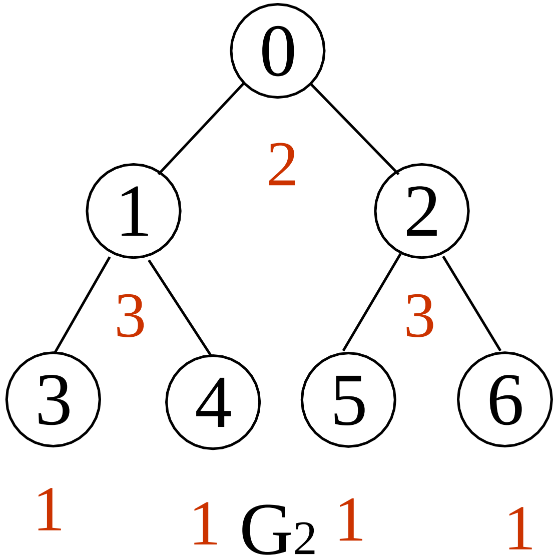
$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

undirected graph

degree



G_1

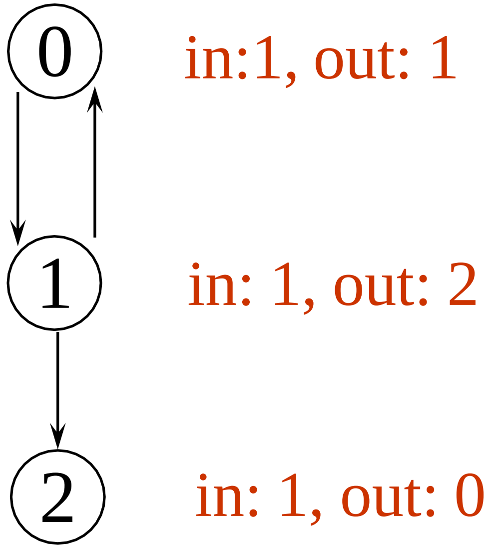


G_2

directed graph

in-degree

out-degree



G_3

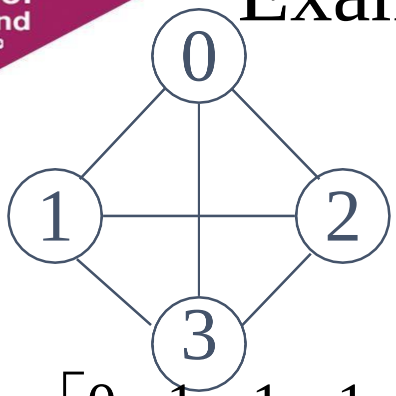
Graph Representations

- 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

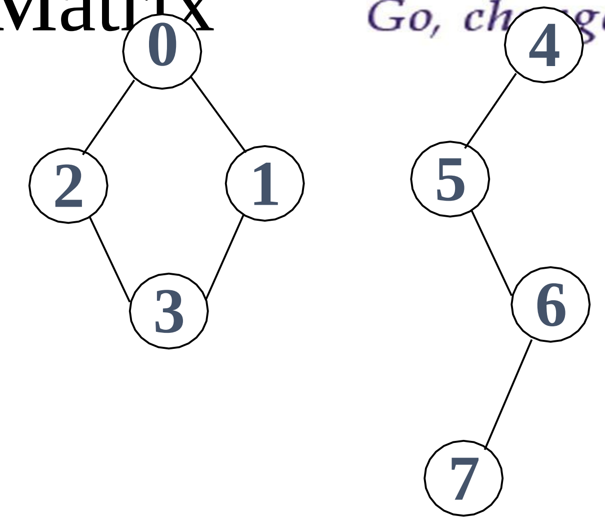


$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & \\ 1 & 1 & & \\ 1 & & & \end{bmatrix}$$

G1

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

G2



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

G4

symmetric

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

Merits of Adjacency Matrix

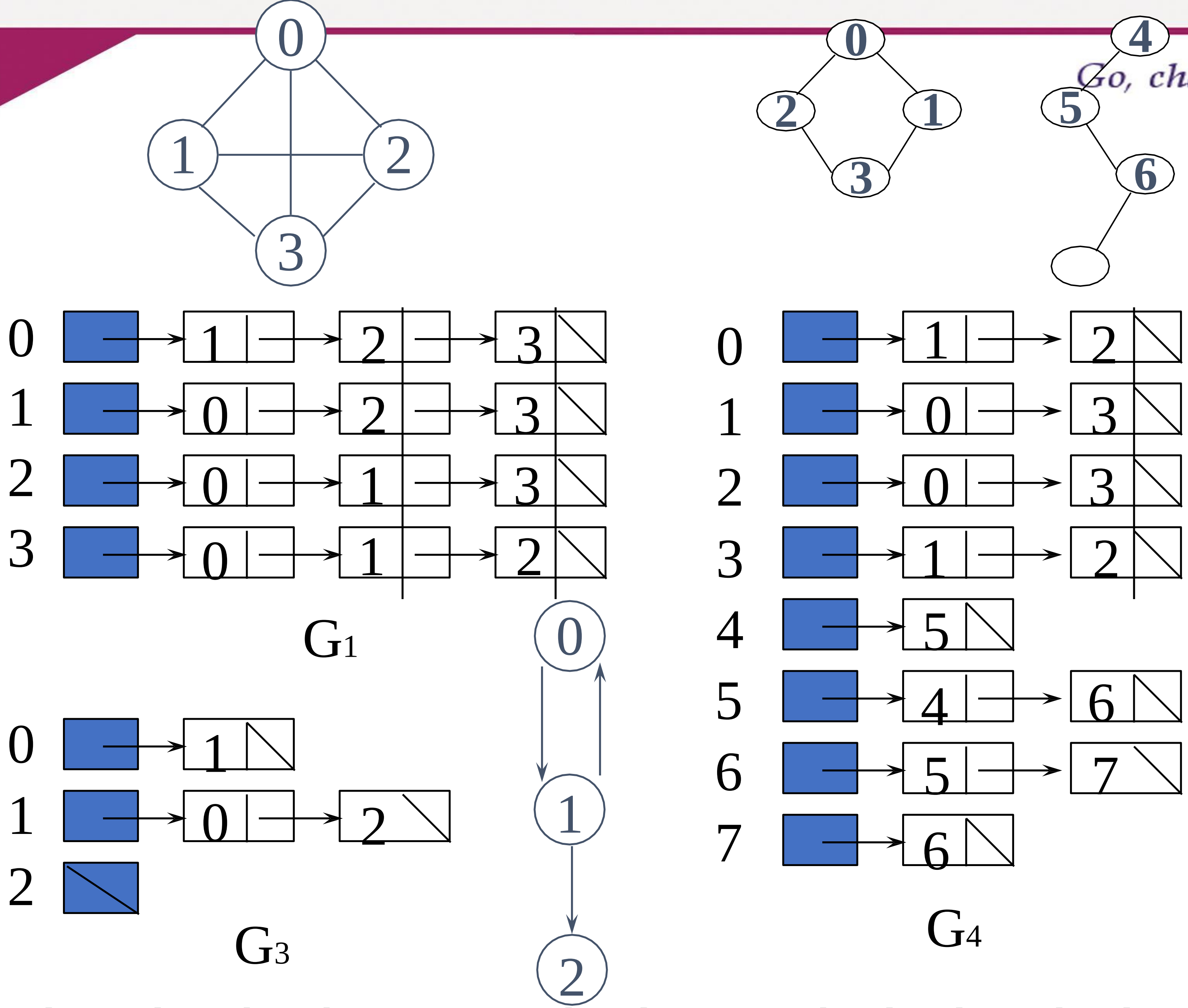
- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj\_m$
- For a digraph, the row sum is the out_degree, while the column sum is the in_degree

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

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 *
```

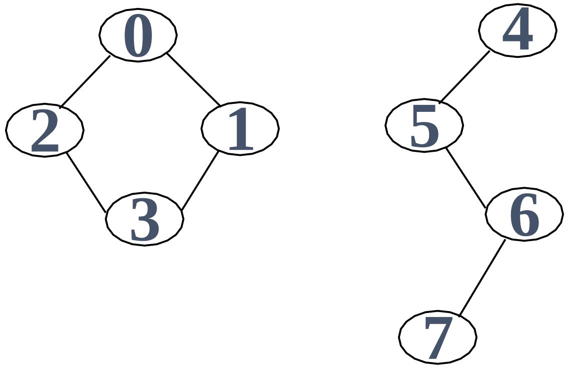


An undirected graph with n vertices and e edges ==> n head nodes and 2e list nodes

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

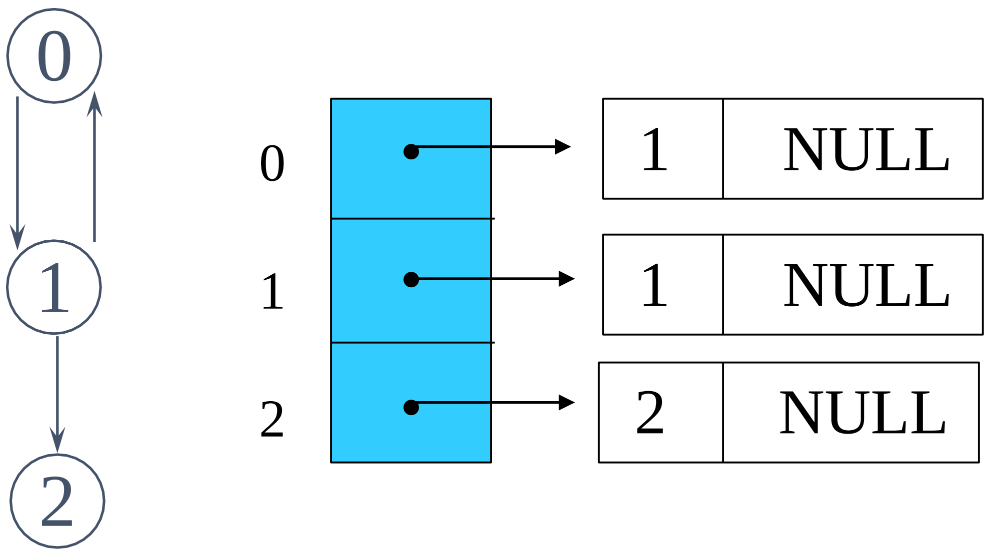
Compact Representation



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)

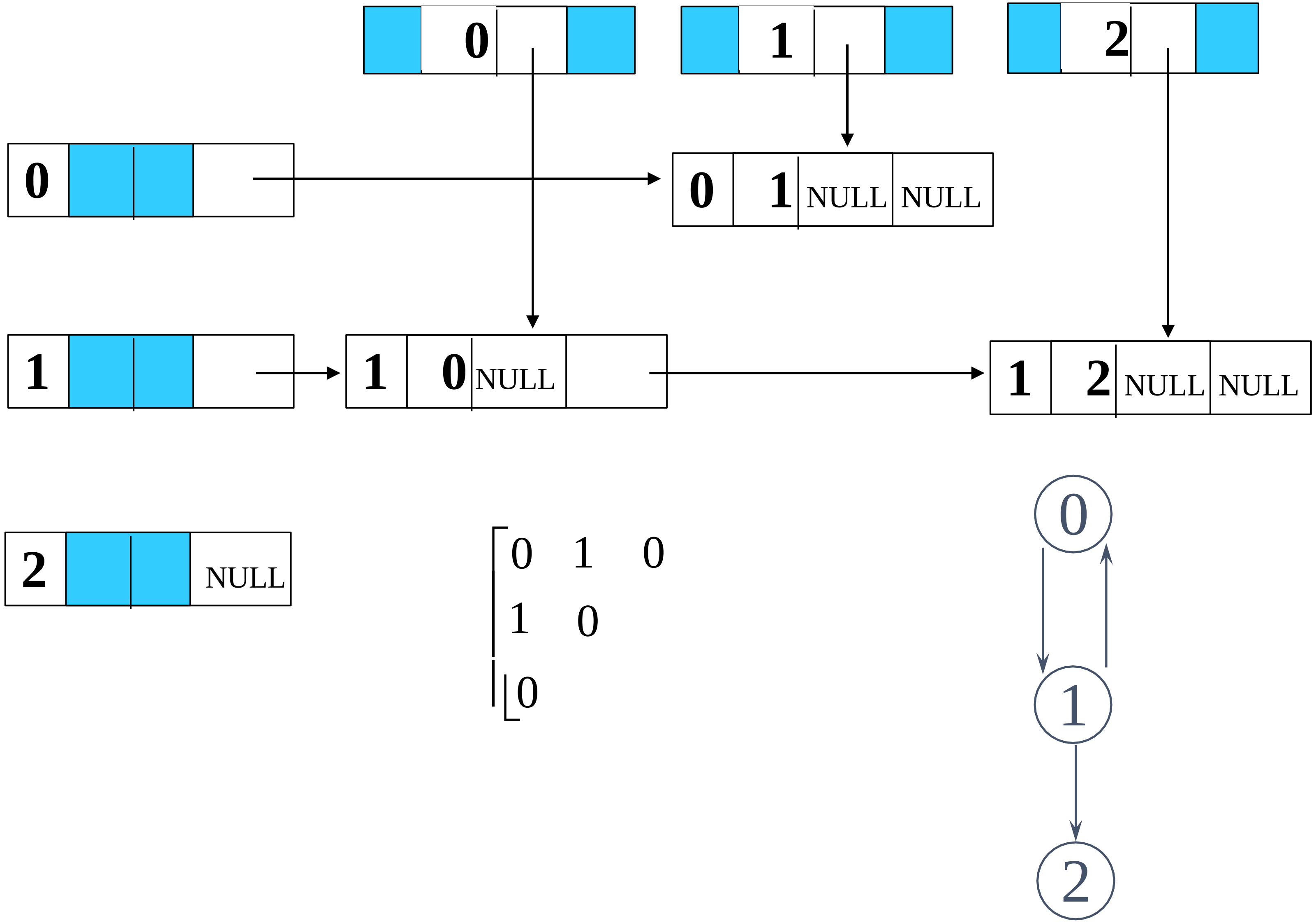
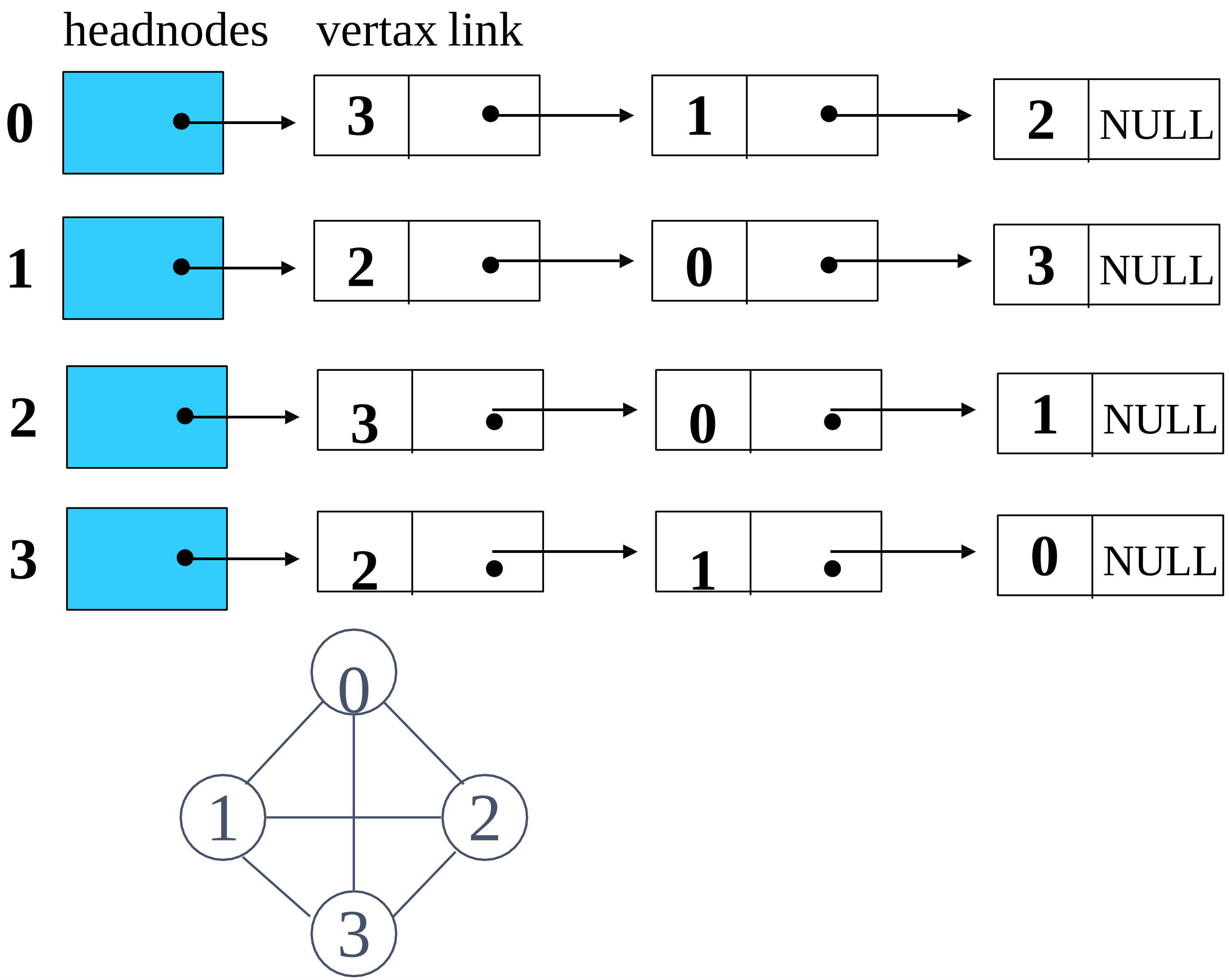


Figure 6.13: Alternate order adjacency list for G_1 (p.268)

Order is of no significance.



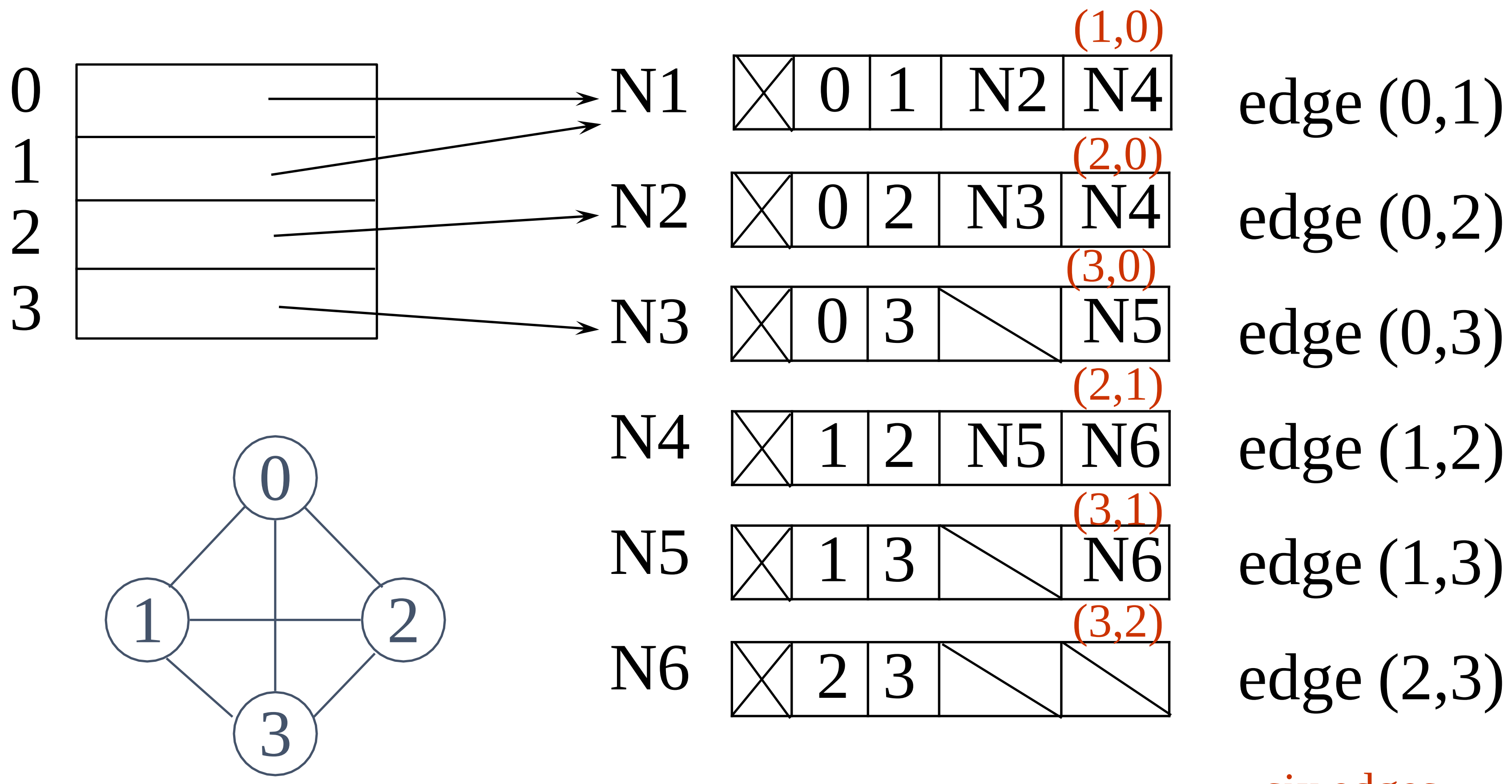
Adjacency Multilists

- 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



six edges

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∈V, such that w connects v.

  - Depth First Search (DFS)

    preorder tree traversal

  - Breadth First Search (BFS)

    level order tree traversal

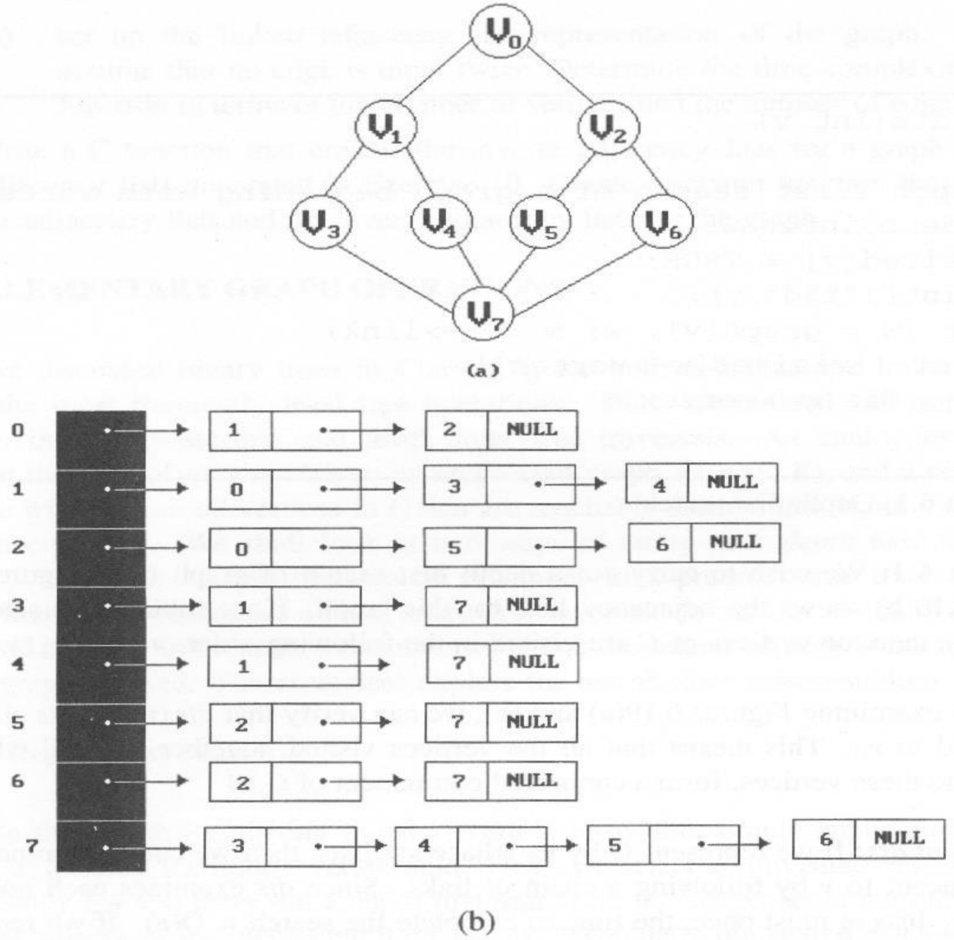
- Connected Components
- Spanning Trees

RV Institute of Technology and Management®

Go, change the world

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

depth first search: v0, v1, v3, v7, v4, v5, v2, v6



breadth first search: v0, v1, v2, v3, v4, v5, v6, v7

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: $O(n^2)$

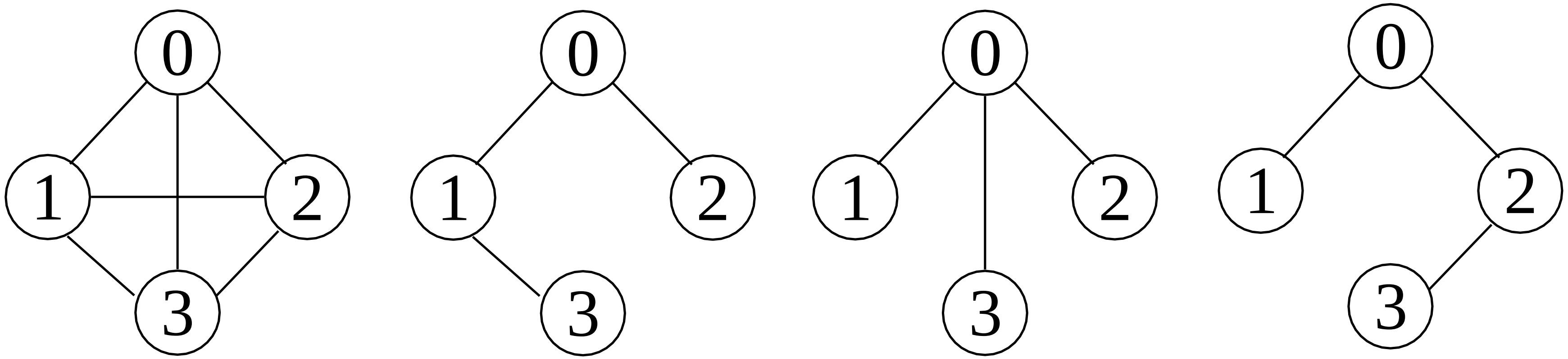
Spanning Trees

- When graph G is connected, a depth first or breadth first search starting at any vertex will visit all vertices in G
- A spanning tree is any tree that consists solely of edges in G and that includes all the vertices
- E(G): T (tree edges) + N (nontree edges)

  where T: set of edges used during search

  N: set of remaining edges

Examples of Spanning Tree



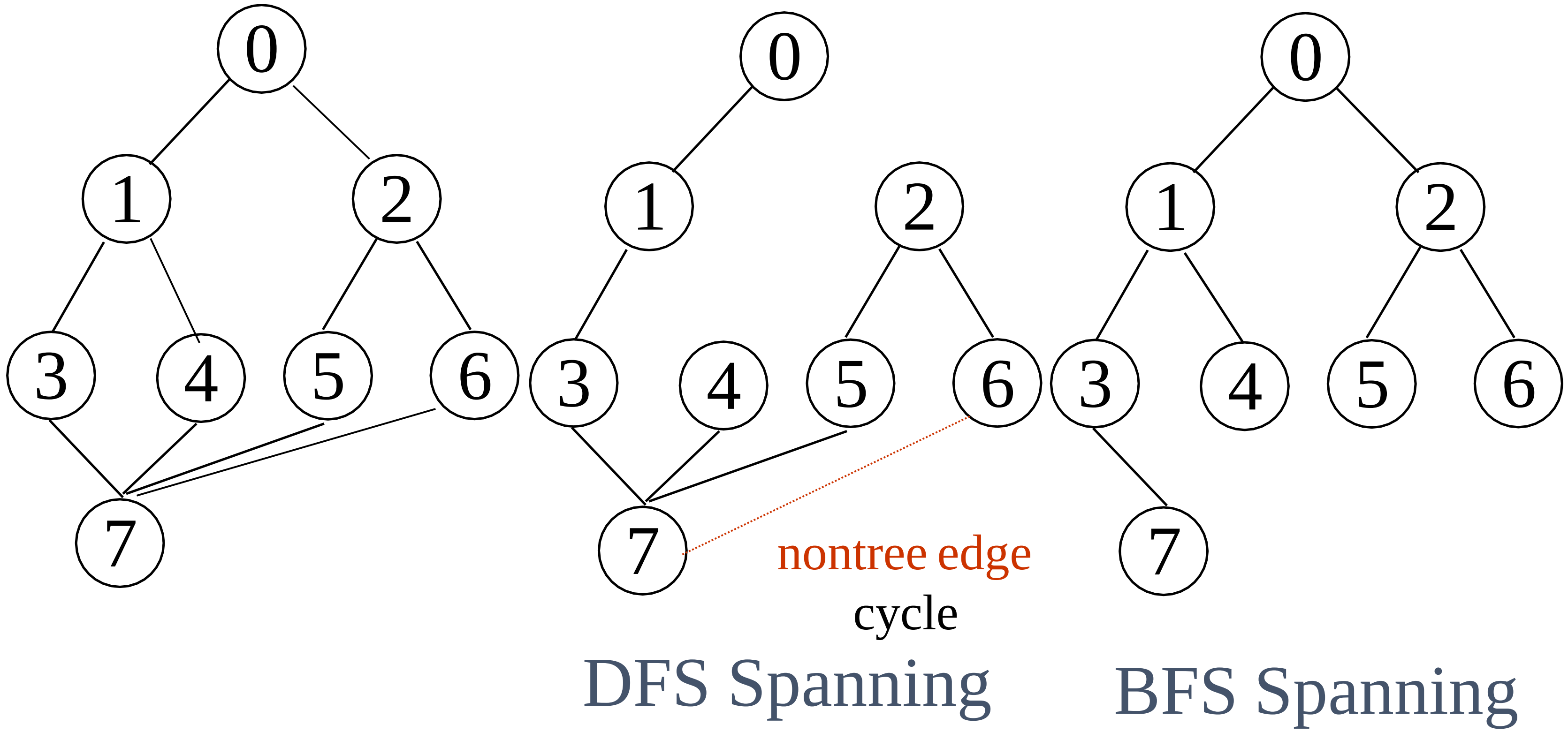
G_1

Possible spanning trees

Spanning Trees

- Either dfs or bfs can be used to create a spanning tree
  - When dfs is used, the resulting spanning tree is known as a depth first spanning tree
  - When bfs is used, the resulting spanning tree is known as a breadth first spanning tree
- While adding a nontree edge into any spanning tree, this will create a cycle

DFS VS BFS Spanning Tree

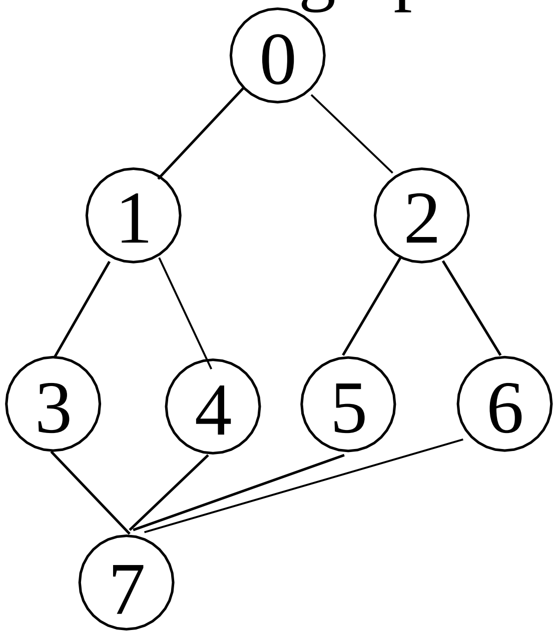


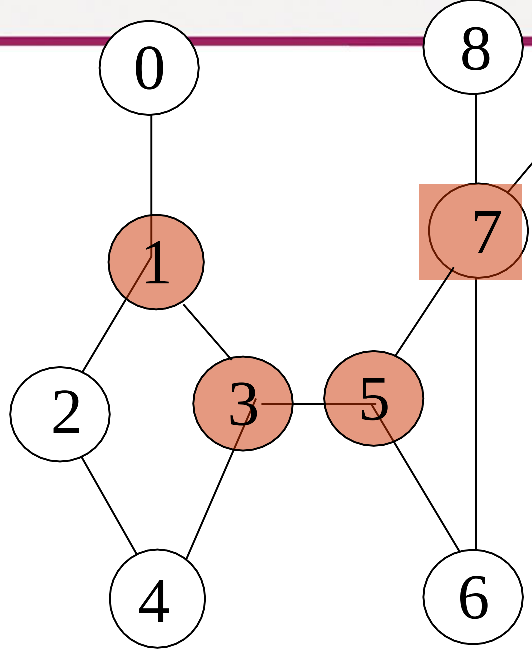
A spanning tree is a minimal subgraph, G’, of G such that V(G’)=V(G) and G’ is connected.

Any connected graph with n vertices must have at least n-1 edges.

A biconnected graph is a connected graph that has no articulation points.

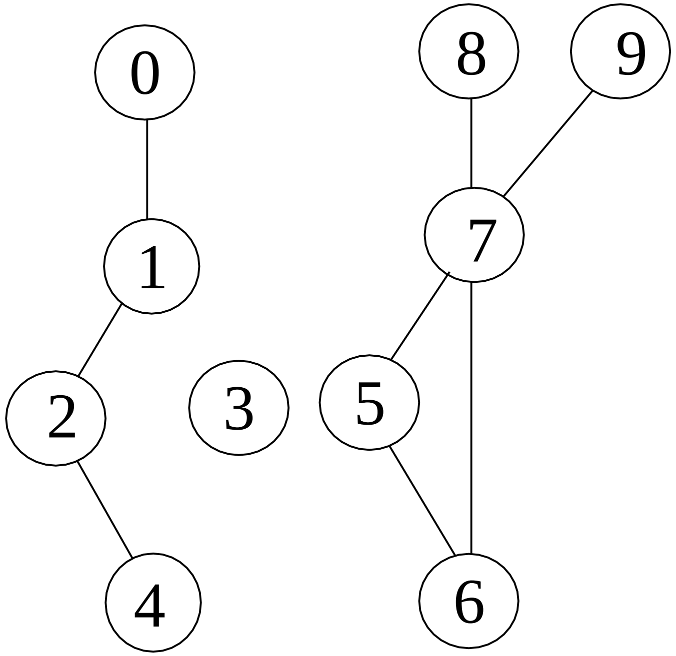
biconnected graph



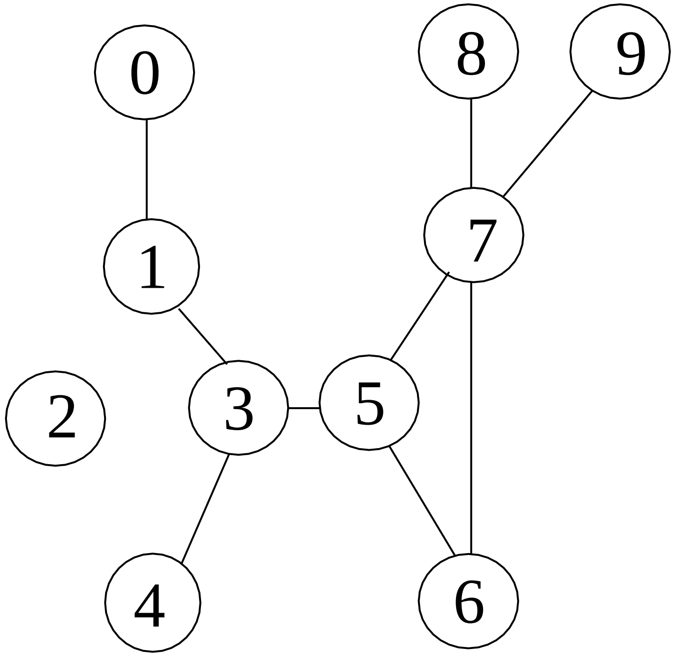


connected graph

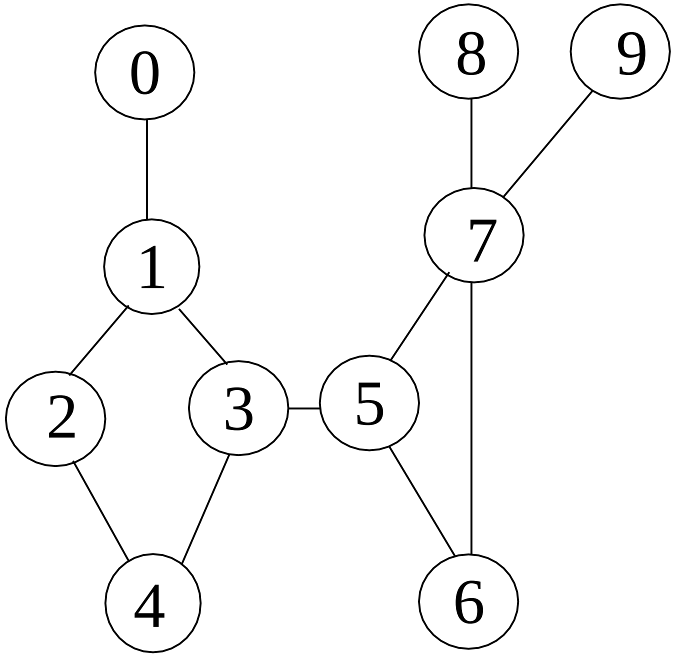
two connected components

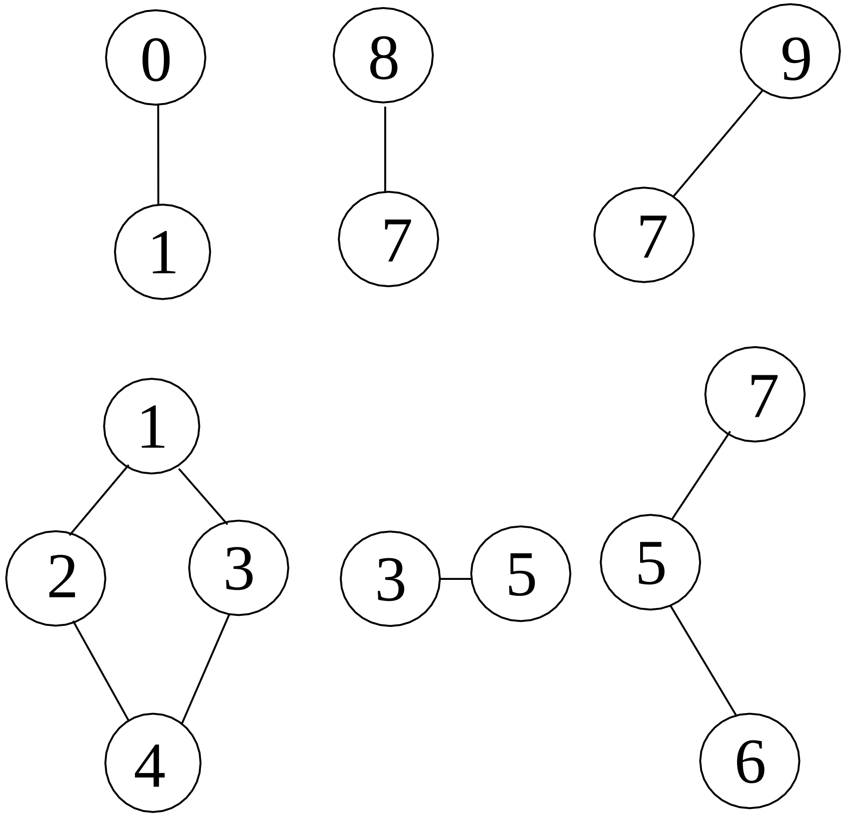


one connected graph



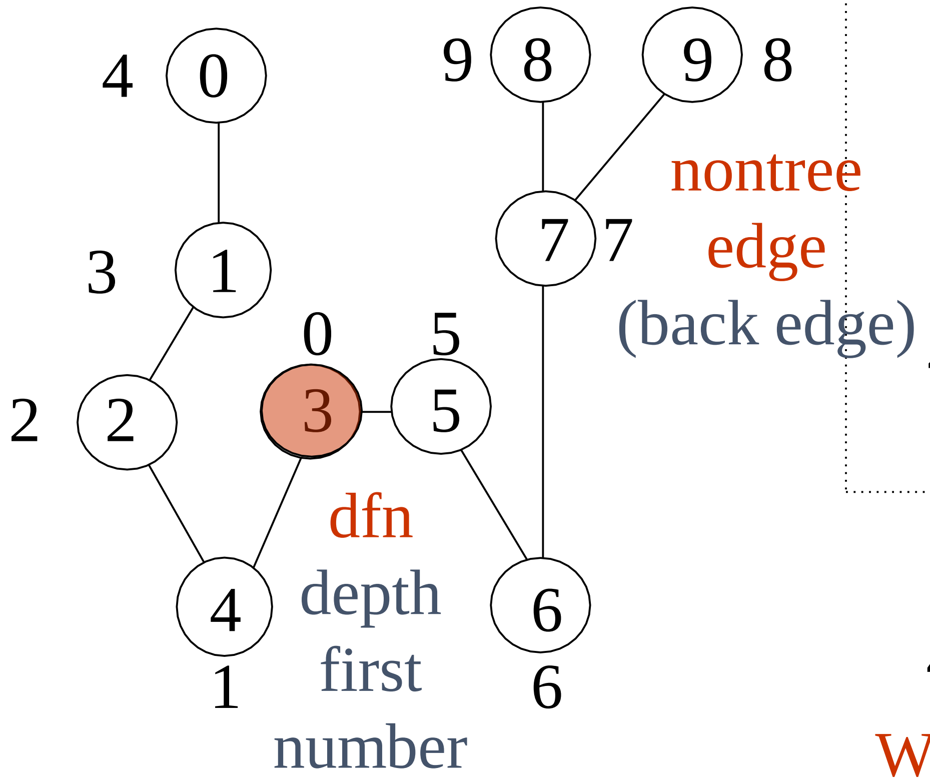
biconnected component: a maximal connected subgraph H
(no subgraph that is both biconnected and properly contains H)



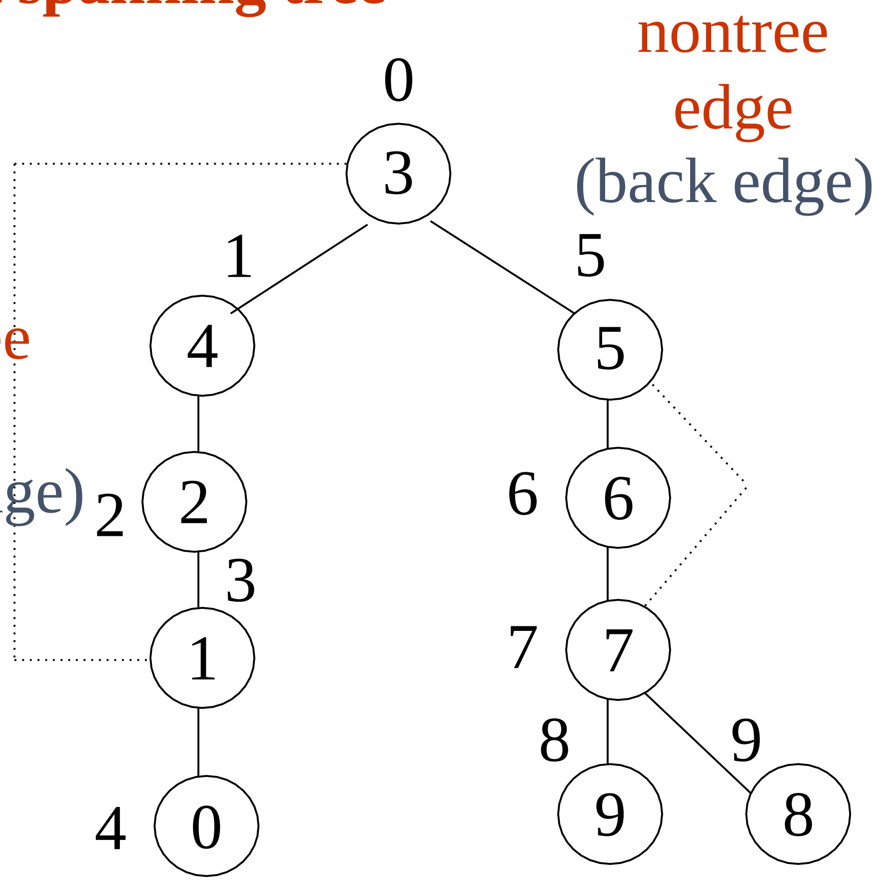


biconnected components

Find biconnected component of a connected undirected graph by depth first spanning tree



(a) depth first spanning tree



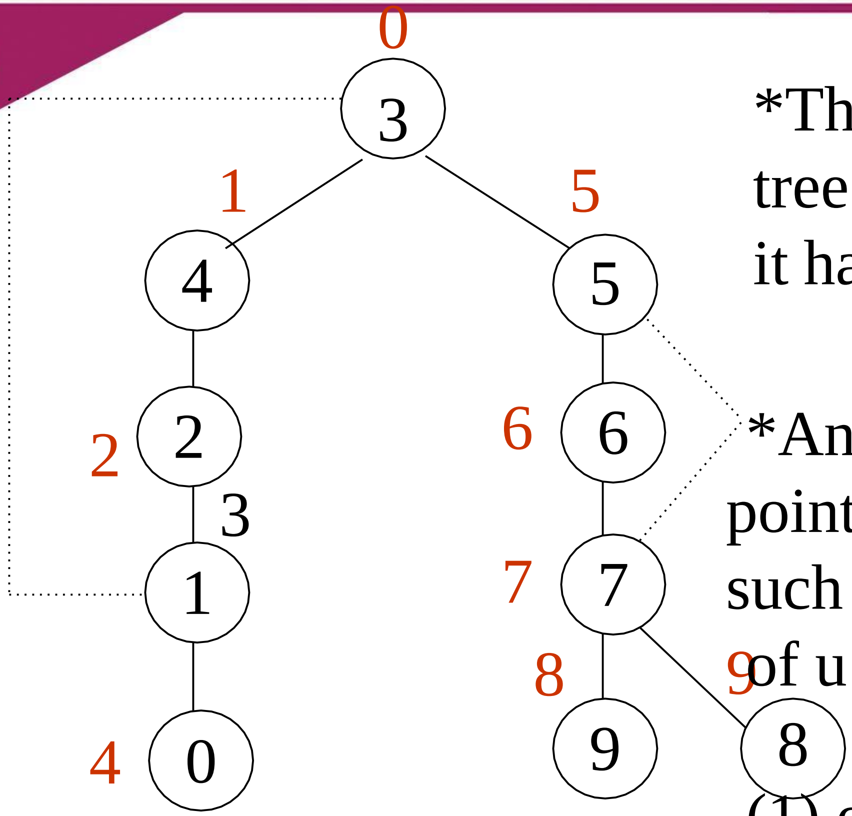
Why is cross edge impossible?

(b)

If u is an ancestor of v then dfn(u) < dfn(v).

***Figure 6.24**: *dfn* and *low* values for *dfs* spanning tree with *root* =3(p.281)*

Vertax	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



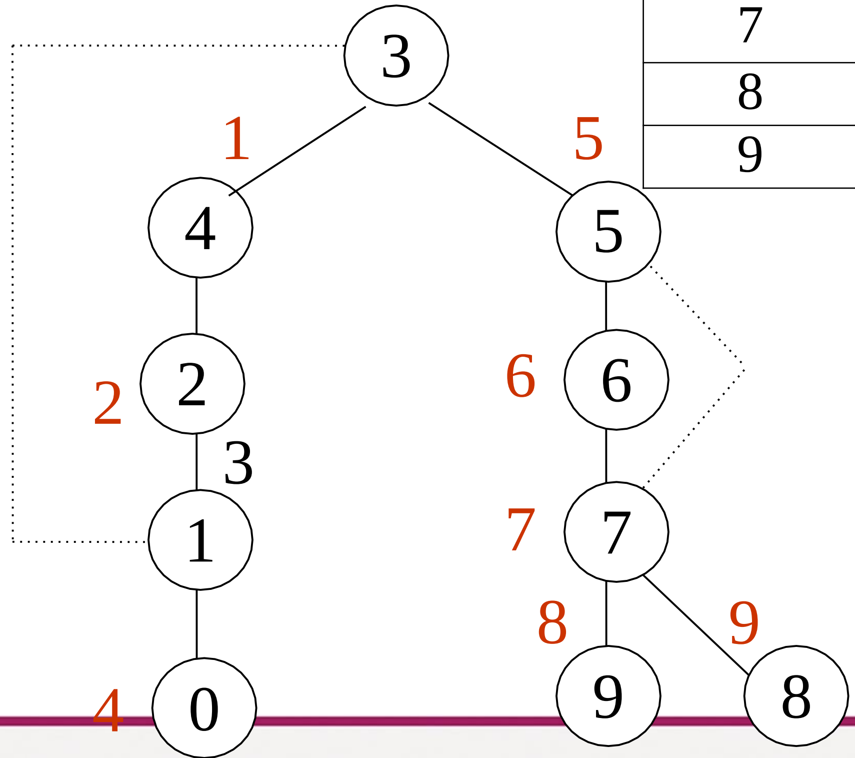
*The root of a depth first spanning tree is an articulation point iff it has at least two children.

*Any other vertex u is an articulation point iff it has at least one child w such that we cannot reach an ancestor of u using a path that consists of (1) only w (2) descendants of w (3) single 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}

u: articulation point
low(child) ≥ dfn(u)

vertex	dfn	low	child	low_child	low:dfn
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



*Program 6.5: Initializaiton of *dfn* and *low* (p.282)

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

*Program 6.4: Determining *dfn* and *low* (p.282)

Initial call: dfn(x,-1)

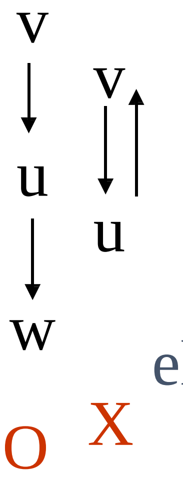
```
void dfnlow(int u, int v)
{
/* compute dfn and low while performing a dfs search
   beginning at vertex u, v is the parent of u (if any) */
      node_pointer ptr;
      int w;
      dfn[u] = low[u] = num++;
      for (ptr = graph[u]; ptr; ptr = ptr ->link) {
         w = ptr ->vertex;
         if (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] );
   }
}
```

low[u]=min{dfn(u), …}

low[u]=min{…, min{low(w)|w is a child of u}, …}

dfn[w]≠0 非第一次，表示藉back edge

low[u]=min{…,…,min{dfn(w)|(u,w) is a back edge}



*Program 6.6: Biconnected components of a graph (p.283)

```
void bicon(int u, int v)
{
/* compute dfn and low, and output the edges of G by their
  biconnected  components , v is the parent ( if any) of the u
  (if any)  in the resulting spanning tree. It is assumed that all
  entries  of dfn[ ] have been initialized to -1, num has been
  initialized to 0,  and the stack has been set to empty */
     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 */
```

low[u]=min{dfn(u), …}

(1) dfn[w]=-1 第一次

(2) dfn[w]!=-1非第一次，藉back edge

```
if(dfn[w] < 0) {
      bicon(w, u);
      low[u] = MIN2(low[u], low[w]);
      if (low[w] >= dfn[u] ){ articulation point
          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]);
  }
}
```

low[u]=min{…, …, min{dfn(w)|(u,w) is a back edge}}

Minimum Cost Spanning Tree

- The cost of a spanning tree of a weighted undirected graph is the sum of the costs 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
  - Kruskal
  - Prim
  - Sollin

Select n-1 edges from a weighted graph of n vertices with minimum cost.

Greedy Strategy

- An optimal solution is constructed in stages
- At each stage, the best decision is made at this time
- Since this decision cannot be changed later, we make sure that the decision will result in a feasible solution
- Typically, the selection of an item at each stage is based on a least cost or a highest profit criterion

Kruskal’s Idea

Build a minimum cost spanning tree T by adding edges to T one at a time

Select the edges for inclusion in T in nondecreasing 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

Examples for Kruskal’s Algorithm

0 —10— 5

2 —12— 3

1 —14— 6

1 —16— 2

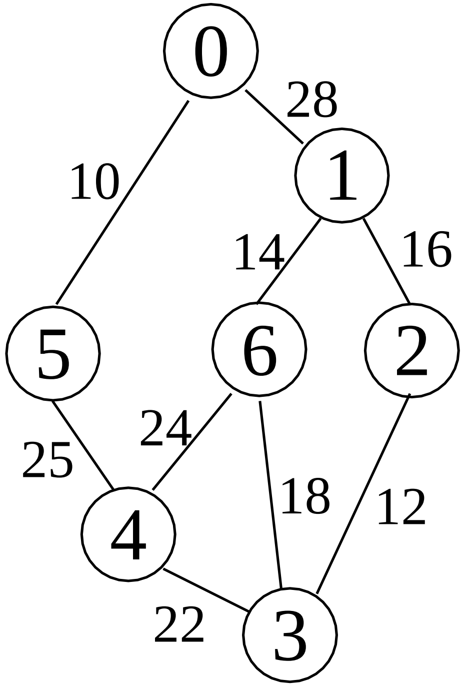
3 —18— 6

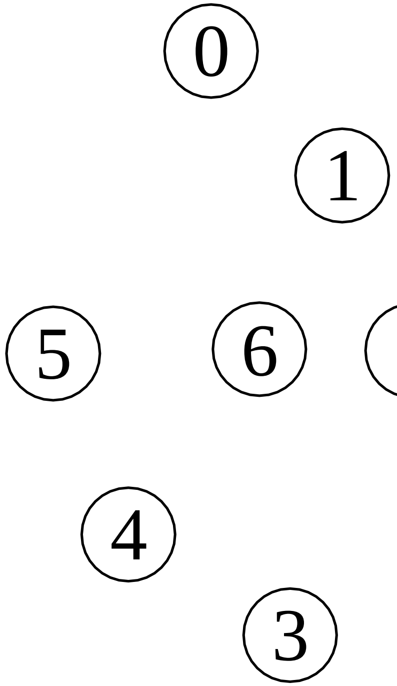
3 —22— 4

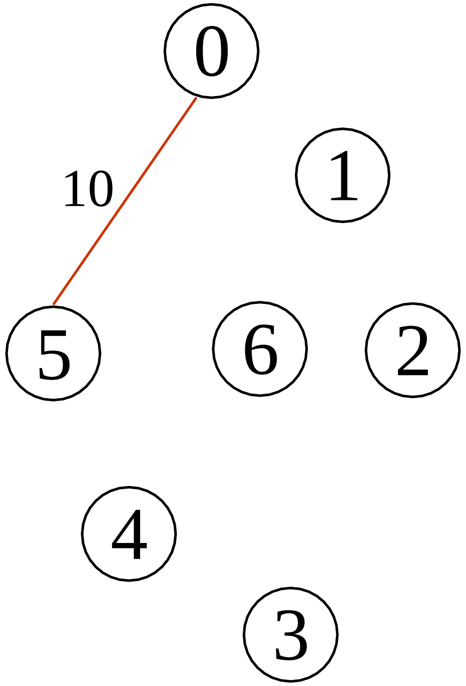
4 —24— 6

4 —25— 5

0 —28— 1







6/9

0 —10— 5

2 —12— 3

1 —14— 6

1 —16— 2

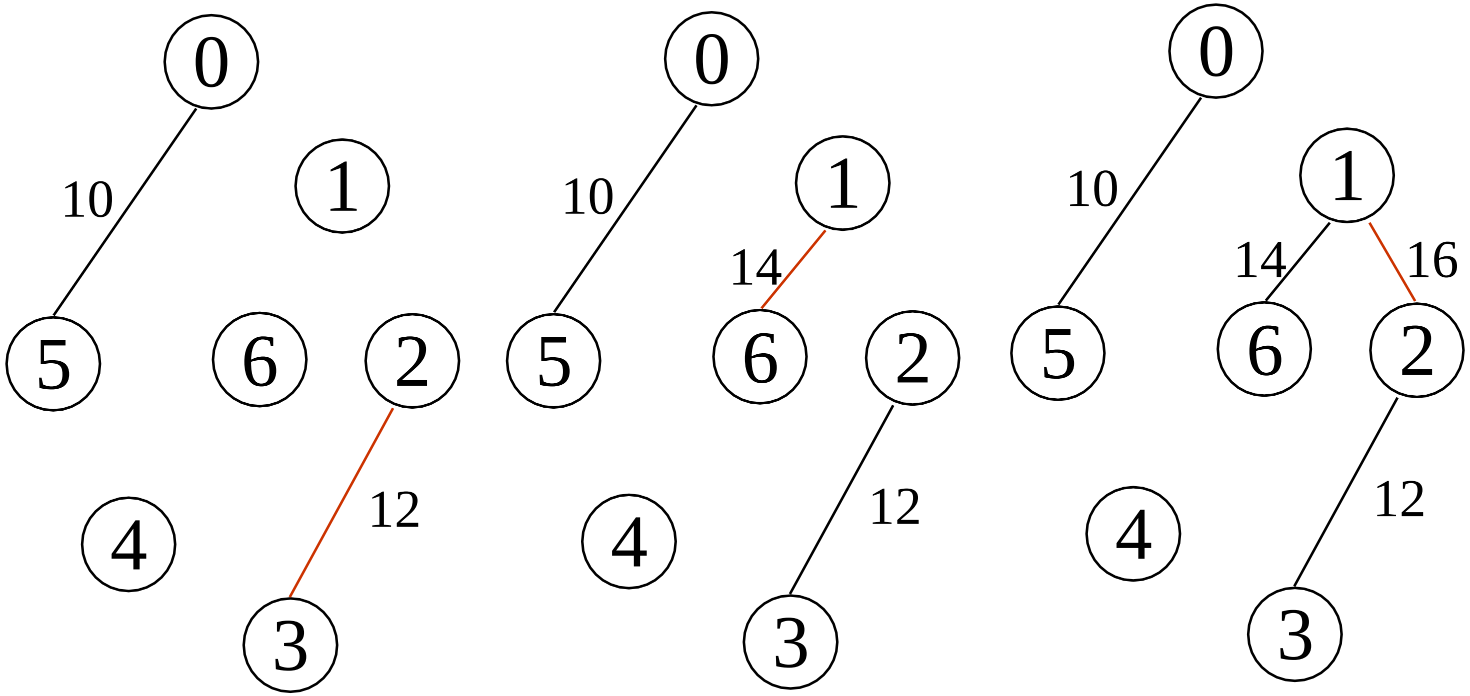
3 —18— 6

3 —22— 4

4 —24— 6

4 —25— 5

0 —28— 1



+ 3 —— 6

cycle

0 — 10 — 5

2 — 12 — 3

1 — 14 — 6

1 — 16 — 2

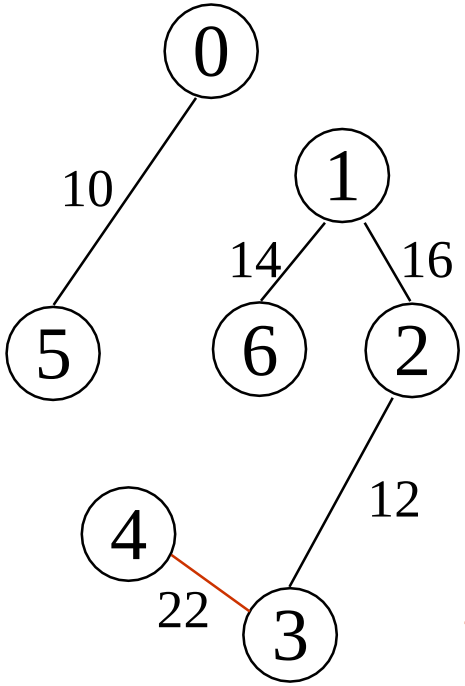
3 — 18 — 6

3 — 22 — 4

4 — 24 — 6

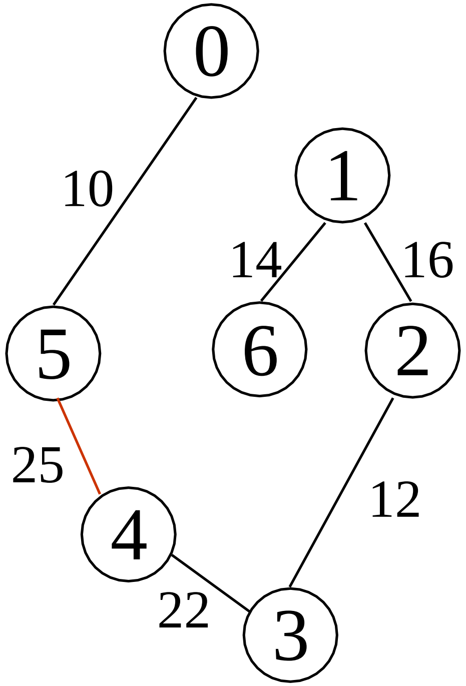
4 — 25 — 5

0 — 28 — 1



+ 4 — 6

cycle



cost = 10 +25+22+12+16+14

Kruskal’s Algorithm

```
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”);
```

min heap construction time O(e)

choose and delete O(log e)

find find & union O(log e)

{0,5}, {1,2,3,6}, {4} + edge(3,6) X + edge(3,4) --> {0,5},{1,2,3,4,6}

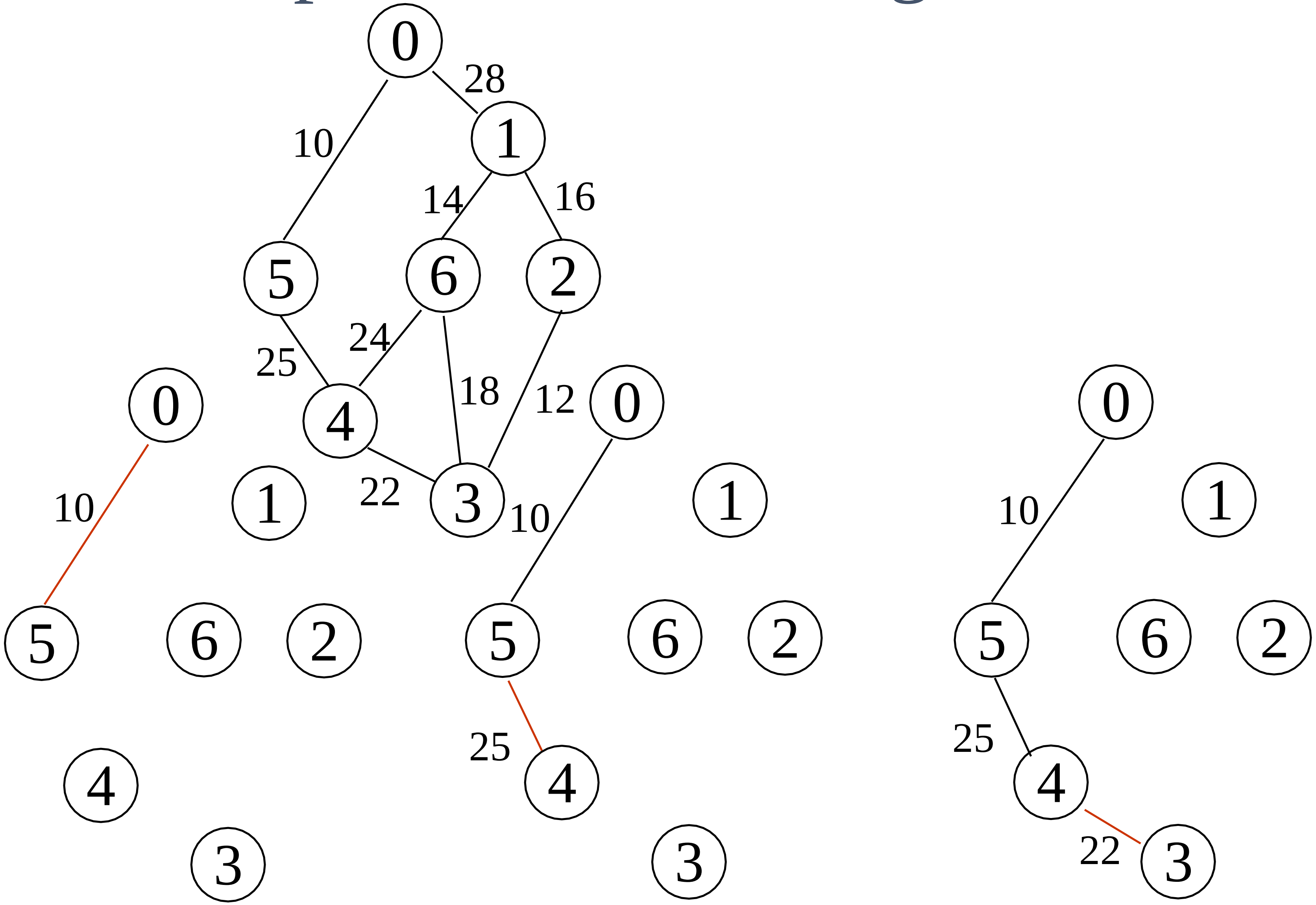
O(e log e)

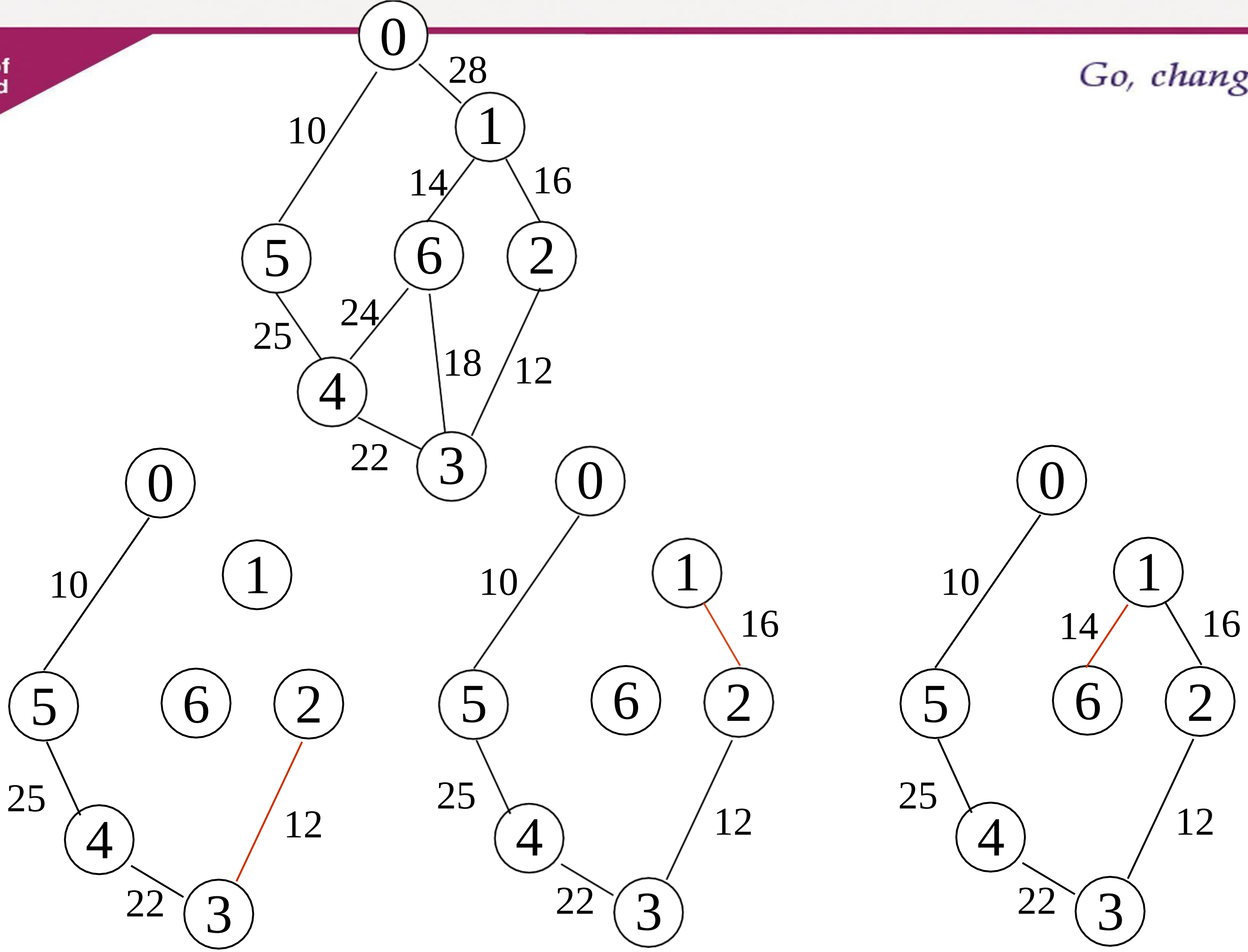
Prim’s Algorithm

(tree all the time vs. forest)

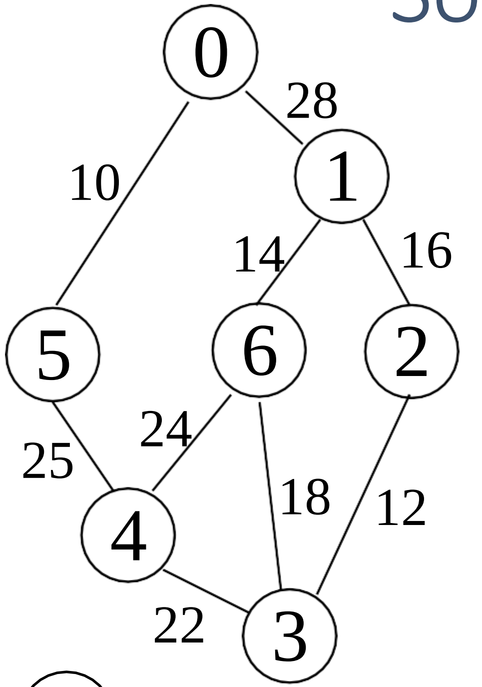
```
T={};
TV={0};
while (T contains fewer than n-1 edges)
{
   let (u,v) be a least cost edge such that u ∈  and
      v ∉
   if (there is no such edge ) break; add v to TV;
   add (u,v) to T;
}
if (T contains fewer than n-1 edges)
   printf(“No spanning tree\n”);
```

Examples for Prim’s Algorithm

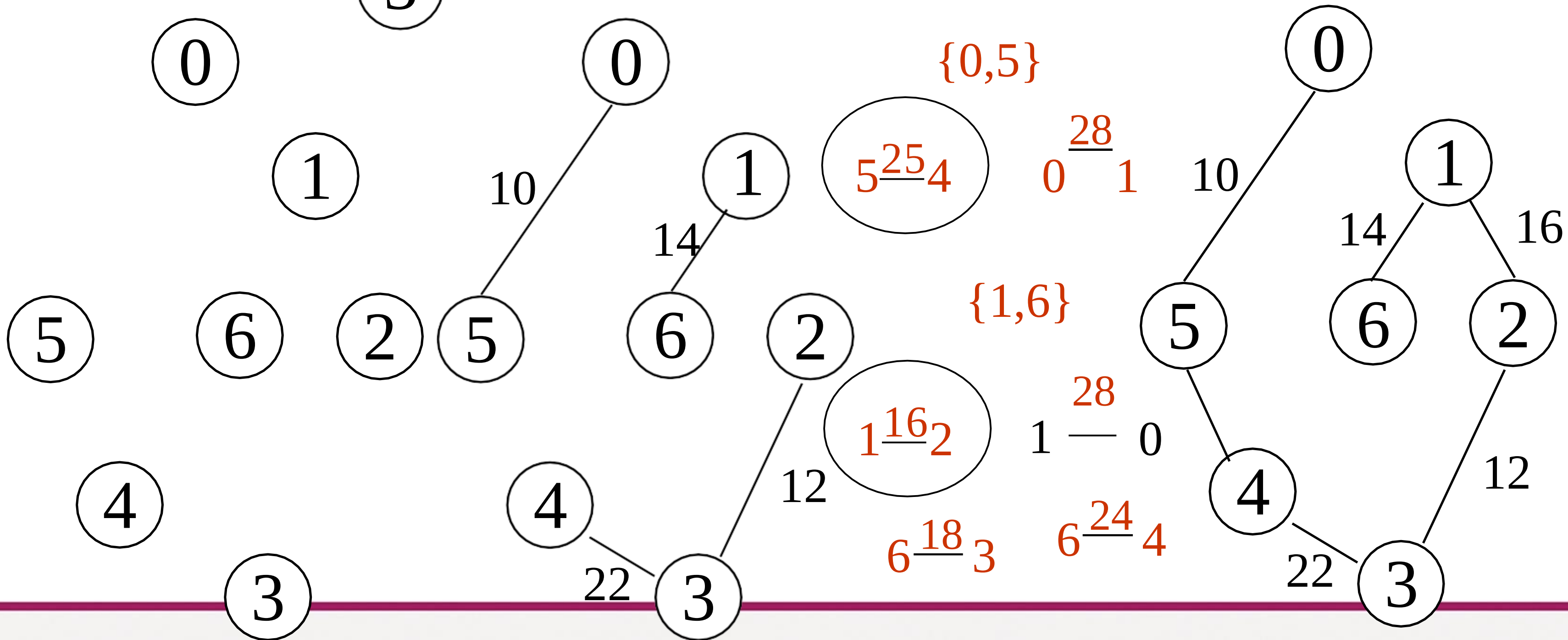



0
28
1
10
14
16
5
6
2
24
25
18
12
4
22
3
0
10
1
5
6
2
25
12
4
22
3
0
10
1
16
5
6
2
25
12
4
22
3
0
10
1
14
16
5
6
2
25
12
4
22
3

Sollin's Algorithm



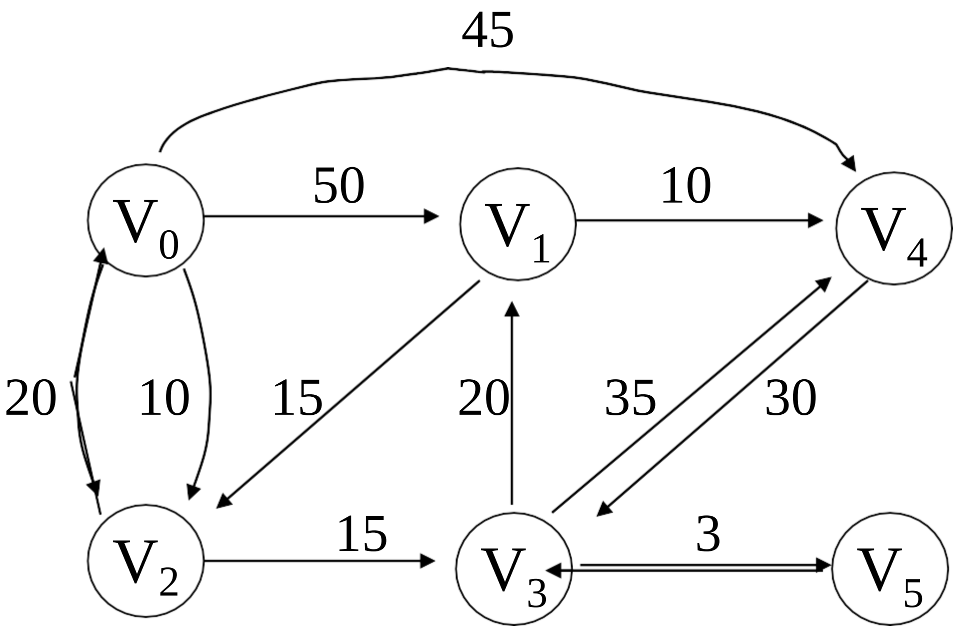
vertex	edge
0	0 -- 10 --> 5, 0 -- 28 --> 1
1	1 -- 14 --> 6, 1-- 16 --> 2, 1 -- 28 --> 0
2	2 -- 12 --> 3, 2 -- 16 --> 1
3	3 -- 12 --> 2, 3 -- 18 --> 6, 3 -- 22 --> 4
4	4 -- 22 --> 3, 4 -- 24 --> 6, 5 -- 25 --> 5
5	5 -- 10 --> 0, 5 -- 25 --> 4
6	6 -- 14 --> 1, 6 -- 18 --> 3, 6 -- 24 --> 4



Single Source All Destinations

Determine the shortest paths from v0 to all the remaining vertices.

*Figure 6.29: Graph and shortest paths from v_0 (p.293)



(a)

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

(b)

All Pairs Shortest Paths

Find the shortest paths between all pairs of vertices.

Solution 1

–Apply shortest path n times with each vertex as source.

$O(n^3)$

Solution 2

–Represent the graph G by its cost adjacency matrix with cost[i][j]

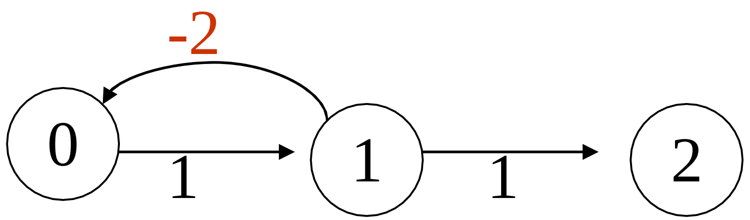
–If the edge <i,j> is not in G, the cost[i][j] is set to some sufficiently large number

–A[i][j] is the cost of the shortest path form i to j, using only those intermediate vertices with an index <= k

All Pairs Shortest Paths *(Continued)*

- The cost of 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
- $A^{-1}[i][j]=cost[i][j]$
- Calculate the $A^0, A^1, A^2, \ldots, A^{n-1}$ from A^{-1} iteratively
- $A^k[i][j]=\min\{A^{k-1}[i][j], A^{k-1}[i][k]+A^{k-1}[k][j]\}, k>=0$

Graph with negative cycle



(a) Directed graph

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

(b) A^{-1}

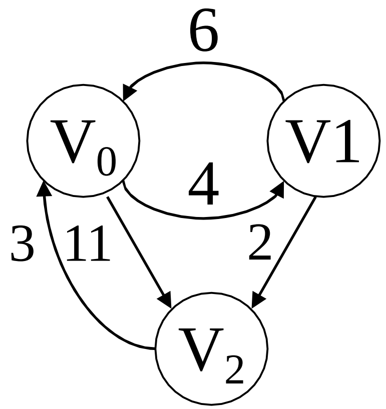
The length of the shortest path from vertex 0 to vertex 2 is $-\infty$.

0, 1, 0, 1,0, 1, …, 0, 1, 2

Algorithm for All Pairs Shortest Paths

```
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];
}
```

* Figure 6.33: Directed graph and its cost matrix (p.299)

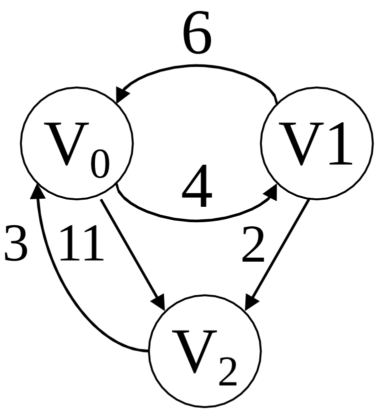


(a)Digraph G

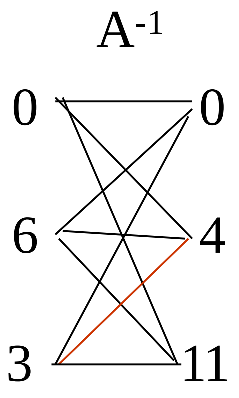
	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(b)Cost adjacency matrix for G

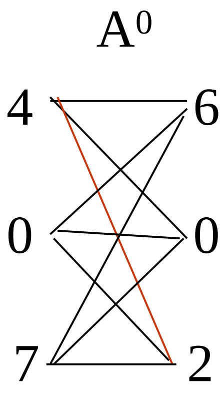
A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0



A^{0}	0	1	2
0	0	4	1
1	6	0	
2	3		



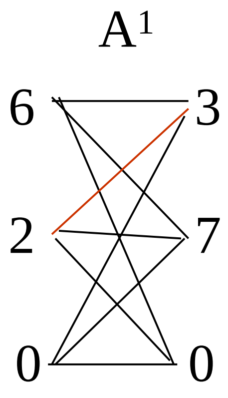
v0



v1

A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0



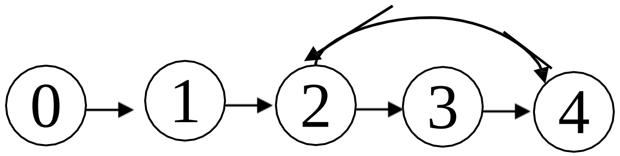
v2

Transitive Closure

Goal: given a graph with unweighted edges, determine if there is a path from i to j for all i and j.

(1) Require positive path (> 0) lengths. transitive closure matrix

(2) Require nonnegative path (≥0) lengths. reflexive transitive closure matrix



(a) Digraph G

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

(b) Adjacency matrix A for G

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

cycle

(c) transitive closure matrix A^+

There is a path of length > 0

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

reflexive

(d) reflexive transitive closure matrix A^*

There is a path of length ≥0