자료구조

Chap 09. Graph

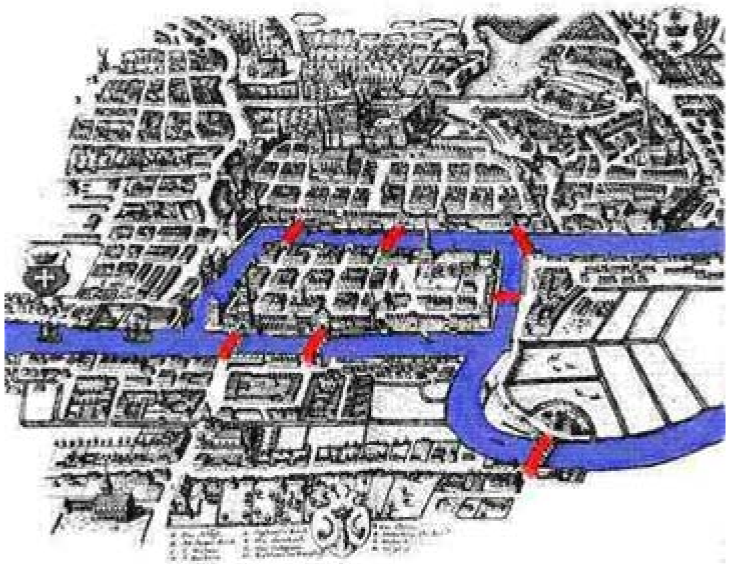
2018년 1학기

컴퓨터과학과
민 경 하

Contents

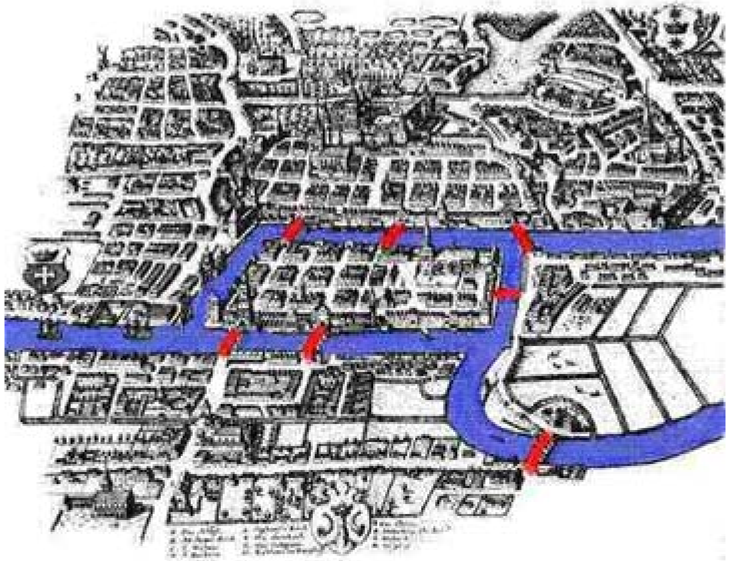
9.1 Introduction

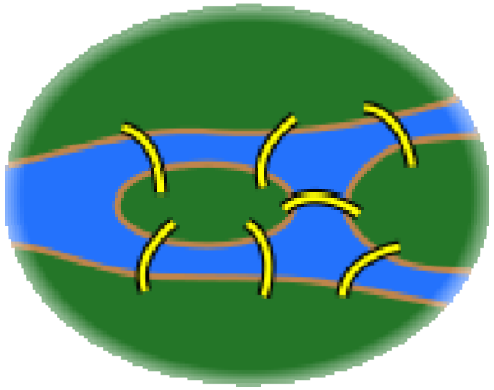
- Konigsberg bridge problem
  - To determine whether, starting at one land area, it is possible to walk across all the bridges exactly once in returning to starting land area

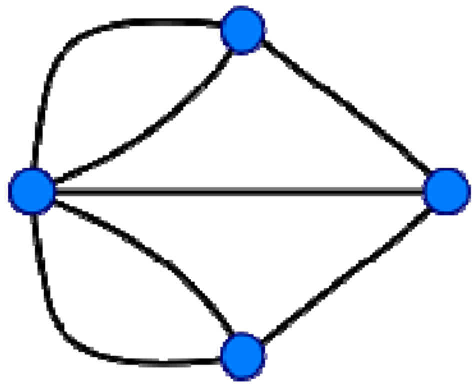


9.1 Introduction

- Konigsberg bridge problem
  - Euler introduced “Graph”
  - Abstraction
    - Land & island → node (vertex)
    - Bridge → edge (link)







9.2 Basic concepts

(1) Graph

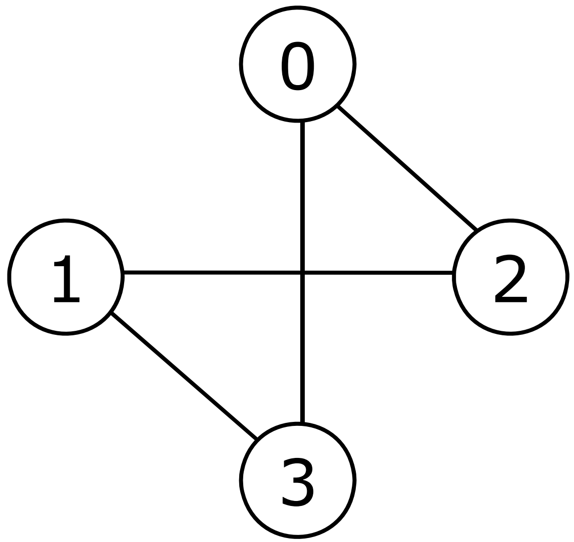
- A mathematical (abstracted) model that represents the one-to-one (binary) relationship between objects visually

- G = (V, E)
  - V
    - Vertex represents objects
  - E
    - Edge represents relationship
    - Edge is a pair of vertices

9.2 Basic concepts

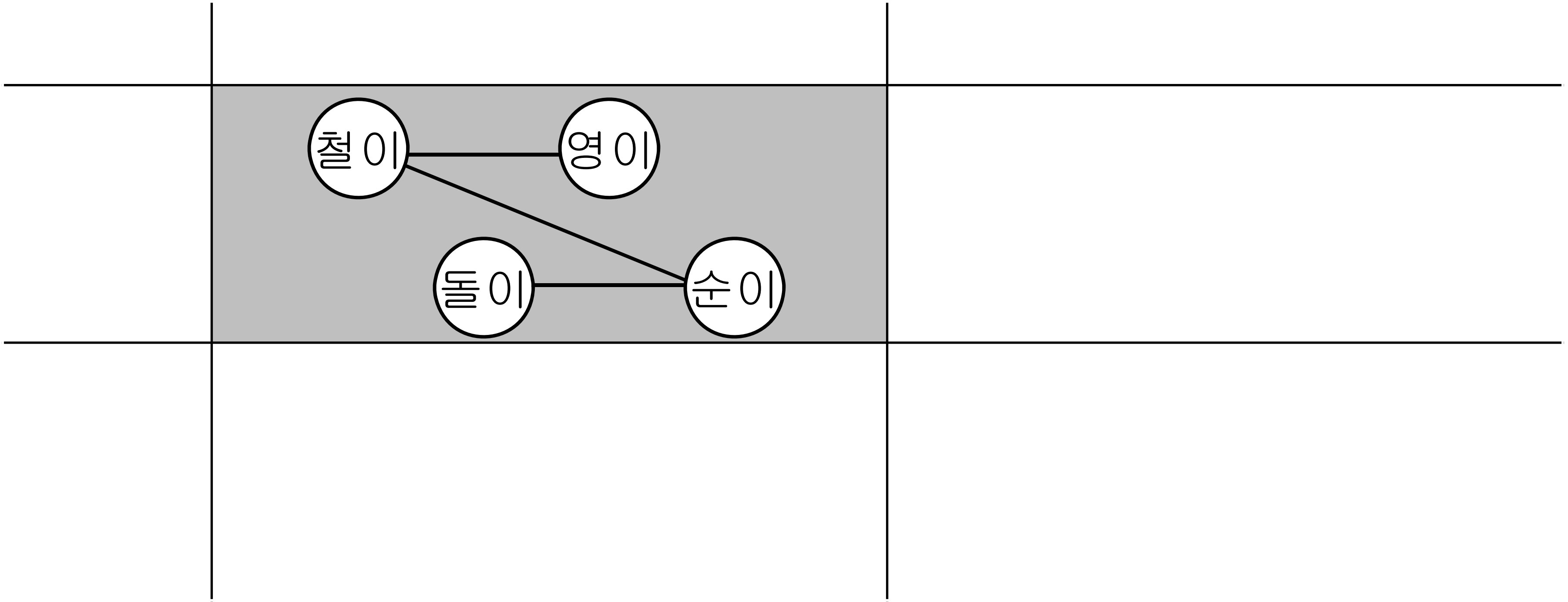
(1) Graph

- Mathematical representation
  - V = {0, 1, 2, 3}
  - E = {(0, 2), (0, 3), (1, 2), (1, 3)}
- Geometrical representation



9.2 Basic concepts

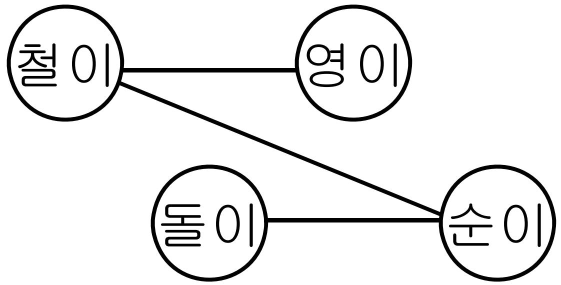
(2) Types of graph

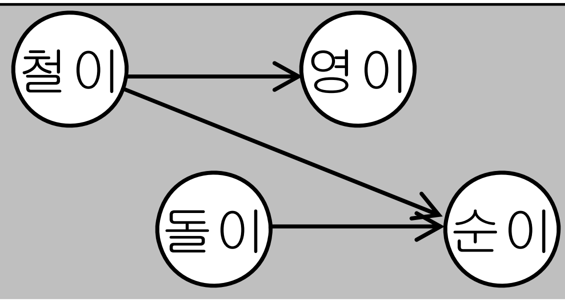


개체: 철이, 영이, 돌이, 순이
관계: 철이, 영이는 페친임.
철이, 순이는 페친임.
돌이, 순이는 페친임.

9.2 Basic concepts

(2) Types of graph





개체: 철이, 영이, 돌이, 순이

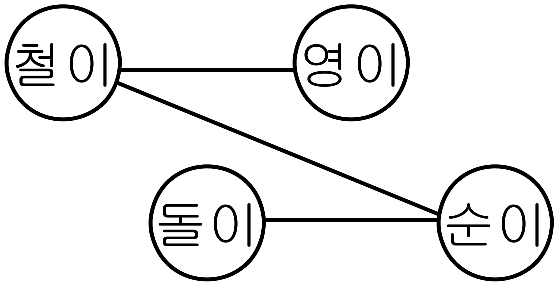
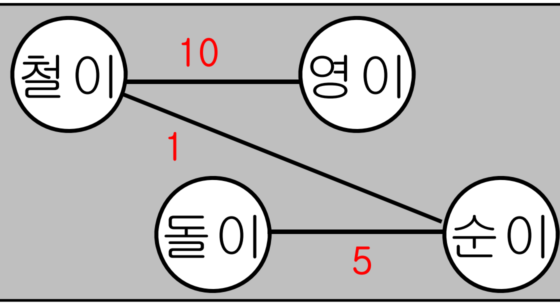
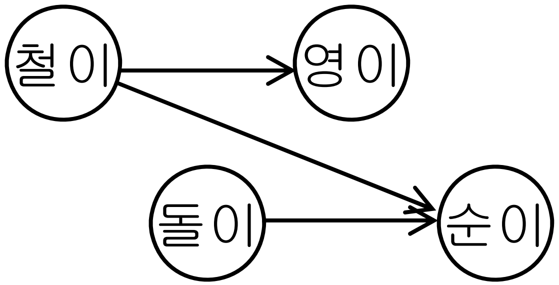
관계: 철이는 영이를 follow함.

철이는 순이를 follow함.

돌이는 순이를 follow함.

9.2 Basic concepts

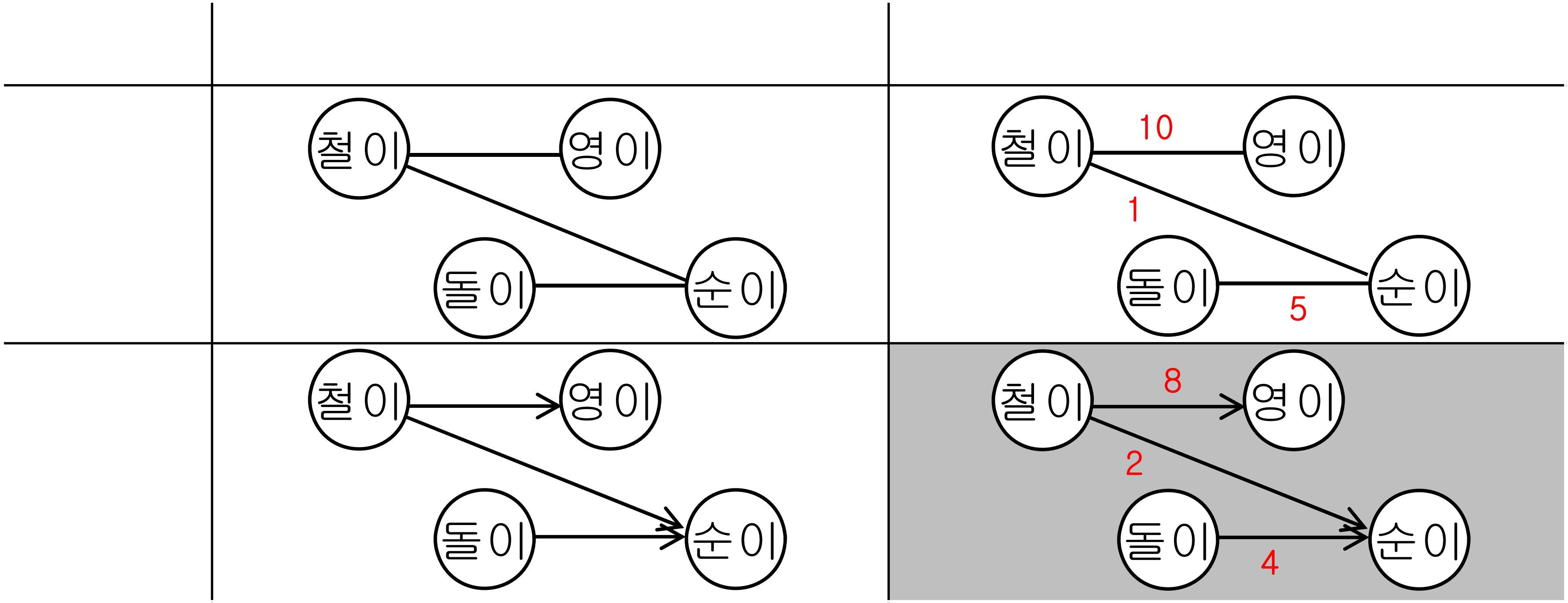
(2) Types of graph

개체: 철이, 영이, 돌이, 순이
관계: 철이, 영이는 심각한 페친임.
철이, 순이는 썰렁한 페친임.
돌이, 순이는 평범한 페친임.

9.2 Basic concepts

(2) Types of graph



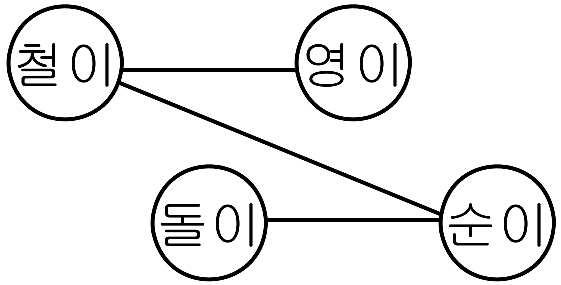
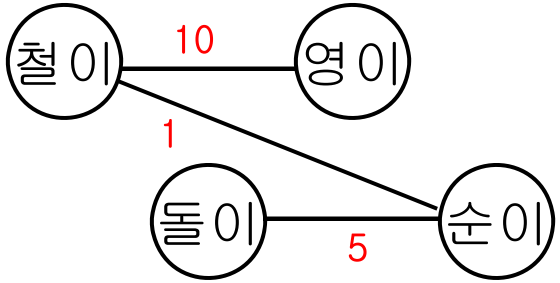
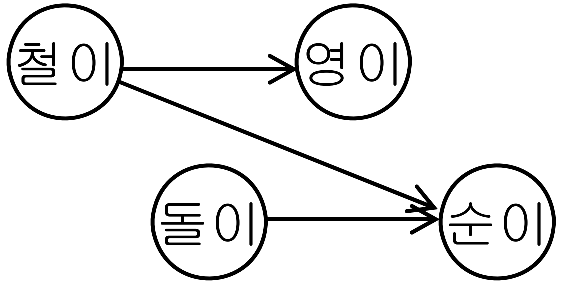
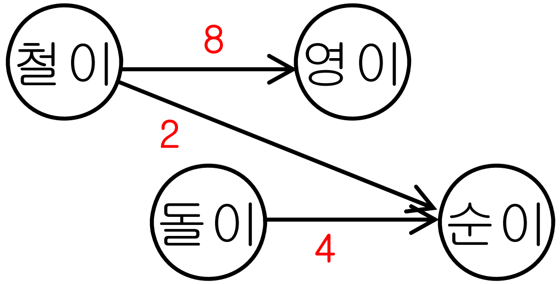
개체: 철이, 영이, 돌이, 순이

관계: 철이는 영이를 많이 follow함.

철이는 순이를 조금 follow함.

돌이는 순이를 보통 follow함

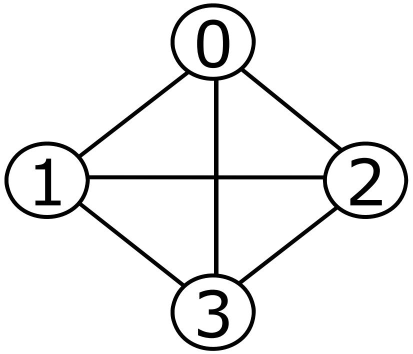
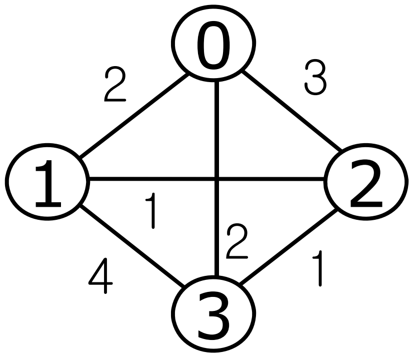
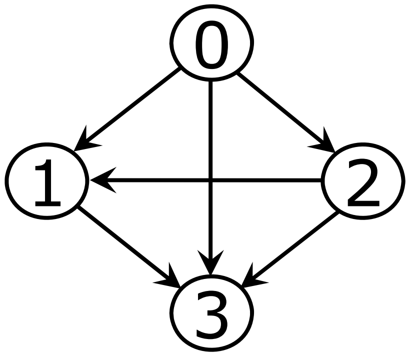
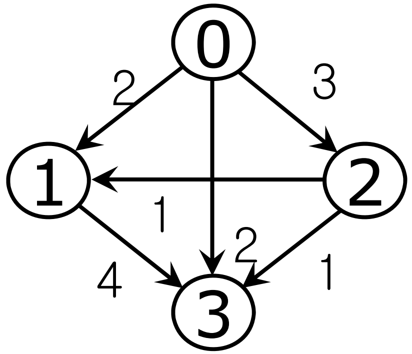
(2) Types of graph

		Weighted graph
Graph		
Directed graph		

(Undirected) Graph: (u, v) = (v, u)

Directed graph: (u, v) ≠ (v, u)

(2) Types of graph

		Weighted or not	
		Non-weighted	**Weighted**
Directed or not	**Undirected**		
	Directed		

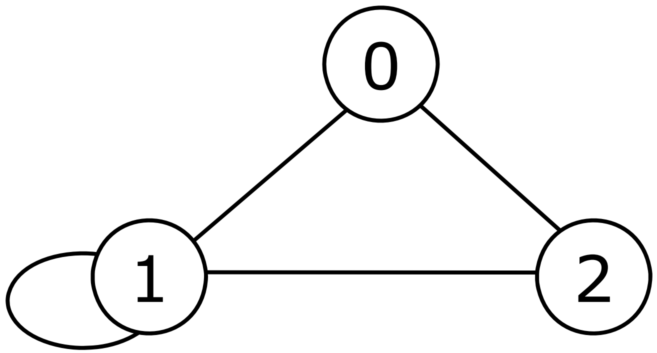
9.2 Basic concepts

(2) Types of graph

– Extraordinary case: Graph with self edge

- self edge: (u, u)

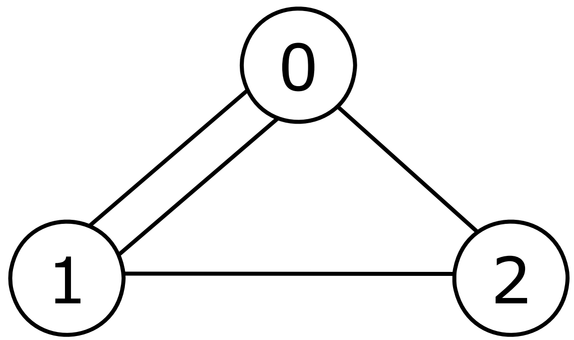
- G = (V, E)
  - V = {0, 1, 2}
  - E = { (0, 1), (0, 2), (1, 2), (1, 1) }



9.2 Basic concepts

(2) Types of graph

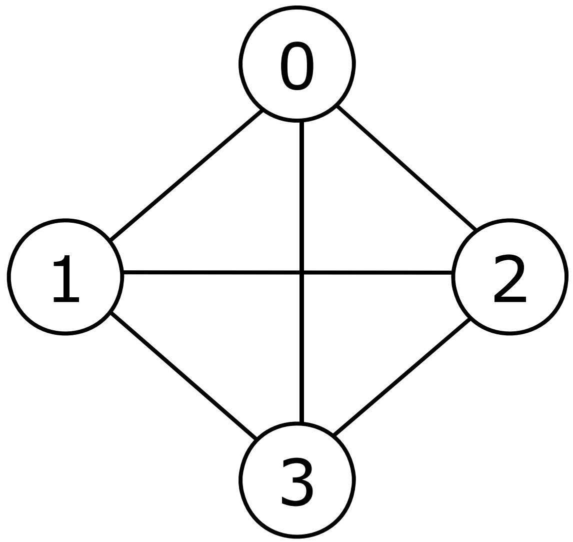
- Extraordinary case: Multigraph
  - A graph with a set of multiple edges



9.2 Basic concepts

(3) Complexity of graph

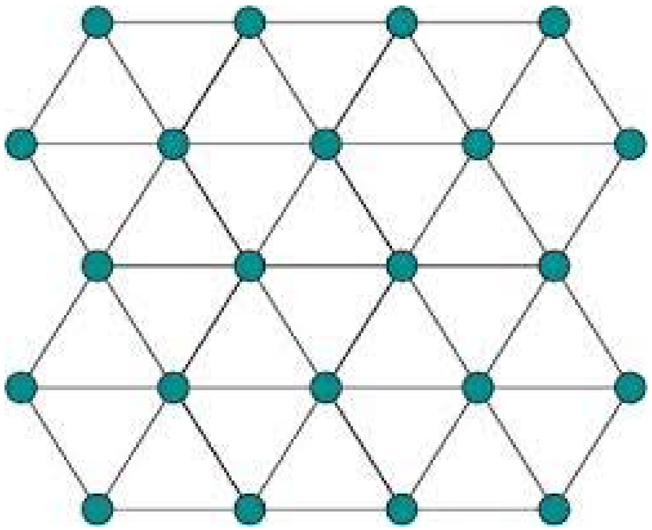
- Complete graph
  - Maximum number of edges of a graph with n vertices → n(n-1)/2
  - Ex)
    - V = {0, 1, 2, 3}
    - n(E) = 4*3/2 = 6
  - Complete graph
    - A graph whose no. of edges is n(n-1)/2
  - Dense graph

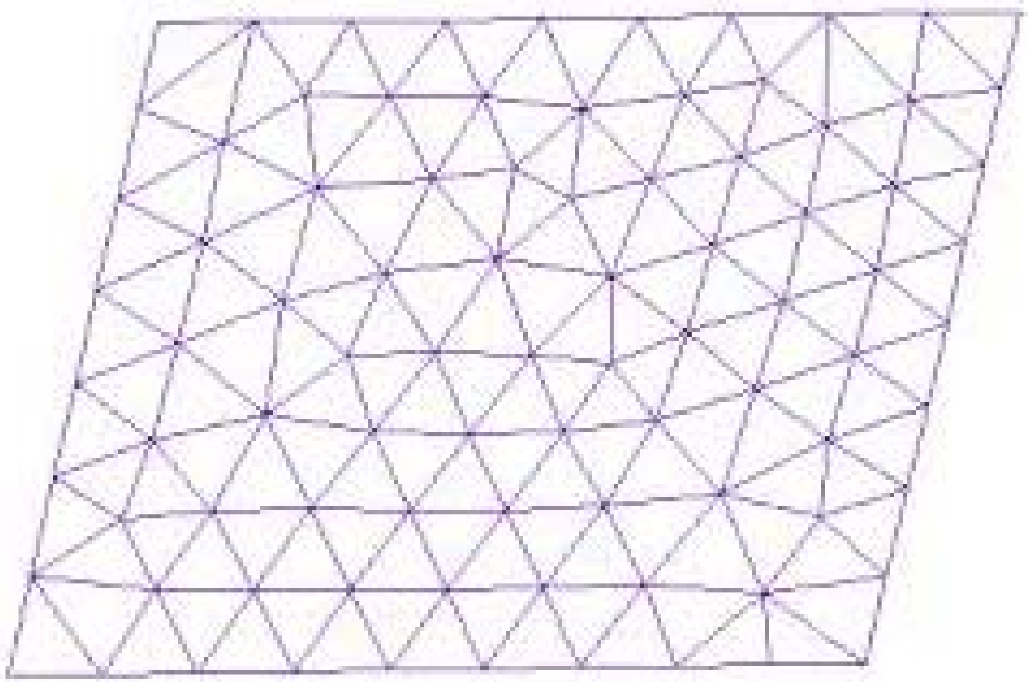


9.2 Basic concepts

(3) Complexity of graph

- Sparse graph
  - A graph whose vertices are connected to a constant number of edges.

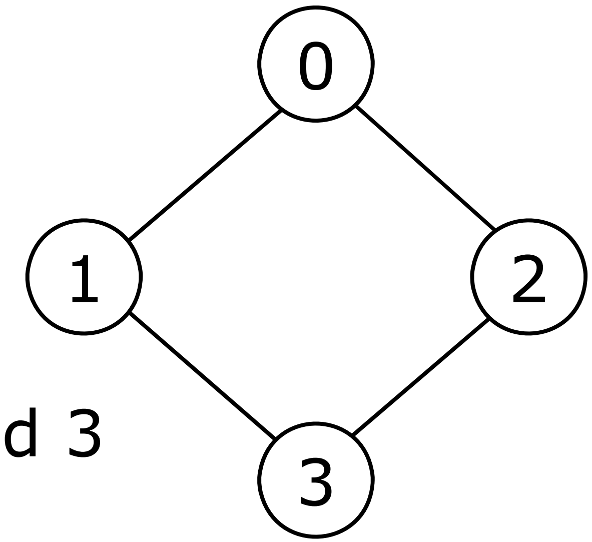




9.2 Basic concepts

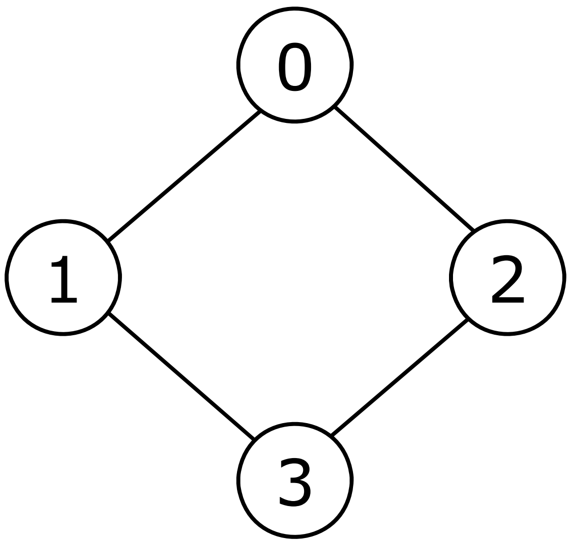
(4) Adjacency & Incidency

- If (u, v) $\in$ E, then
  - u and v are adjacent
  - (u, v) is incident to u and v
- Ex)
  - 0 and 1 are adjacent
  - 0 and 3 are not adjacent
  - (0, 1) is incident to 0 and 1
  - (0, 1) is not incident to 2 and 3

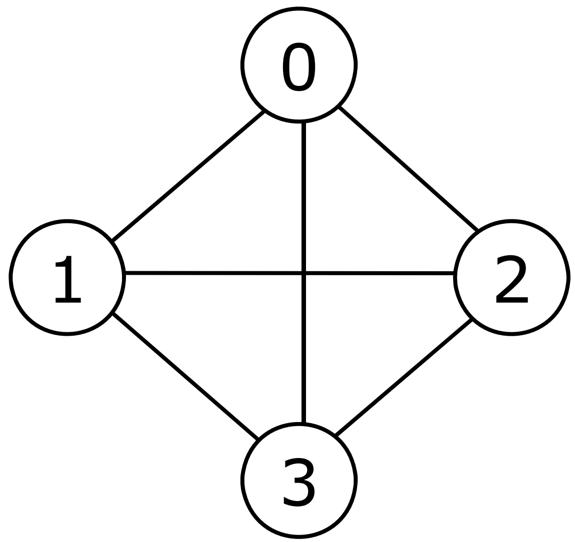


(5) Subgraph

- Graph G′ = (V′, E′) is a subgraph of G = (V, E), if V′ $\subseteq$ V and E′ $\subseteq$ E



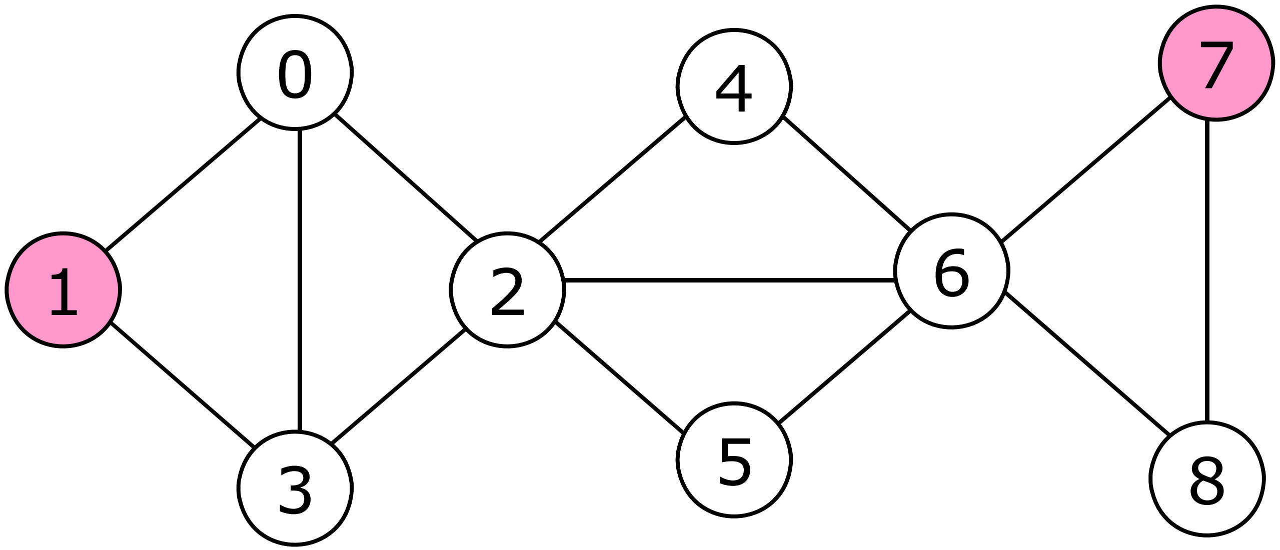
is a subgraph of



9.2 Basic concepts

(6) Path

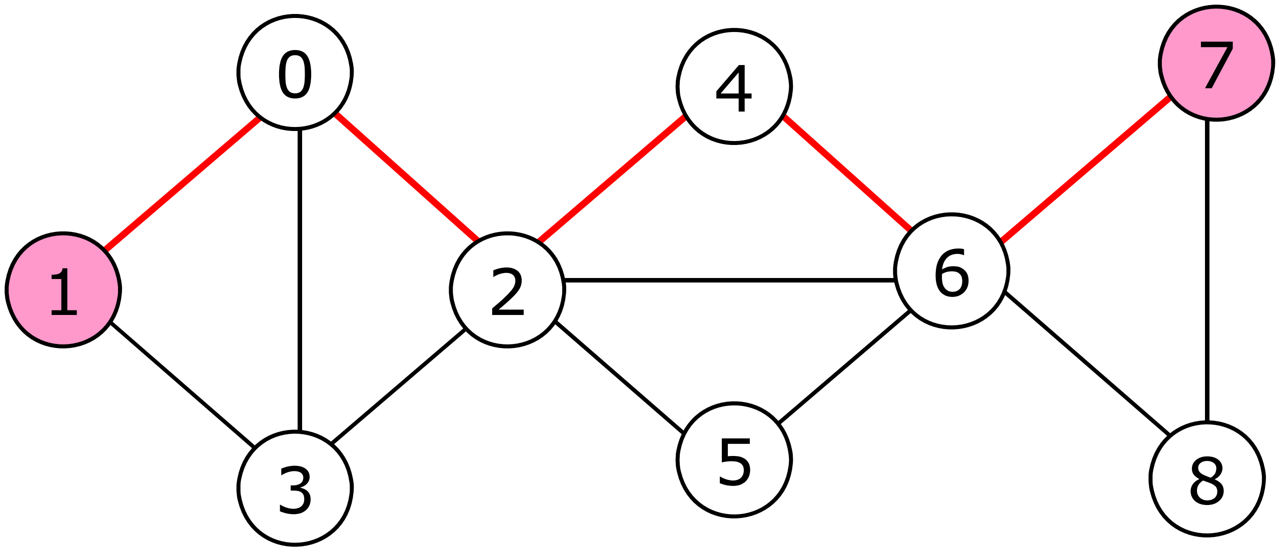
- A path from a vertex u to a vertex v is a sequence of vertices u, i_1, i_2, …, i_k, v such that (u, i_1), (i_1, i_2), …, (i_k, v) $\in$ E
- Path from 1 to 7: 1, 0, 2, 4, 6, 7



9.2 Basic concepts

(6) Path

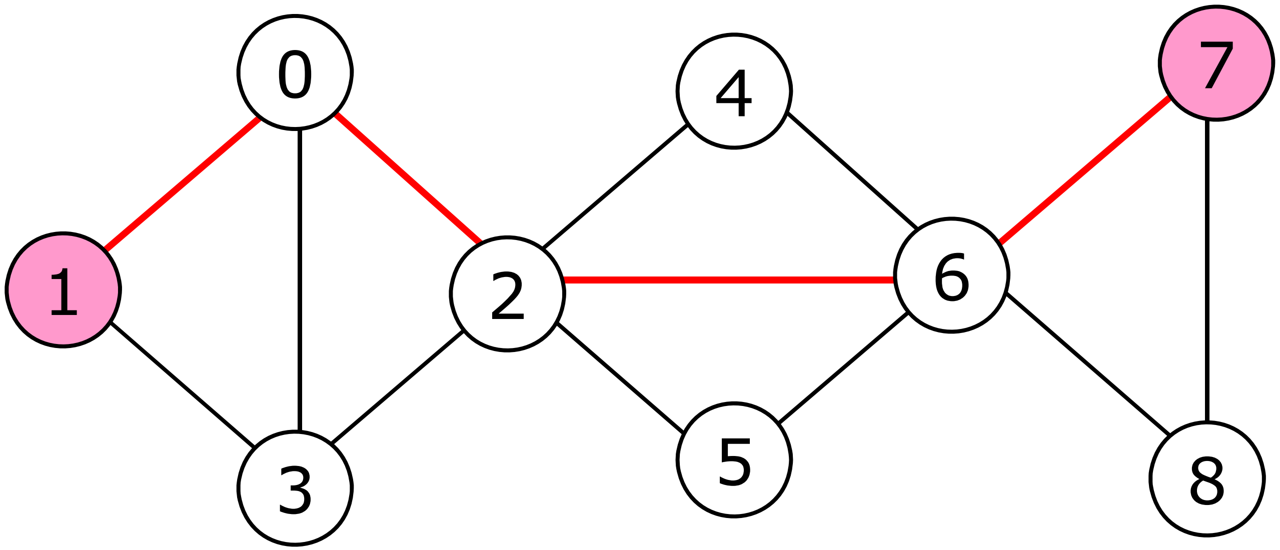
- A path from a vertex u to a vertex v is a sequence of vertices u, i_1, i_2, …, i_k, v such that (u, i_1), (i_1, i_2), …, (i_k, v) $\in$ E
- Path from 1 to 7: 1, 0, 2, 4, 6, 7



9.2 Basic concepts

(6) Path

- A path from a vertex u to a vertex v is a sequence of vertices u, i_1, i_2, …, i_k, v such that (u, i_1), (i_1, i_2), …, (i_k, v) $\in$ E
- Another Path: 1, 0, 2, 6, 7



9.2 Basic concepts

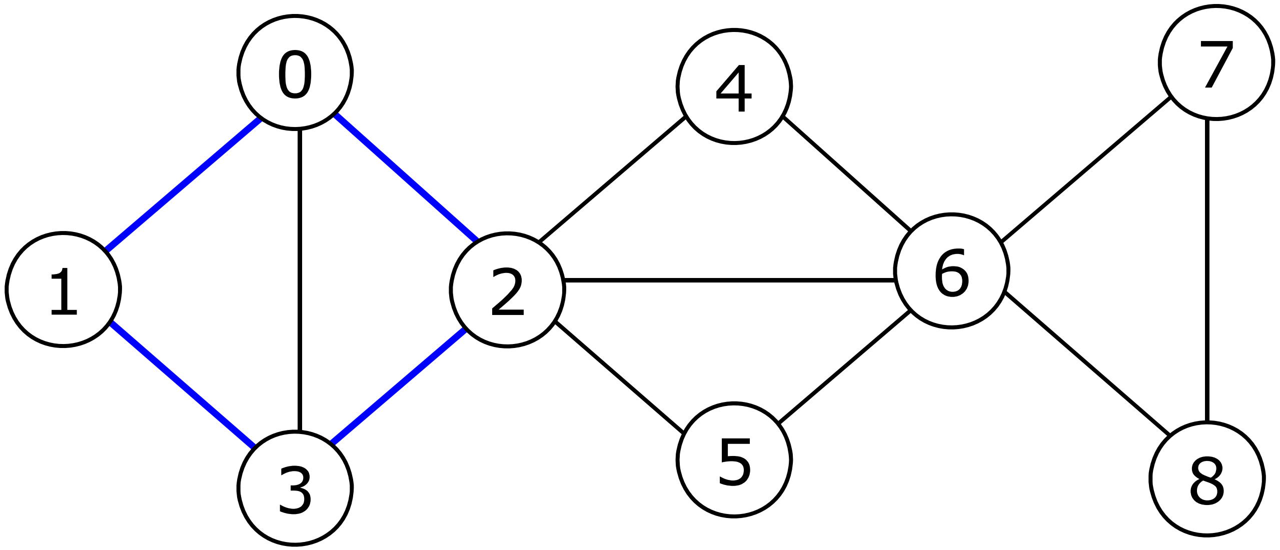
(6) Path

- A path from a vertex u to a vertex v is a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$ such that $(u, i_1), (i_1, i_2), \ldots, (i_k, v) \in E$
- Multiple paths can exist
- The length of a path
  - The number of edges on it
  - The number of vertices on it
- Simple path
  - A path in which all vertices except possibly the first and last are distinct

9.2 Basic concepts

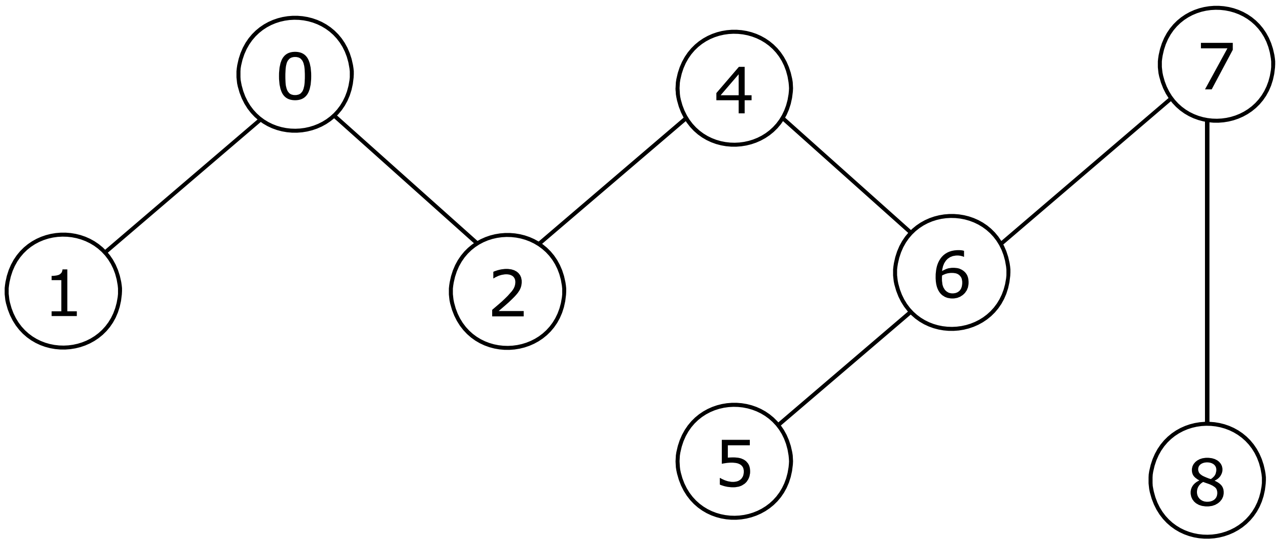
(7) Cycle

- A simple path in which the first and last vertices are the same
- Ex)
  - Path “0, 1, 3, 2, 0” is a cycle



(7) Cycle

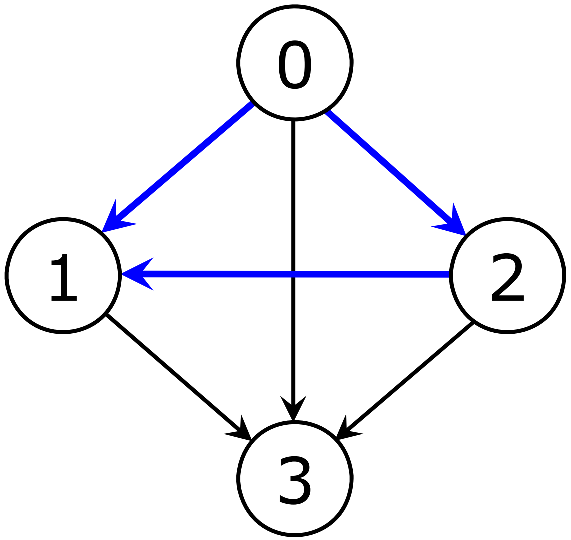
- Acyclic graph
  - A graph that does not contain a cycle



9.2 Basic concepts

(7) Cycle

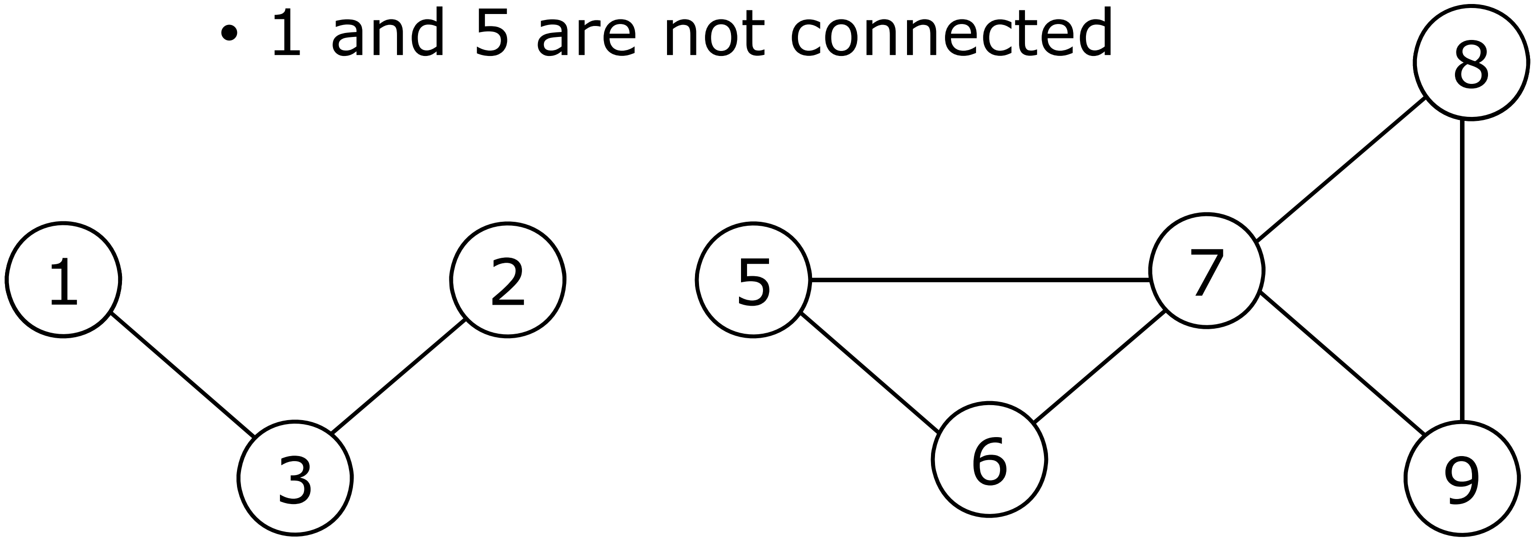
- A simple path in which the first and last vertices are the same
- Ex)
  - Path "0, 2, 1, 0" is not a cycle



9.2 Basic concepts

(8) Connected

- Two vertices u and v are connected if there is a path from u to v
- Ex)
  - 5 and 8 are connected
  - 1 and 5 are not connected



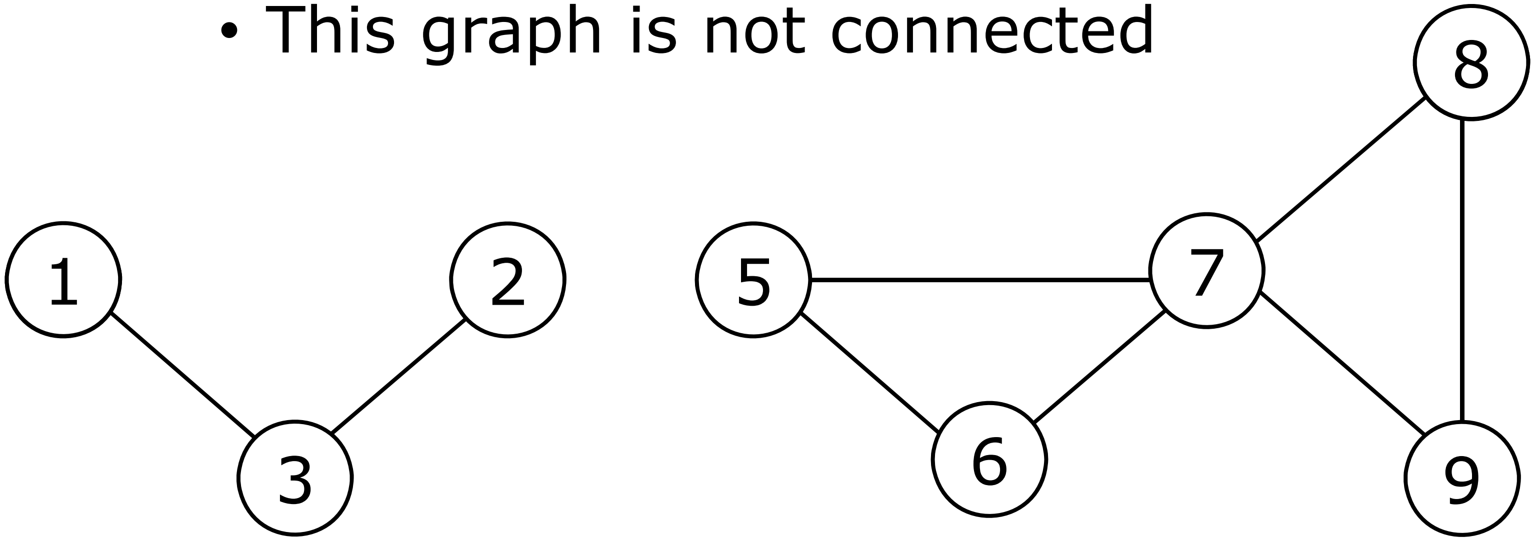
9.2 Basic concepts

(8) Connected

– A graph is **connected** if for every pair of distinct vertices u and v, there is a path from u to v

– Ex)

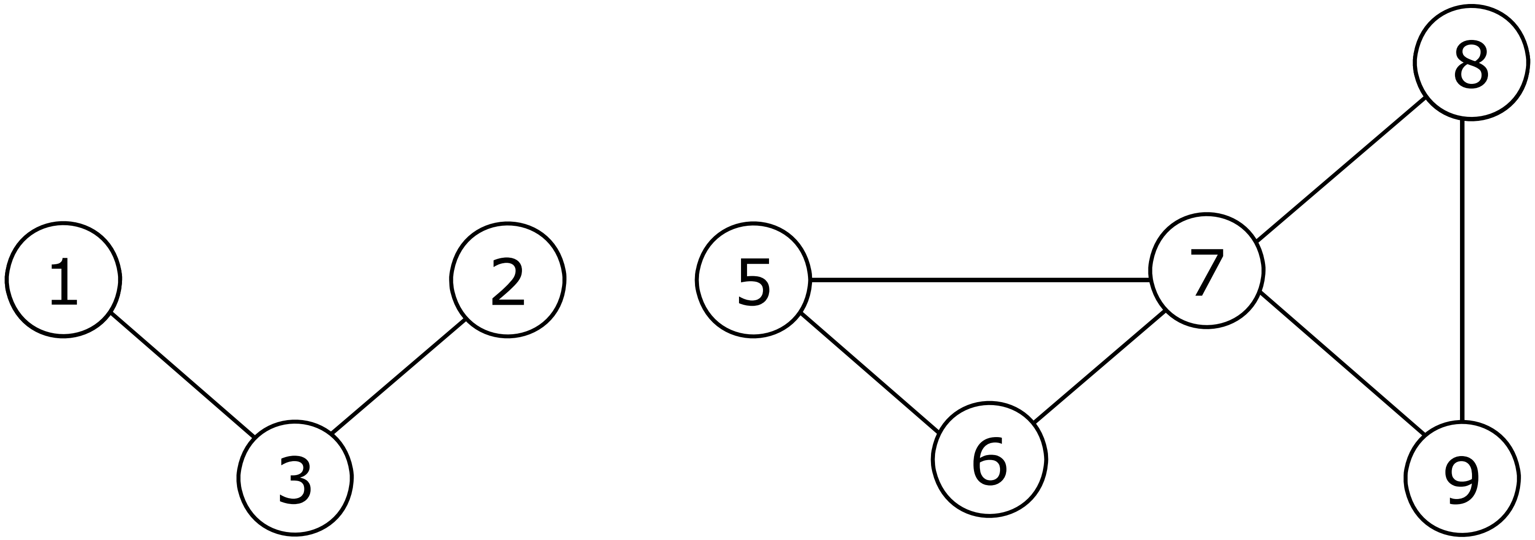
- This graph is not connected



9.2 Basic concepts

(8) Connected

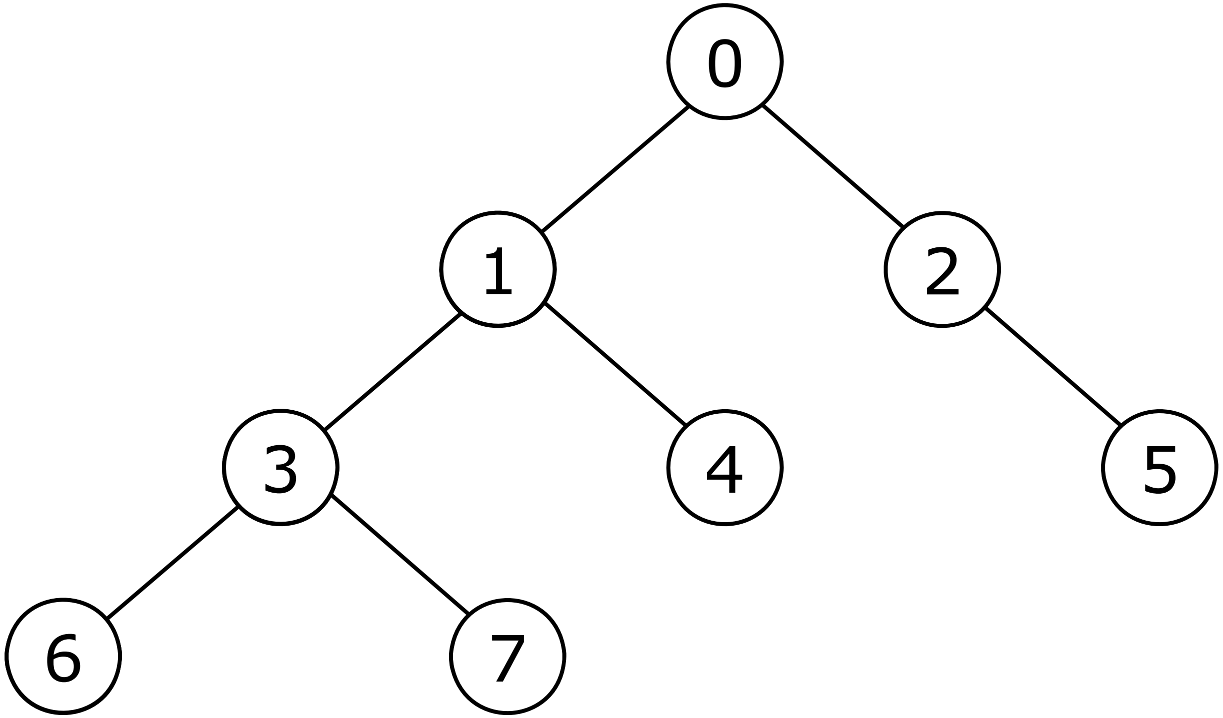
- Connected component
  - A maximal connected subgraph
- Ex)
  - Two connected subgraphs



9.2 Basic concepts

(8) Connected

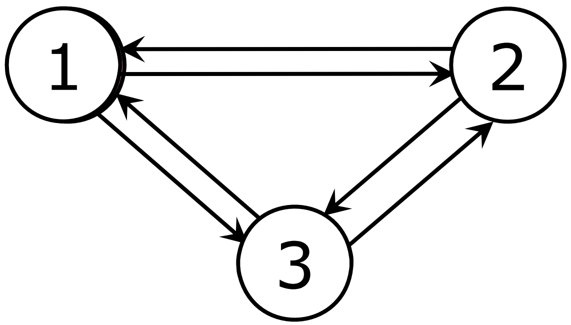
– Tree is a connected acyclic graph



9.2 Basic concepts

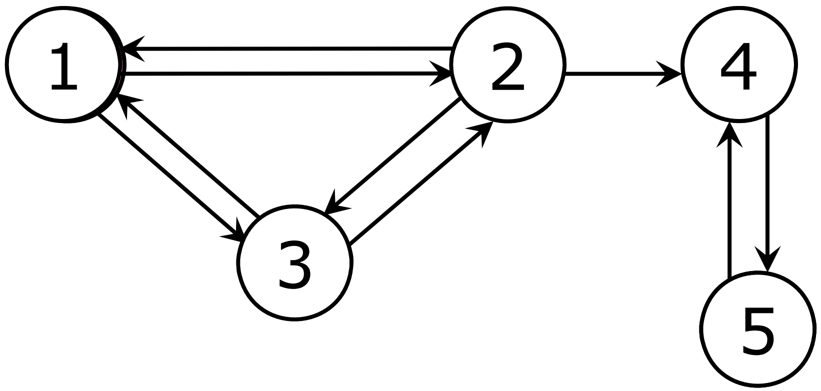
(8) Connected

– A directed graph is **strongly connected**, if for every pair of distinct vertices u and v, there is a directed path from u to v and also from v to u



(8) Connected

- A strongly connected component is a maximal subgraph that is strongly connected
- Ex)
  - {1, 2, 3} is a strongly connected component
  - {4, 5} is a strongly connected component



9.2 Basic concepts

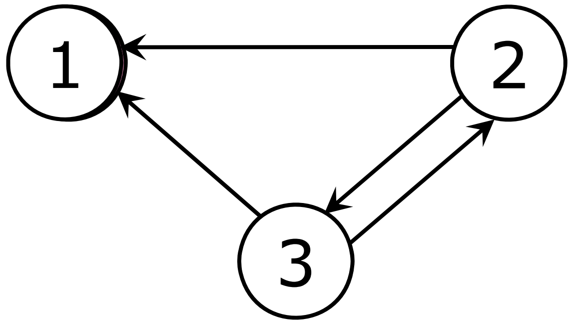
(9) Degree of a vertex

- The number of edges incident to a vertex
- In-degree of a vertex v
  - The number of edges for which v is the head
- Out-degree of a vertex v
  - The number of edges for which v is the tail

9.2 Basic concepts

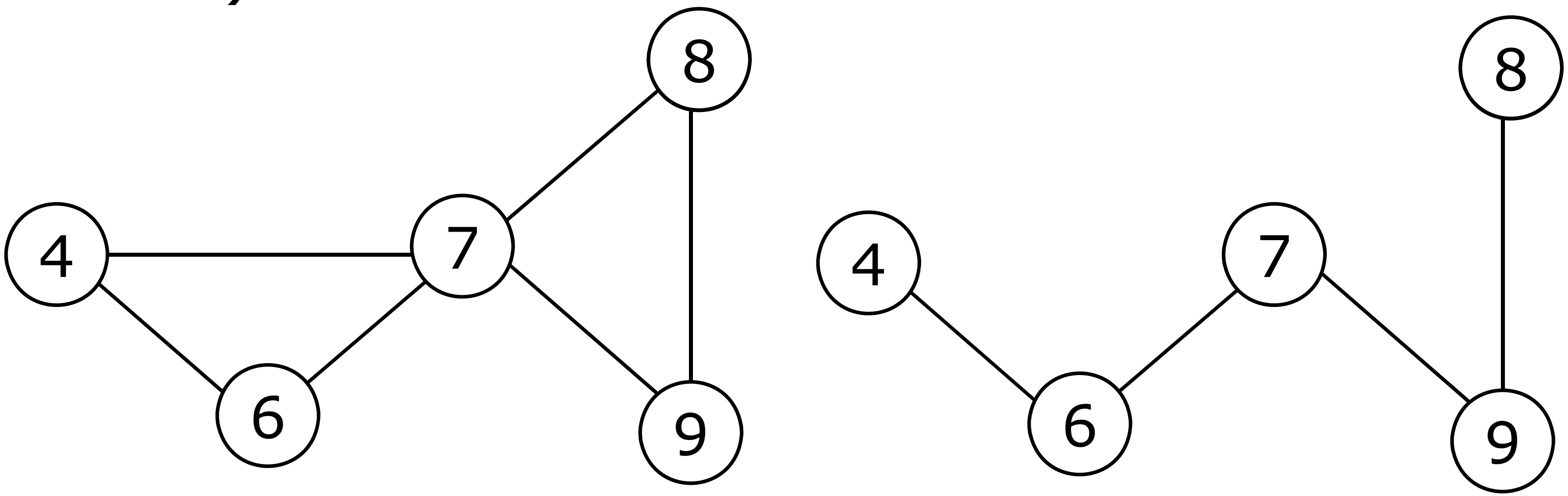
(10) Degree of a vertex

- Ex)
  - Degree of vertex 2 = 3
  - In-degree of vertex 3: 1
  - Out-degree of vertex 3: 2



(11) Spanning tree

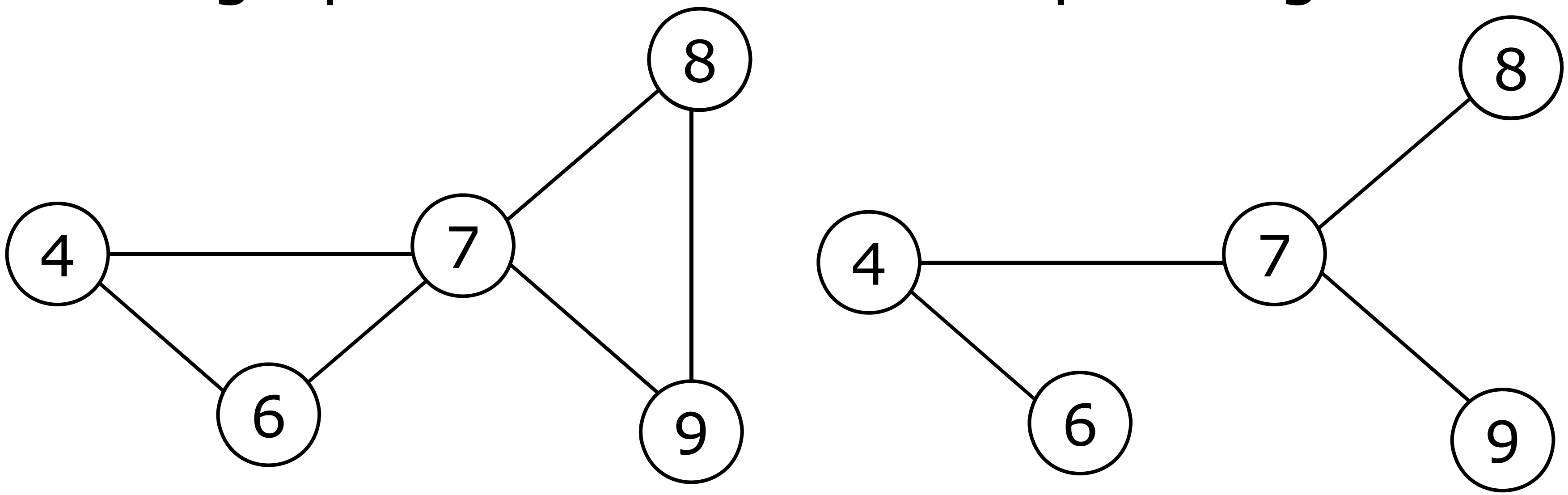
- T = (V', E') is a spanning tree of a graph G = (V, E), if V' = V and E' $\subseteq$ E and E' does not contain a cycle
- Ex)



9.2 Basic concepts

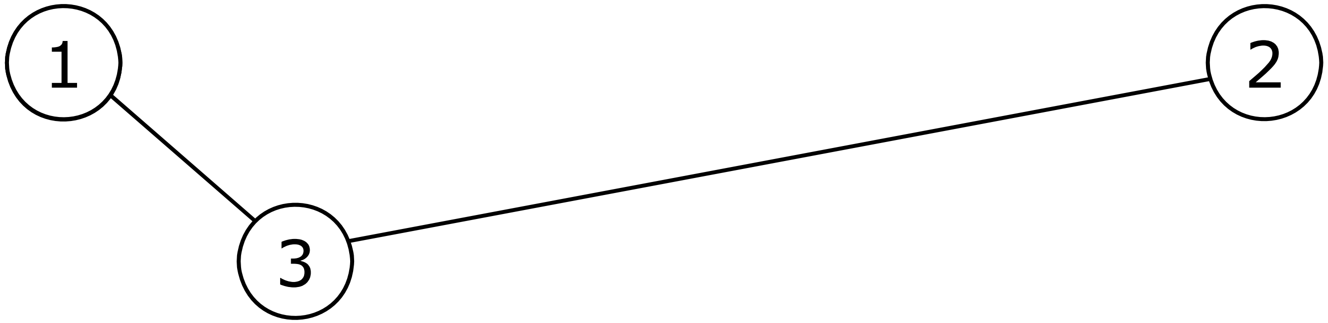
(11) Spanning tree

- T = (V′, E′) is a spanning tree of a graph G = (V, E), if V′ = V and E′ $\subseteq$ E and E′ does not contain a cycle
- A graph can have several spanning trees



(12) Weighted graph

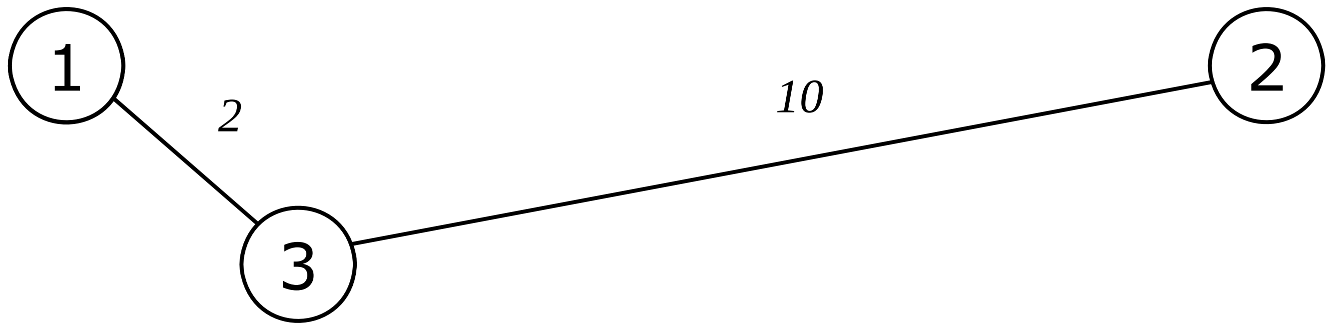
– A graph whose edges have some weights



9.2 Basic concepts

(12) Weighted graph

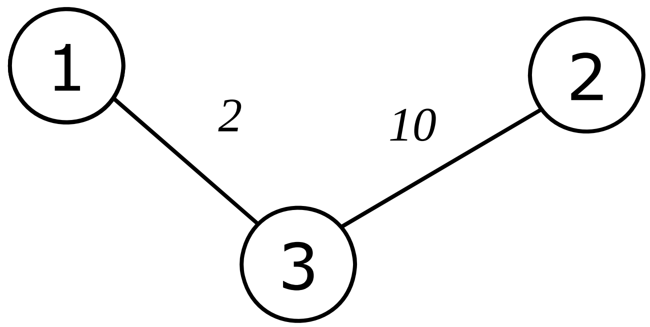
– A graph whose edges have some weights



9.2 Basic concepts

(12) Weighted graph

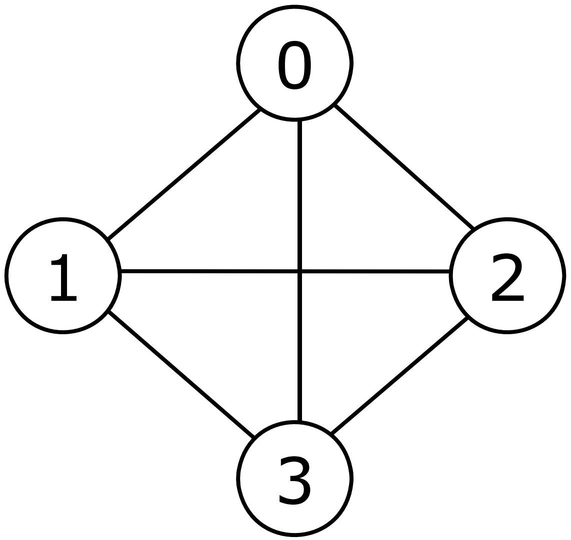
– A graph whose edges have some weights



9.3 Graph Representation

(1) Edge list

- A list of edges
- Available on many coding problems

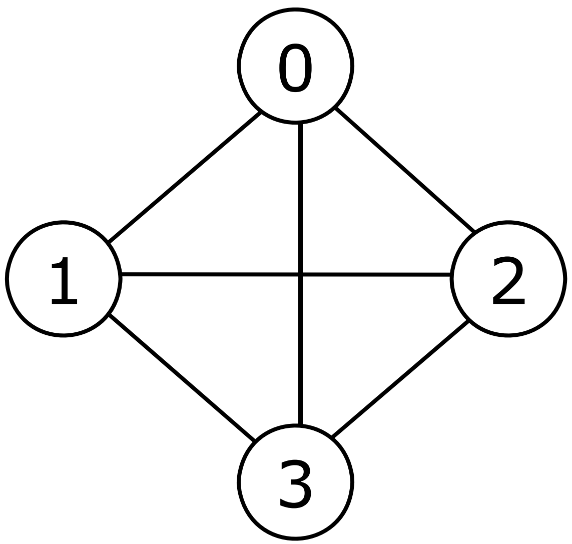


4, 6
0, 1
0, 2
0, 3
1, 2
1, 3
2, 3

9.3 Graph Representation

(2) Adjacency matrix of G = (V, E)

- A two-dimensional n X n array: a[n][n]
- a[i][j] = 1, if $(v_i, v_j) \in E$
- a[i][j] = 0, if (v_i, v_j) $! \in$ E

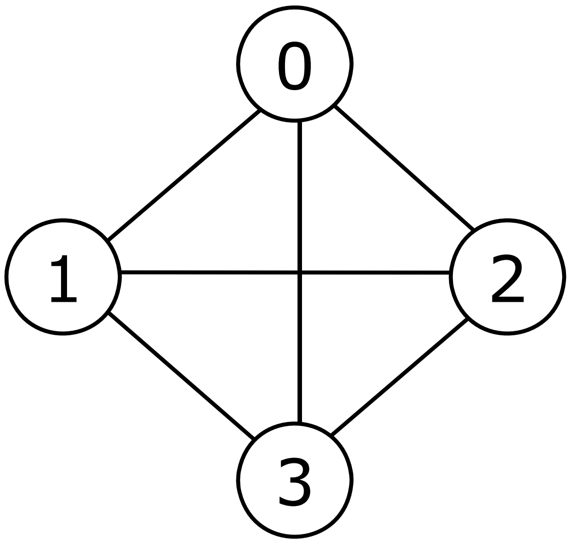


$$\begin{array}{cc} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} \ & \ & \ & \ \\ \ & \ & \ & \ \\ \ & \ & \ & \ \\ \ & \ & \ & \ \end{bmatrix} \end{array}$$

9.3 Graph Representation

(2) Adjacency matrix of G = (V, E)

- A two-dimensional n X n array: a[n][n]
- a[i][j] = 1, if $(v_i, v_j) \in E$
- a[i][j] = 0, if $(v_i, v_j) \, !\!\in E$

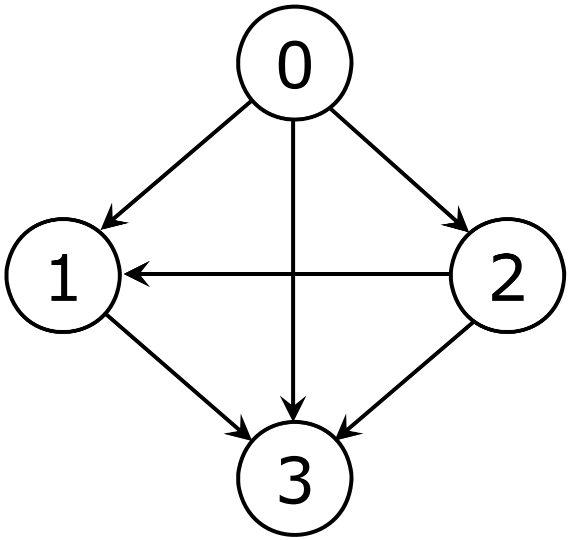


$$\begin{array}{cc} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

9.3 Graph Representation

(2) Adjacency matrix of G = (V, E)

- A two-dimensional n X n array: a[n][n]
- a[i][j] = 1, if $<v_i, v_j> \in E$
- a[i][j] = 0, if $<v_i, v_j>\ !\in E$

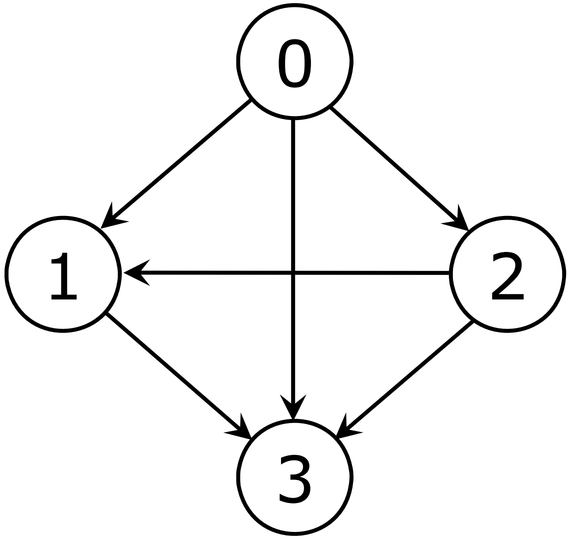


$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} \end{array}$$

9.3 Graph Representation

(2) Adjacency matrix of G = (V, E)

- A two-dimensional n X n array: a[n][n]
- $a[i][j] = 1$, if $<v_i, v_j> \in E$
- $a[i][j] = 0$, if $<v_i, v_j>\, !\!\in E$

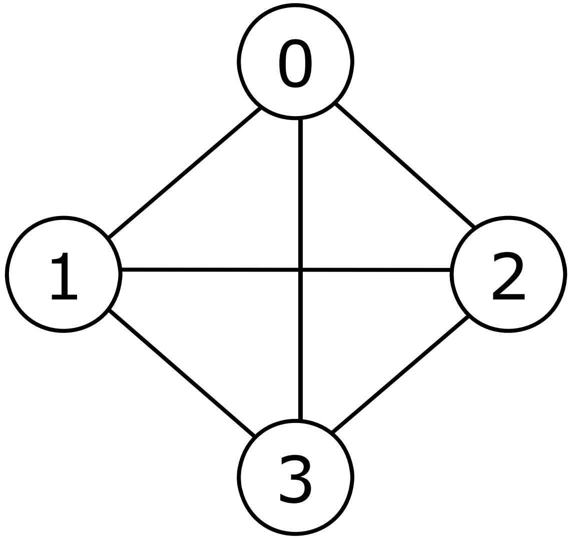


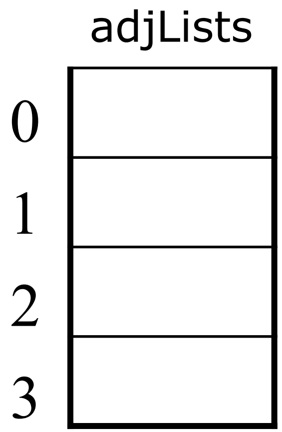
$$\begin{array}{c c} & \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

9.3 Graph Representation

(3) Adjacency list of G = (V, E)

- adjLists[n]
- adjLists[i] is a pointer to the first node in the adjacency list for vertex i

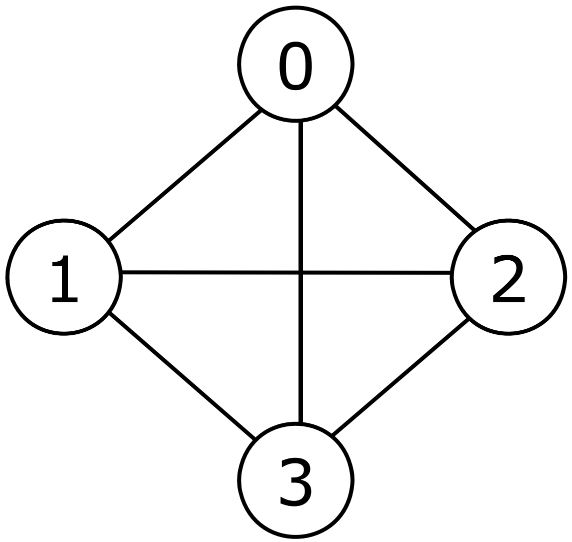


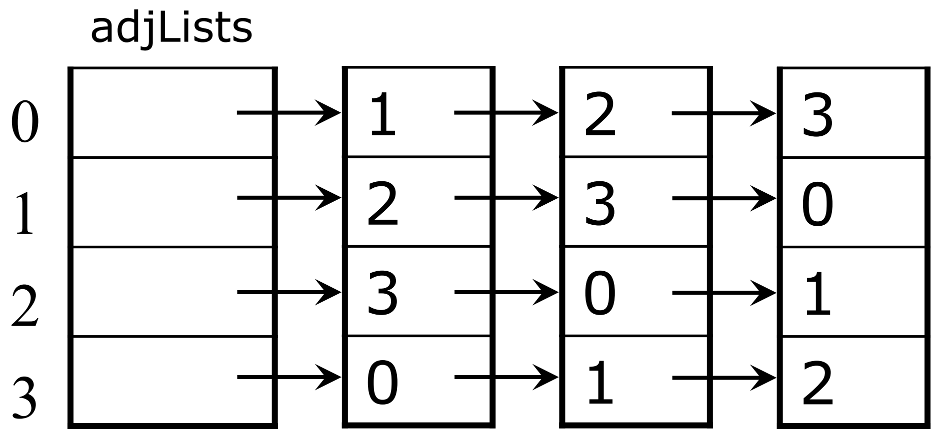


9.3 Graph Representation

(3) Adjacency list of G = (V, E)

- adjLists[n]
- adjLists[i] is a pointer to the first node in the adjacency list for vertex i

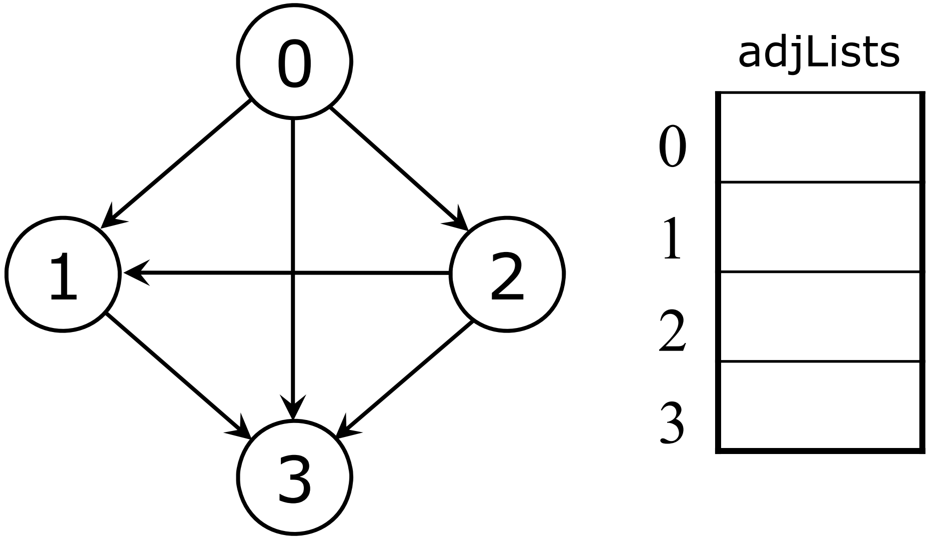




9.3 Graph Representation

(3) Adjacency list of G = (V, E)

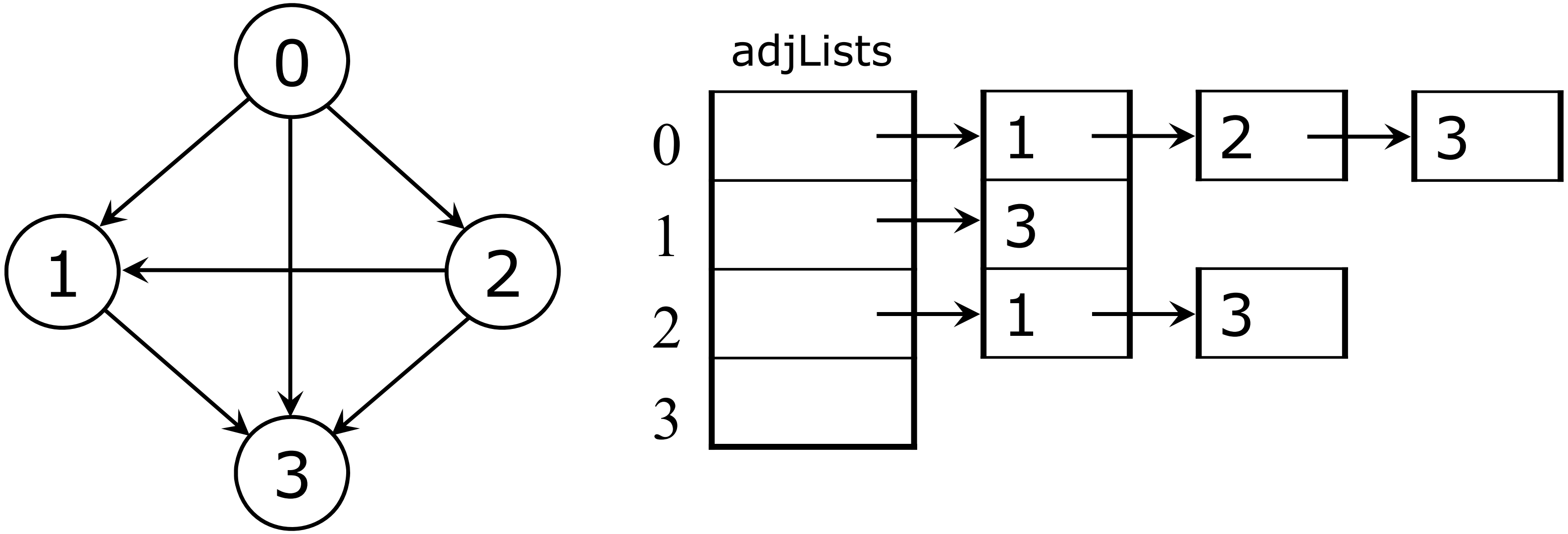
- adjLists[n]
- adjLists[i] is a pointer to the first node in the adjacency list for vertex i



9.3 Graph Representation

(3) Adjacency list of G = (V, E)

- adjLists[n]
- adjLists[i] is a pointer to the first node in the adjacency list for vertex i

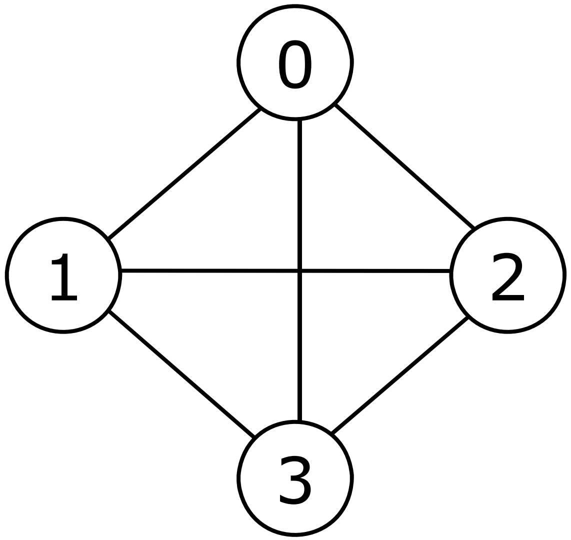


9.3 Graph Representation

- Comparison of performance
  - Ex) At all vertices, list all the incident vertices

```
for all vertices v in G
    for all vertices w adjacent to v
        report (v, w);
```

  - Expected output:
    - 0: 1, 2, 3
    - 1: 0, 2, 3
    - 2: 0, 1, 3
    - 3: 0, 1, 2



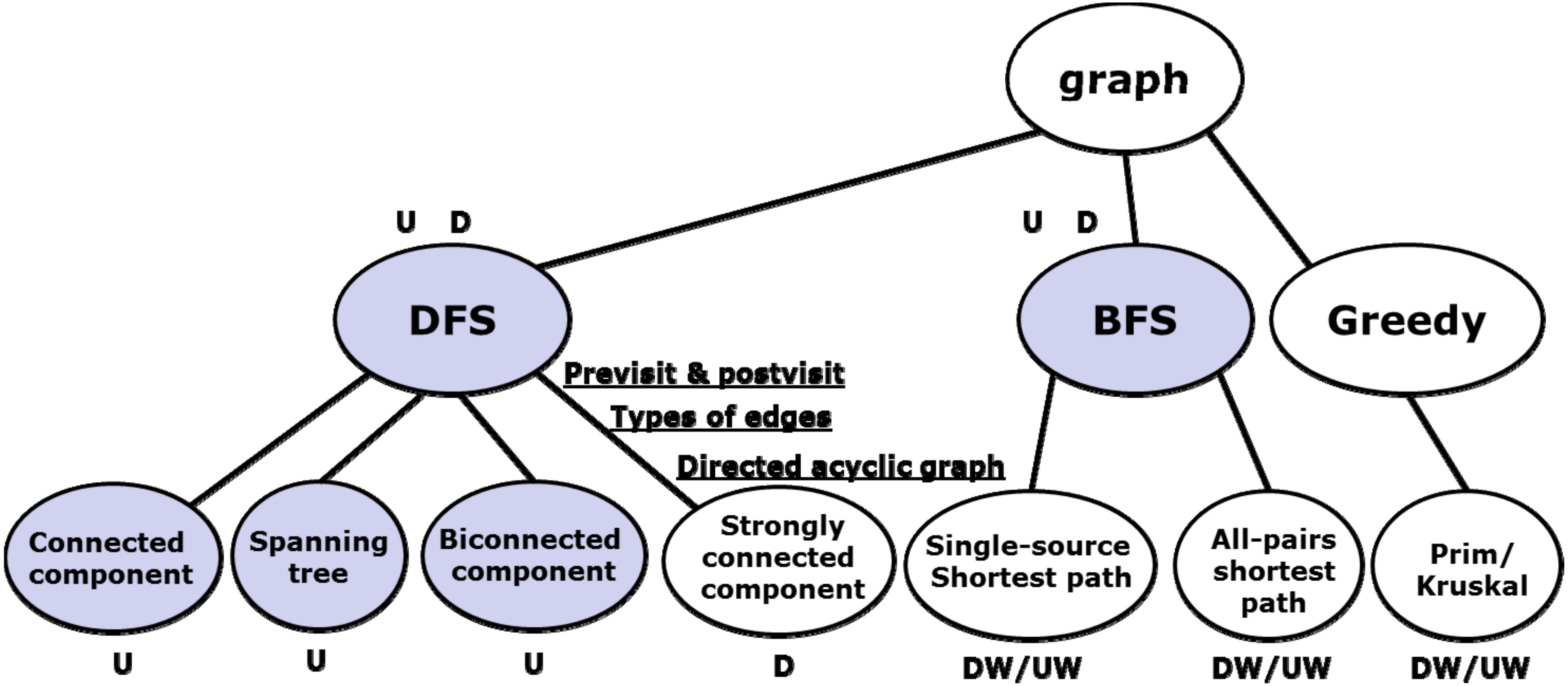
9.3 Graph Representation

- Comparison of performance
  - Ex) At all vertices, list all the incident vertices

```
for all vertices v in G
    for all vertices w adjacent to v
        report (v, w);
```

	Adjacency matrix	**Adjacency list**
Complete graph	$O(n^2)$	$O(n^2)$
Sparse graph	$O(n^2)$	**O(n)**

9.4 Search



graph
U D
DFS
U D
BFS
Greedy
Previsit & postvisit
Types of edges
Directed acyclic graph
Connected component
Spanning tree
Biconnected component
Strongly connected component
Single-source Shortest path
All-pairs shortest path
Prim/ Kruskal
U
U
U
D
DW/UW
DW/UW
DW/UW

9.4 Search

- Problem of search on a graph
  - Given a graph $G = (V, E)$ and a vertex $v \in V$, find all the vertices that are reachable from v.
  - Depth-first search
  - Breadth-first search

9.4.1 Depth-First Search

- Depth-first search
  - In visiting a node v,
    - Mark node v as VISITED
    - Select an unvisited node w, which is adjacent to v
    - Push w to a stack
    - Find the next node to visit among the adjacent vertices of v
    - If cannot find, v ← pop a stack

9.4.1 Depth-First Search

- Required data structure
  - visit[n]
    - visit[i] == 1, if node i is visited,
      == 0, if node i is unvisited
  - Stack → recursive call → function stack

9.4.1 Depth-First Search

```
void dfs ( int v )
{
    nodePointer w;
    visit[v] = TRUE;

    for ( w = graph[v]; w; w = w->link ) {
        if ( !visit[w->vertex] )
            dfs ( w->vertex );
    }
}


…
    for all v in G
        if ( visit[v] == FALSE )
            dfs ( v );
…
```

Adjacency matrix → O(n)

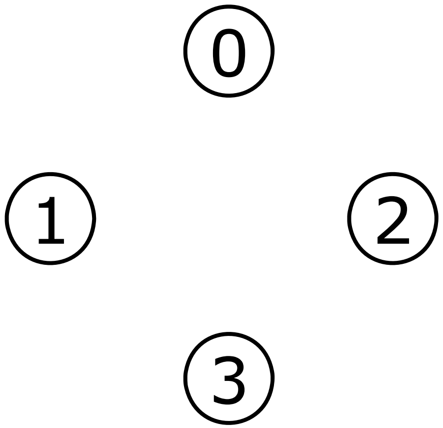
```
for ( int i = 0; i < n; i++ )
    if ( adjacency_matrix[v][i] != 0 )
       ……
```

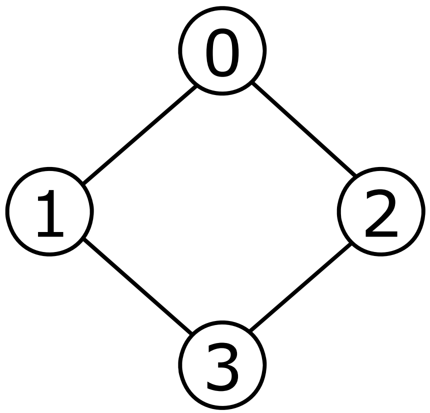
Adjacency list → O(1) ~ O(n)

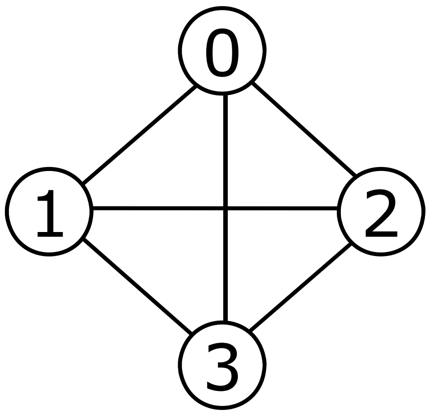
```
for ( t = v; t != NULL; t = t->next)
    ……
```

9.4.1 Depth-First Search

- Time complexity?
  - O(n + m)
    - O(m) for visiting all edges
    - O(n) for visiting all vertices
      - O(n) > O(m) for a disconnected graph
      - O(n) == O(m) for a sparse graph
      - O(n) < O(m) for a dense graph







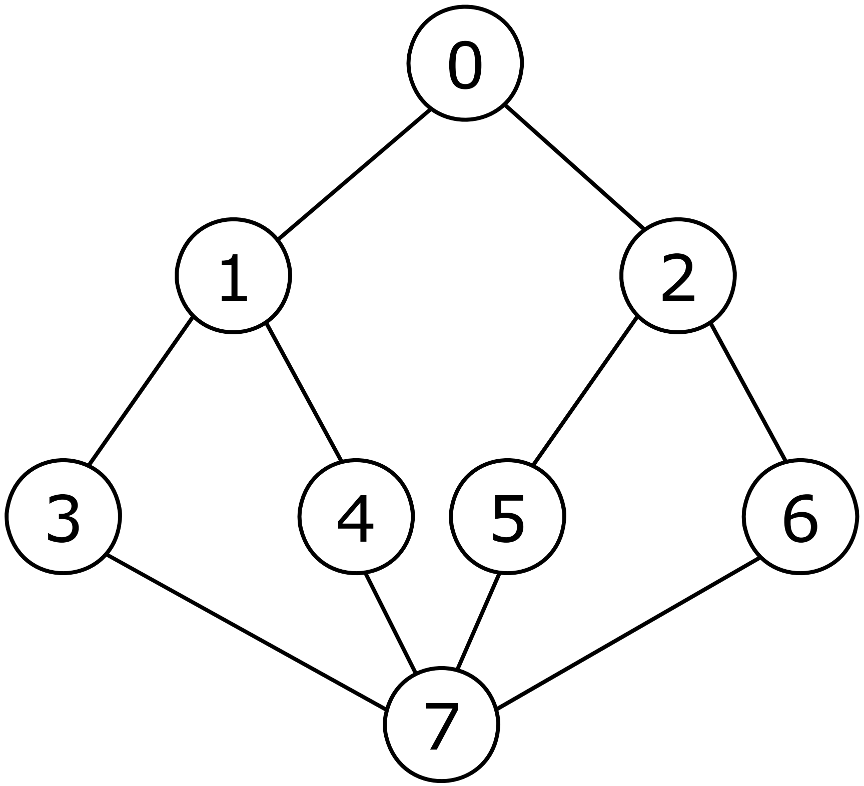
9.4.1 Depth-First Search

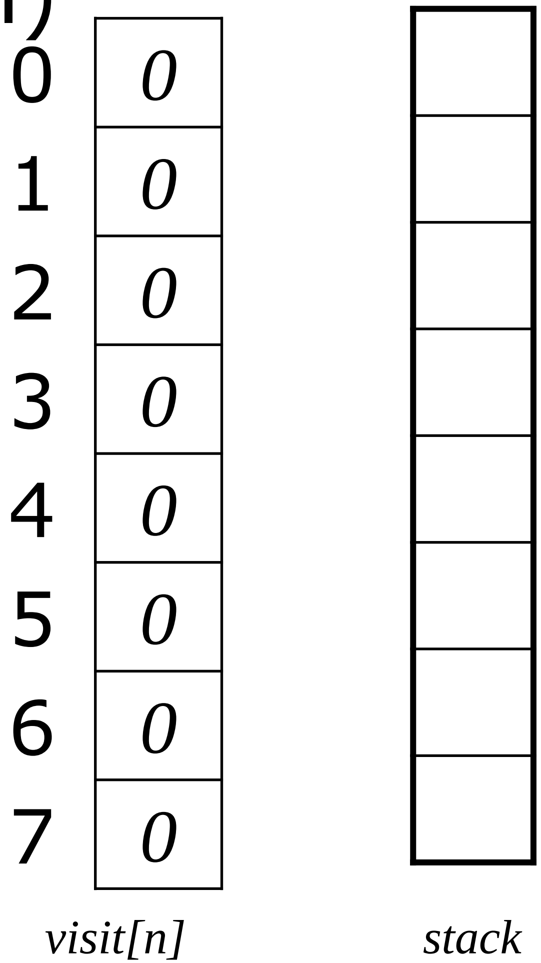
- Time complexity?
  - $O(n + m)$
  - It depends on the representation
    - A sparse graph on a adjacency matrix ?
    - A sparse graph on a adjacency list ?

	Adjacency matrix	**Adjacency list**
Disconnected	$O(n)$	$O(n)$
Sparse	$O(n^2)$	$O(n)$
Dense	$O(n^2)$	$O(n)$

9.4.1 Depth-First Search

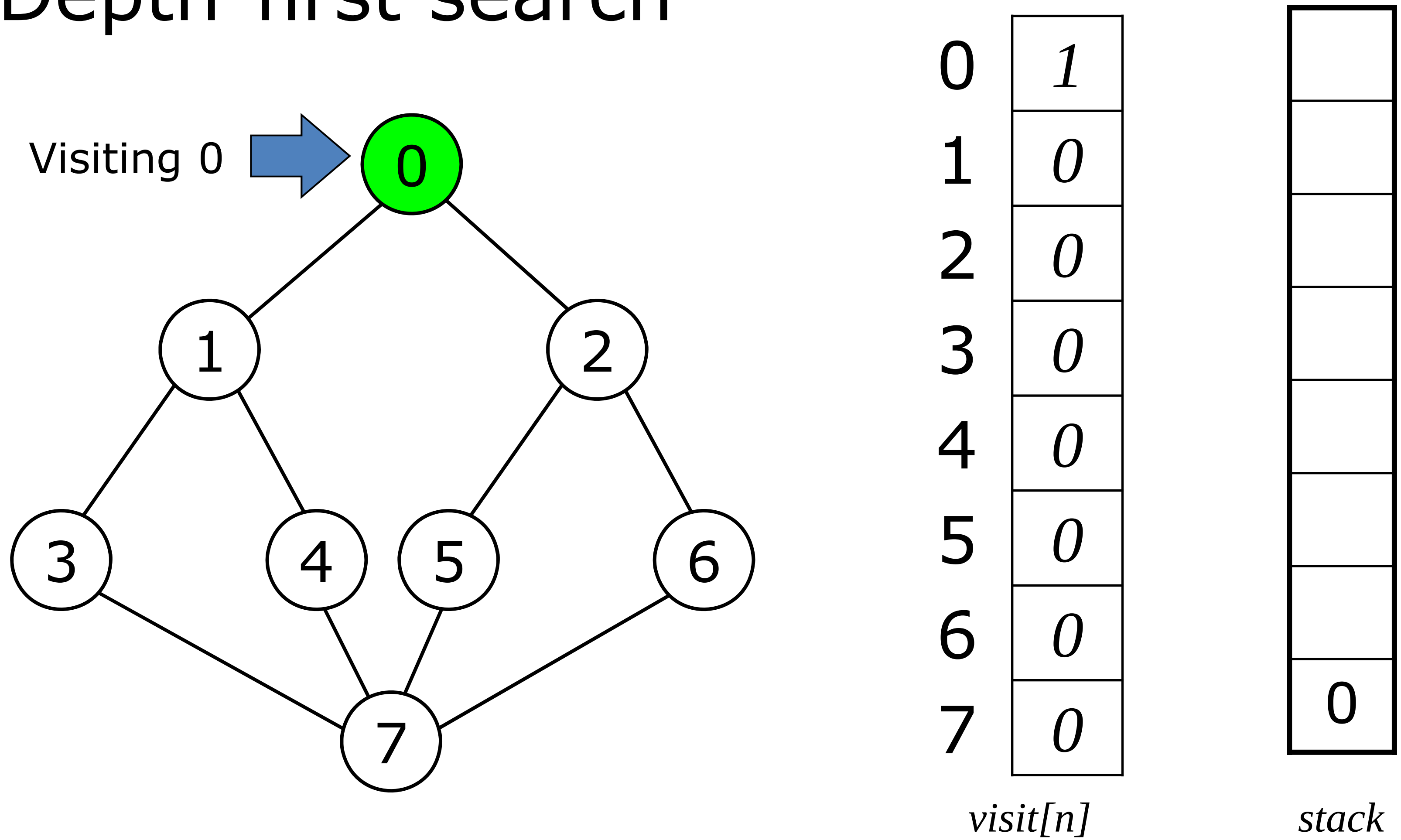
- Depth-first search (Initial)





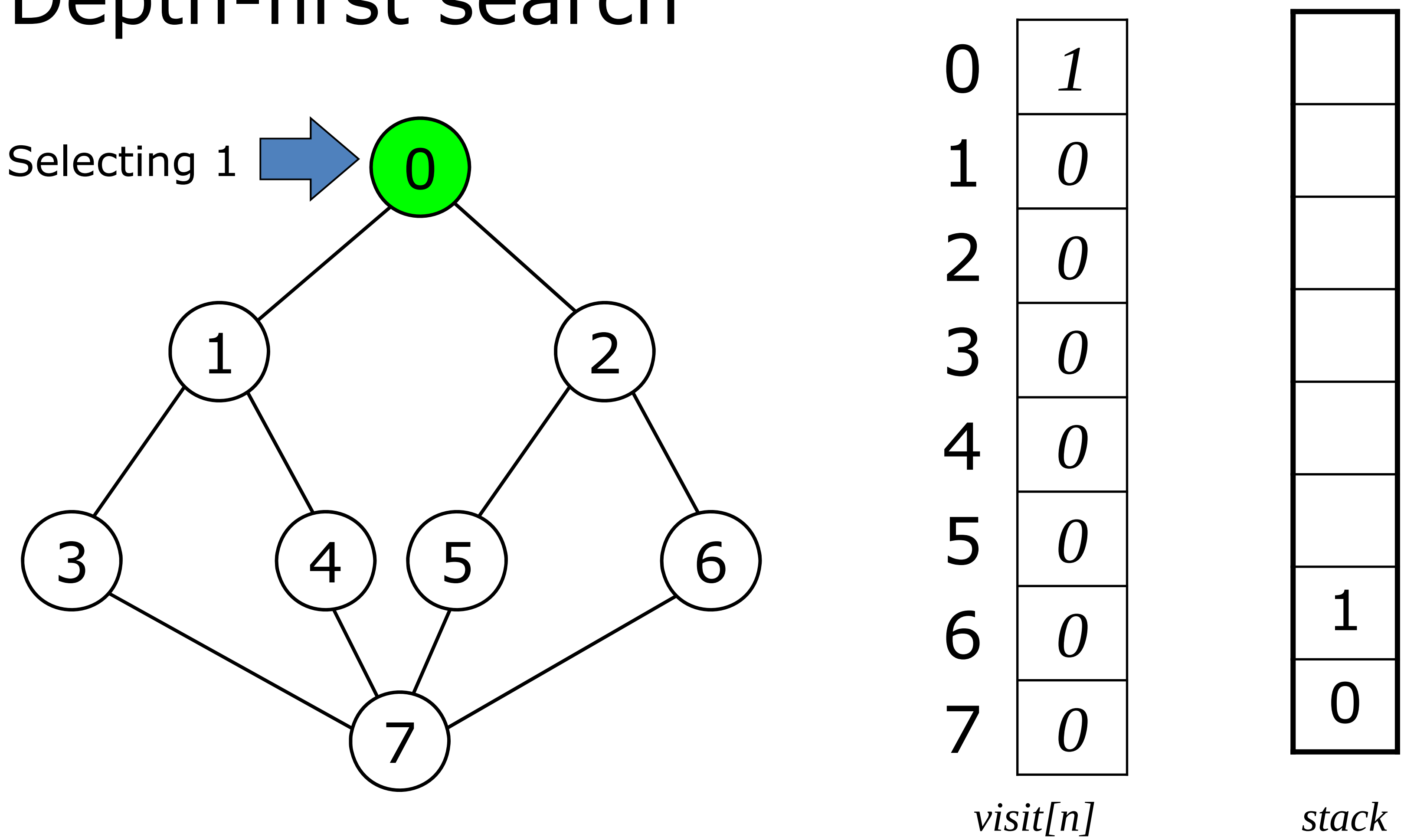
9.4.1 Depth-First Search

- Depth-first search



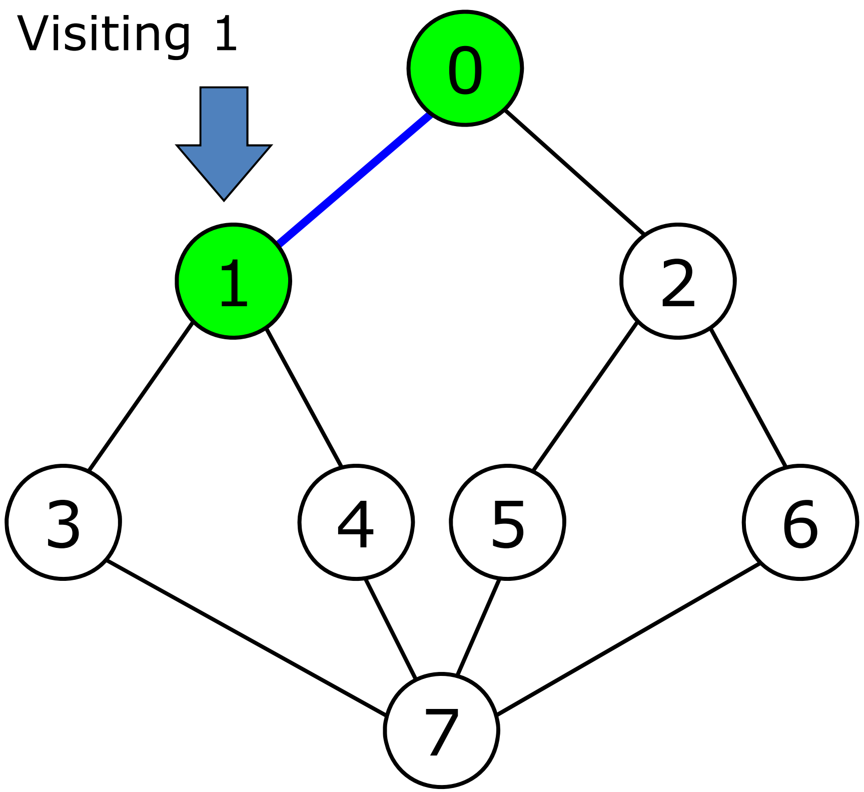
9.4.1 Depth-First Search

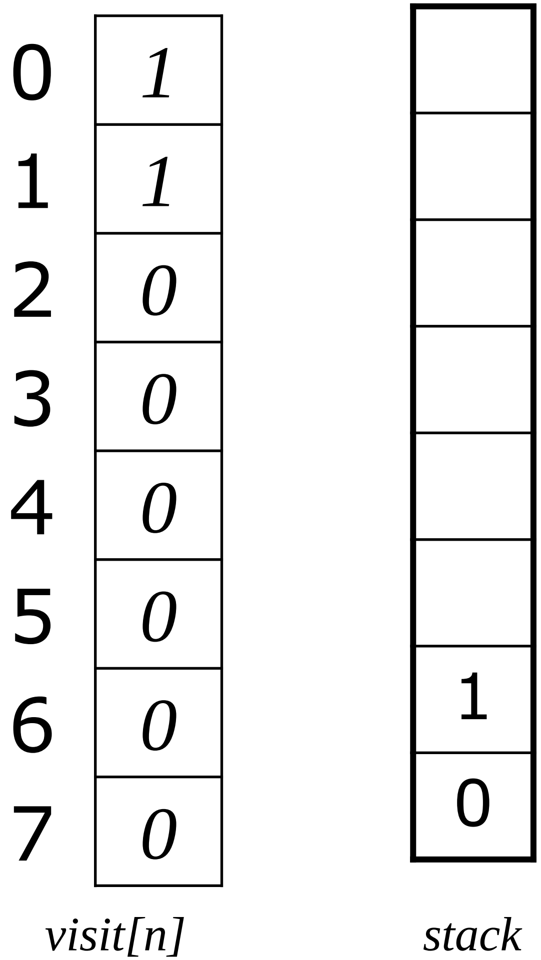
- Depth-first search



9.4.1 Depth-First Search

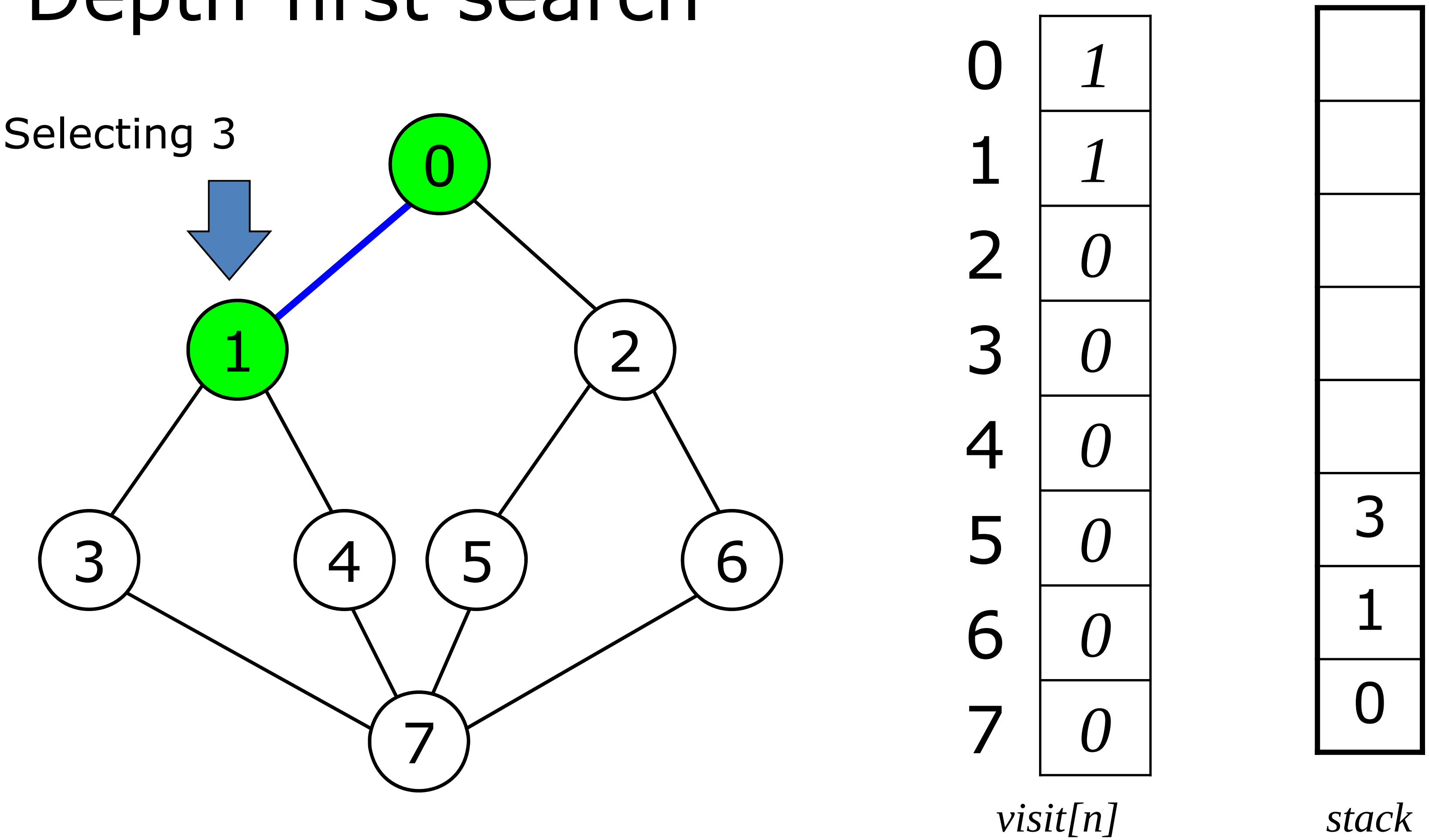
- Depth-first search





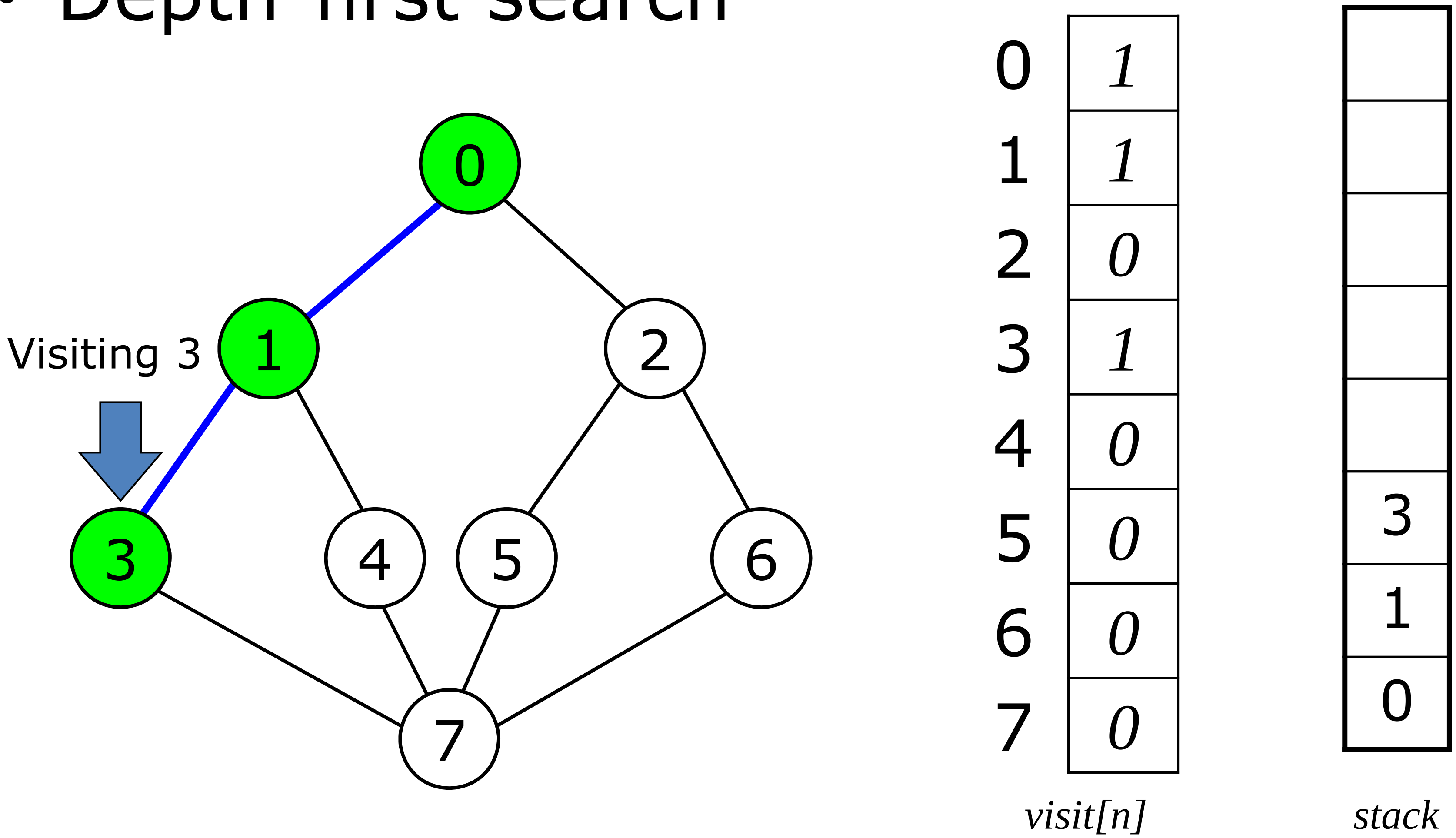
9.4.1 Depth-First Search

- Depth-first search



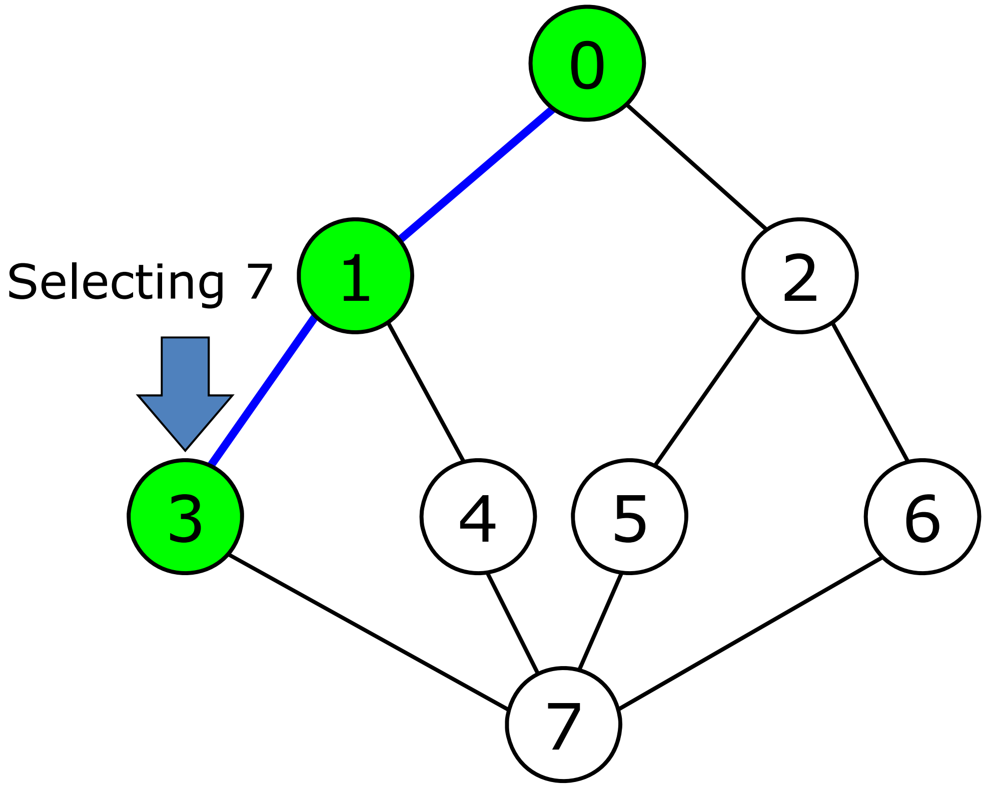
9.4.1 Depth-First Search

- Depth-first search



9.4.1 Depth-First Search

- Depth-first search



n	visit[n]
0	*1*
1	*1*
2	*0*
3	*1*
4	*0*
5	*0*
6	*0*
7	*0*

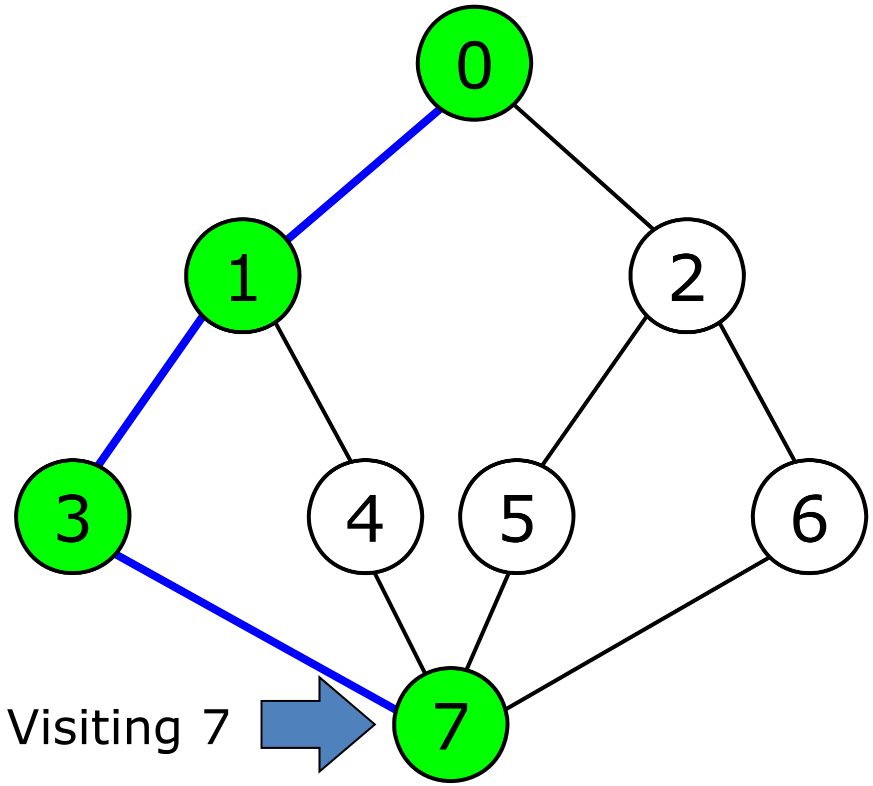
visit[n]

stack
7
3
1
0

stack

9.4.1 Depth-First Search

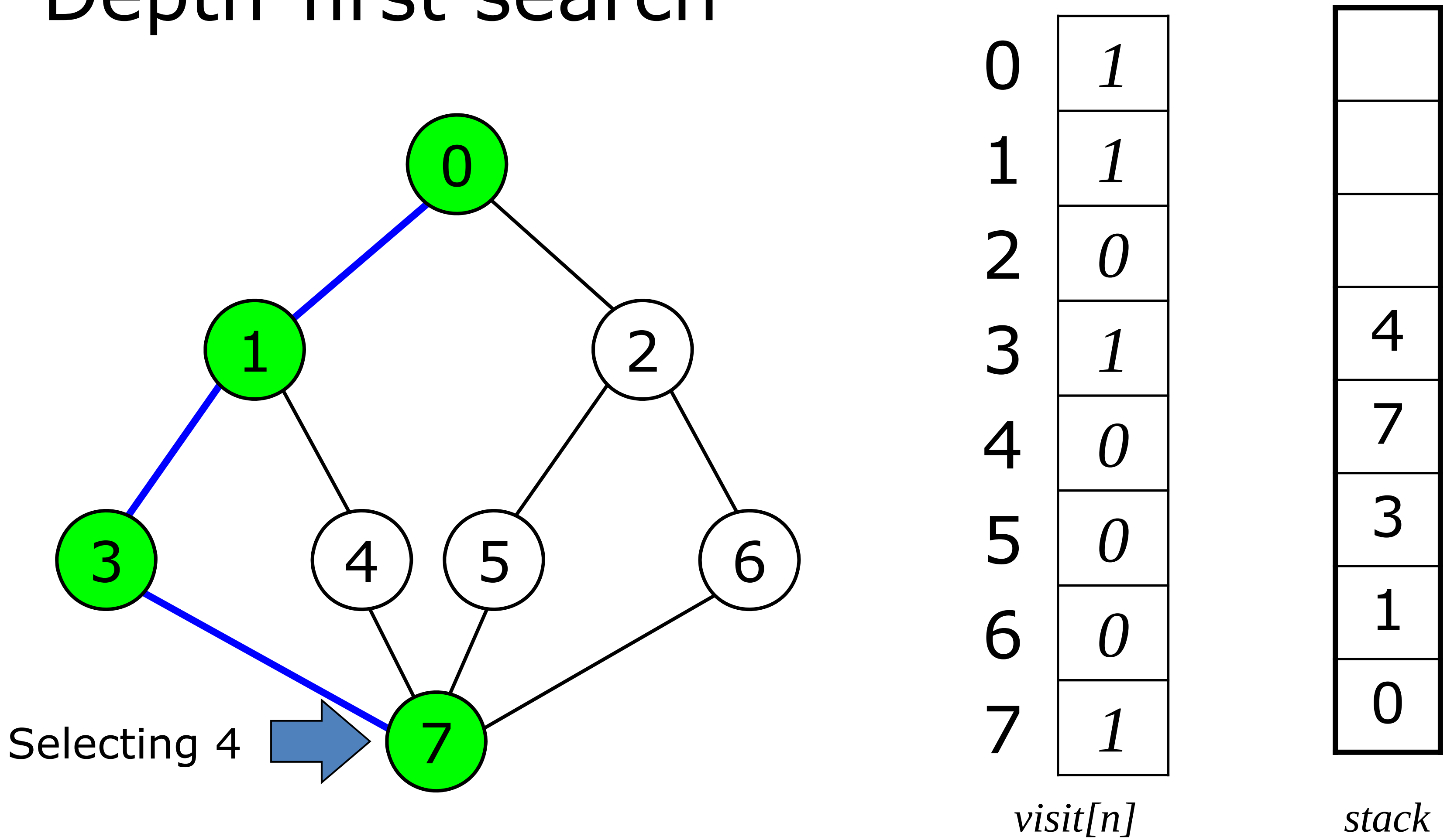
- Depth-first search



	visit[n]
0	*1*
1	*1*
2	*0*
3	*1*
4	*0*
5	*0*
6	*0*
7	*1*

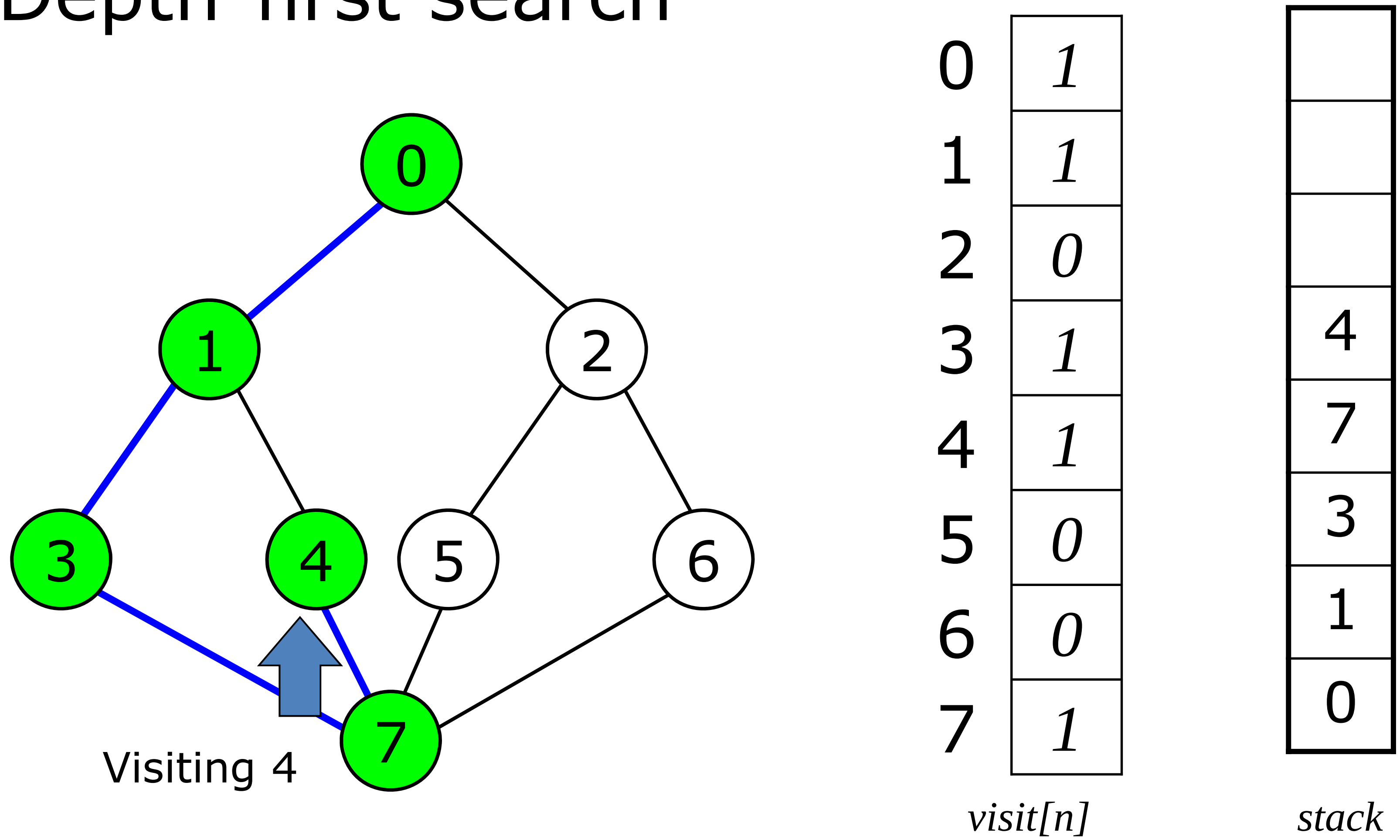
stack
7
3
1
0

- Depth-first search



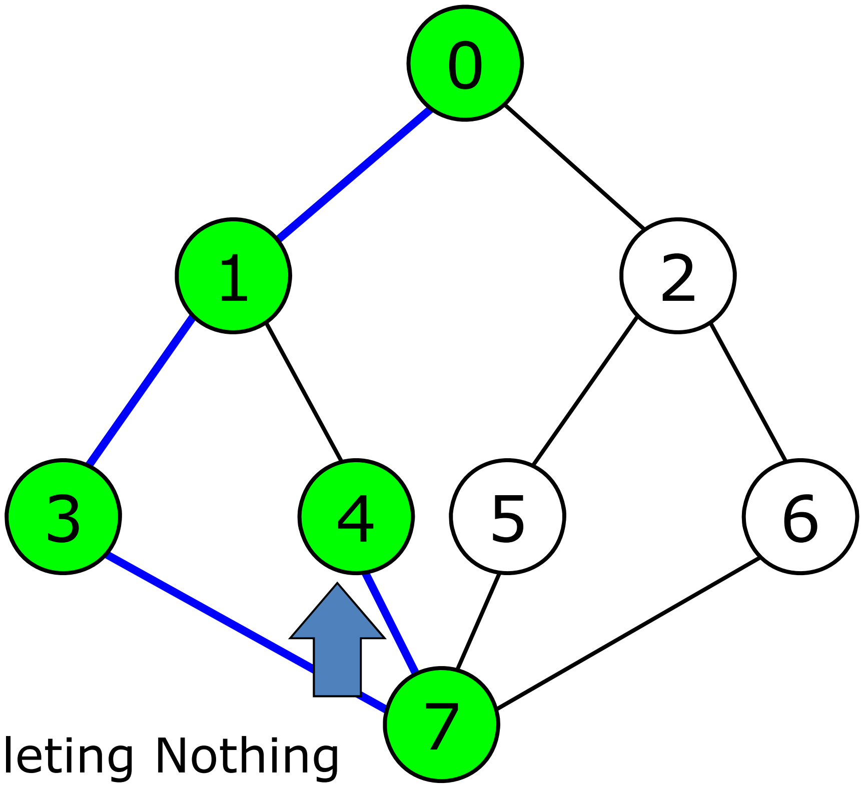
9.4.1 Depth-First Search

- Depth-first search



9.4.1 Depth-First Search

- Depth-first search



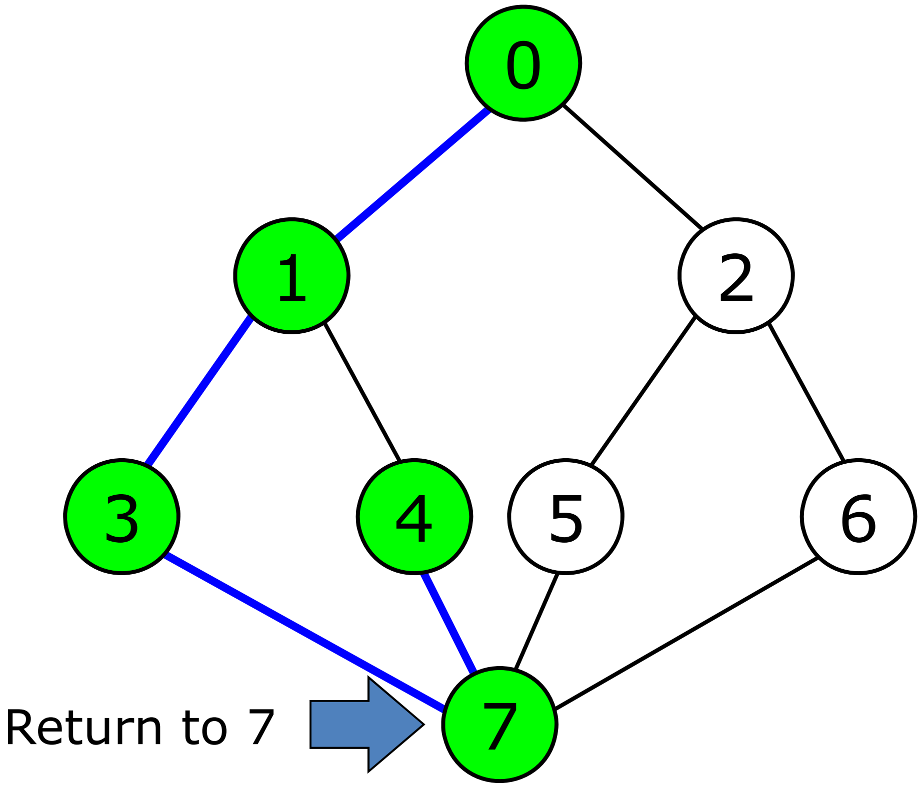
Seleting Nothing
➔ Pop a stack

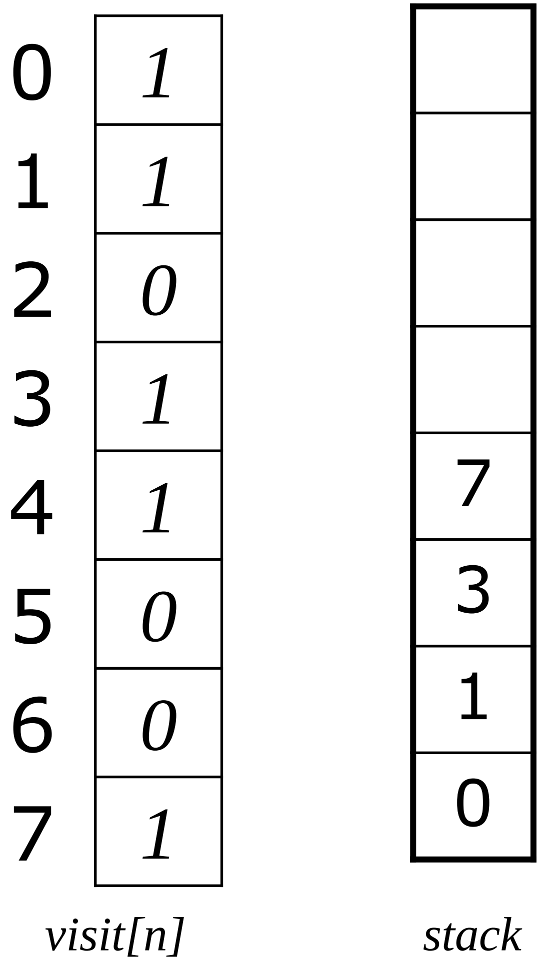
	visit[n]
0	*1*
1	*1*
2	*0*
3	*1*
4	*1*
5	*0*
6	*0*
7	*1*

stack
4
7
3
1
0

9.4.1 Depth-First Search

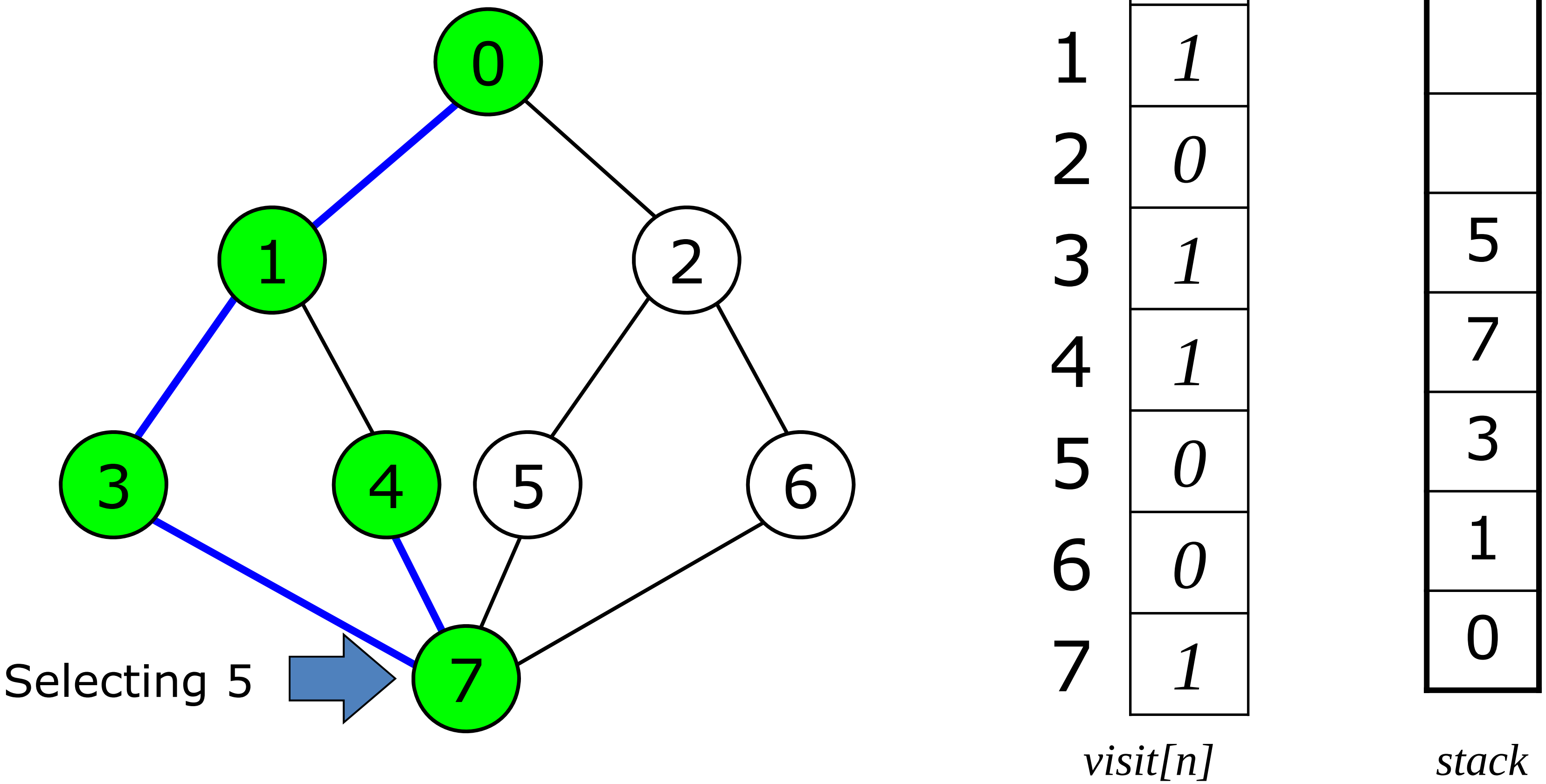
- Depth-first search





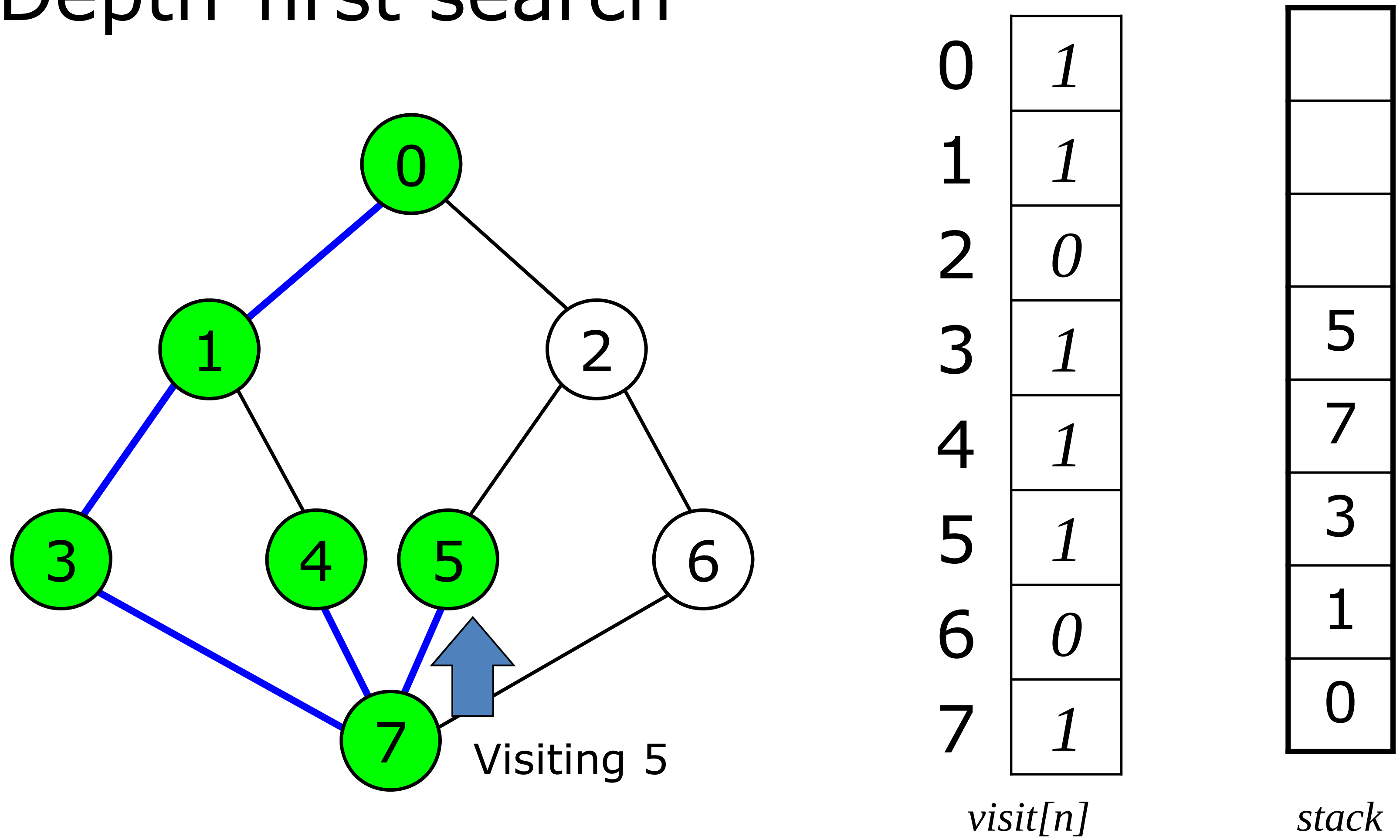
9.4.1 Depth-First Search

- Depth-first search



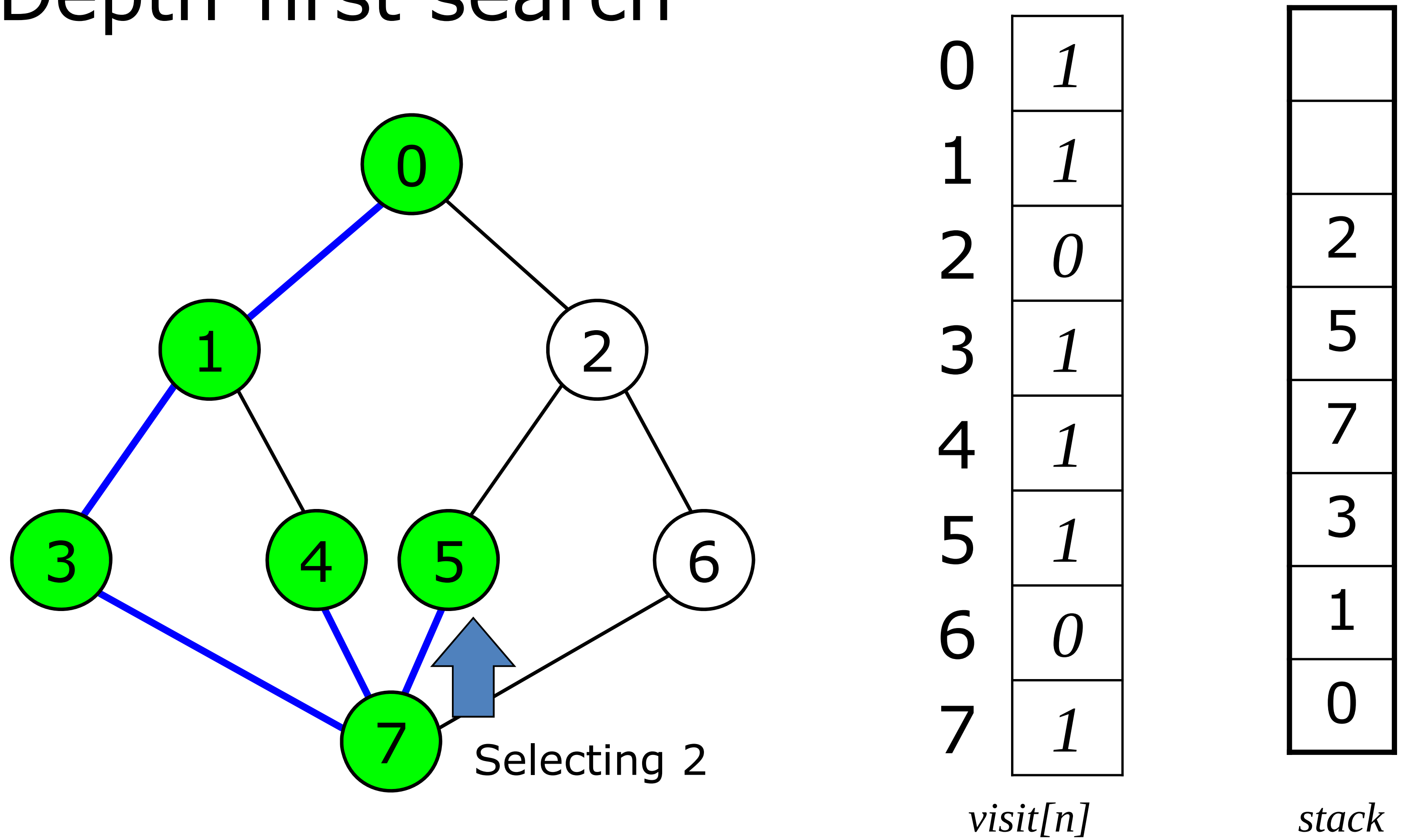
9.4.1 Depth-First Search

- Depth-first search



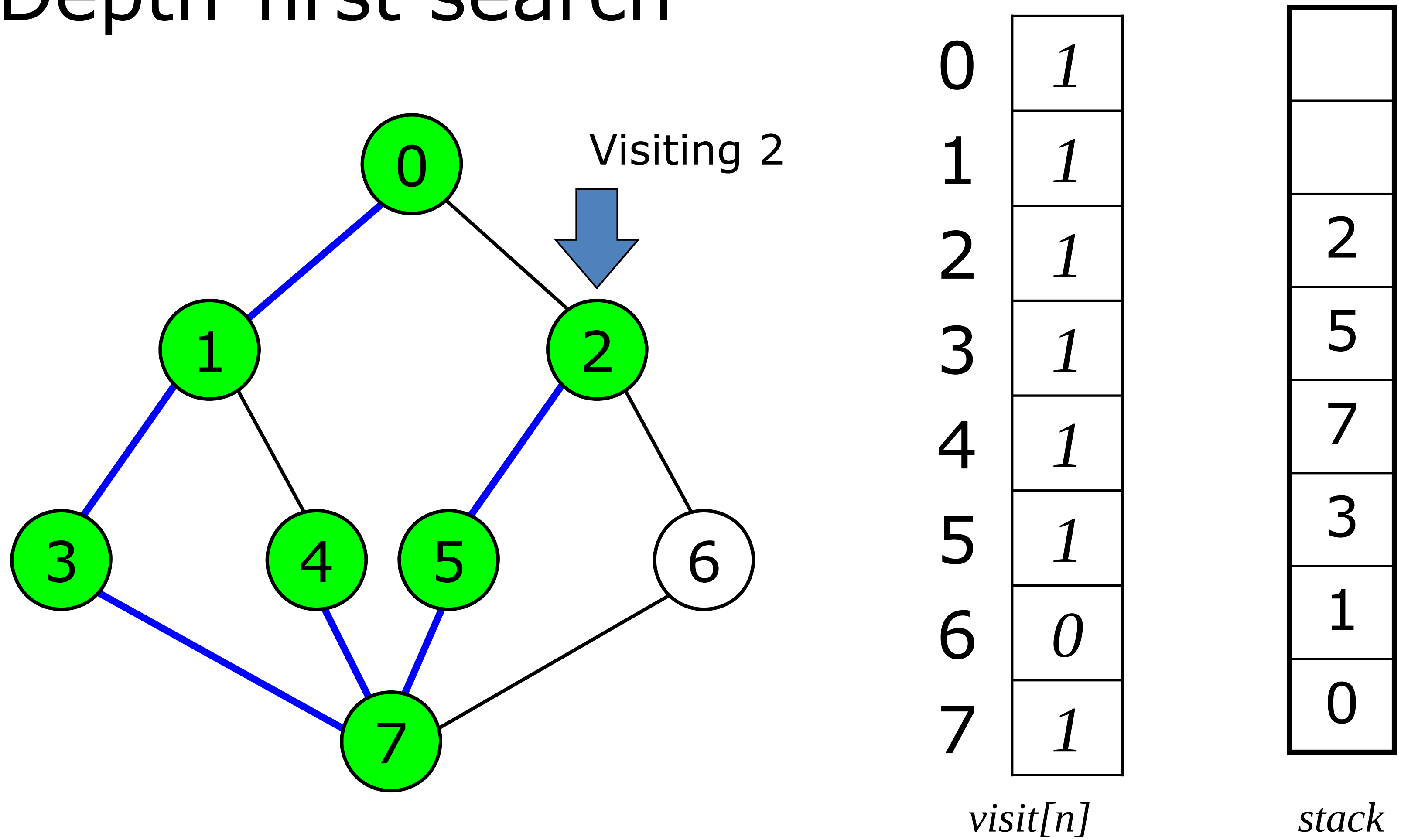
9.4.1 Depth-First Search

- Depth-first search

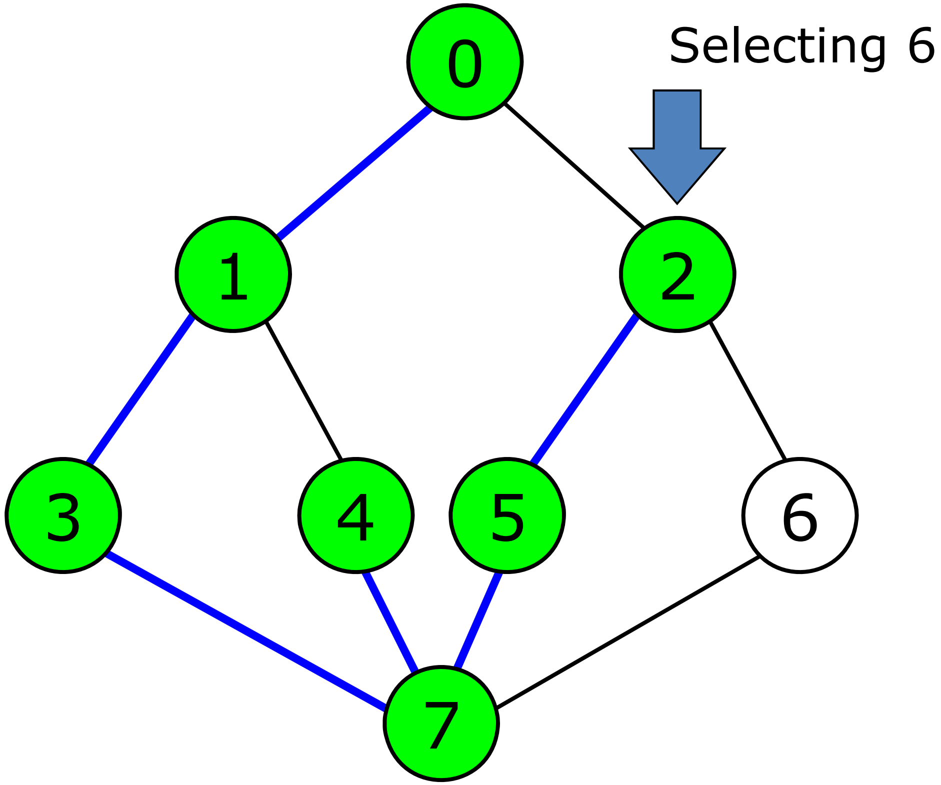


9.4.1 Depth-First Search

- Depth-first search



- Depth-first search



0	*1*
1	*1*
2	*1*
3	*1*
4	*1*
5	*1*
6	*0*
7	*1*

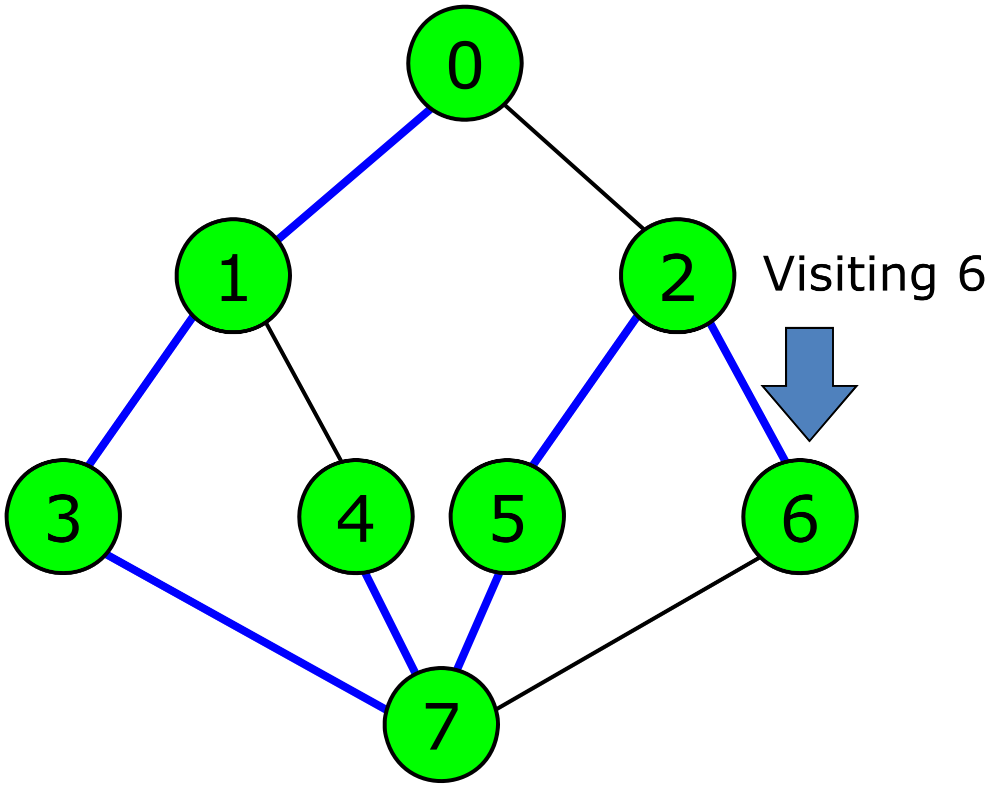
visit[n]

6
2
5
7
3
1
0

stack

9.4.1 Depth-First Search

- Depth-first search



0	*1*
1	*1*
2	*1*
3	*1*
4	*1*
5	*1*
6	*1*
7	*1*

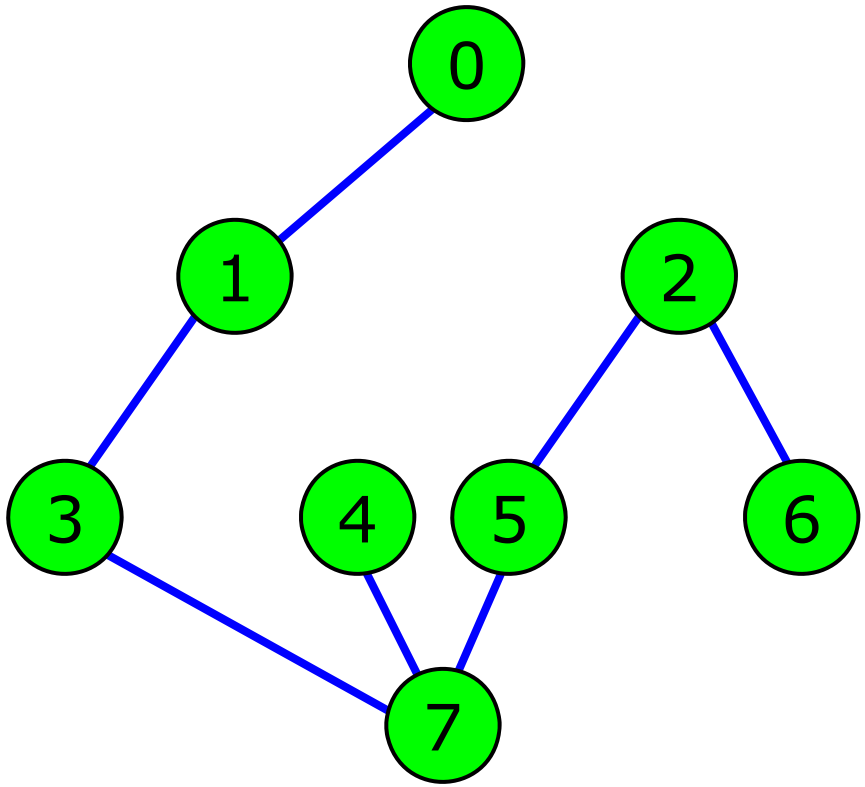
visit[n]

6
2
5
7
3
1
0

stack

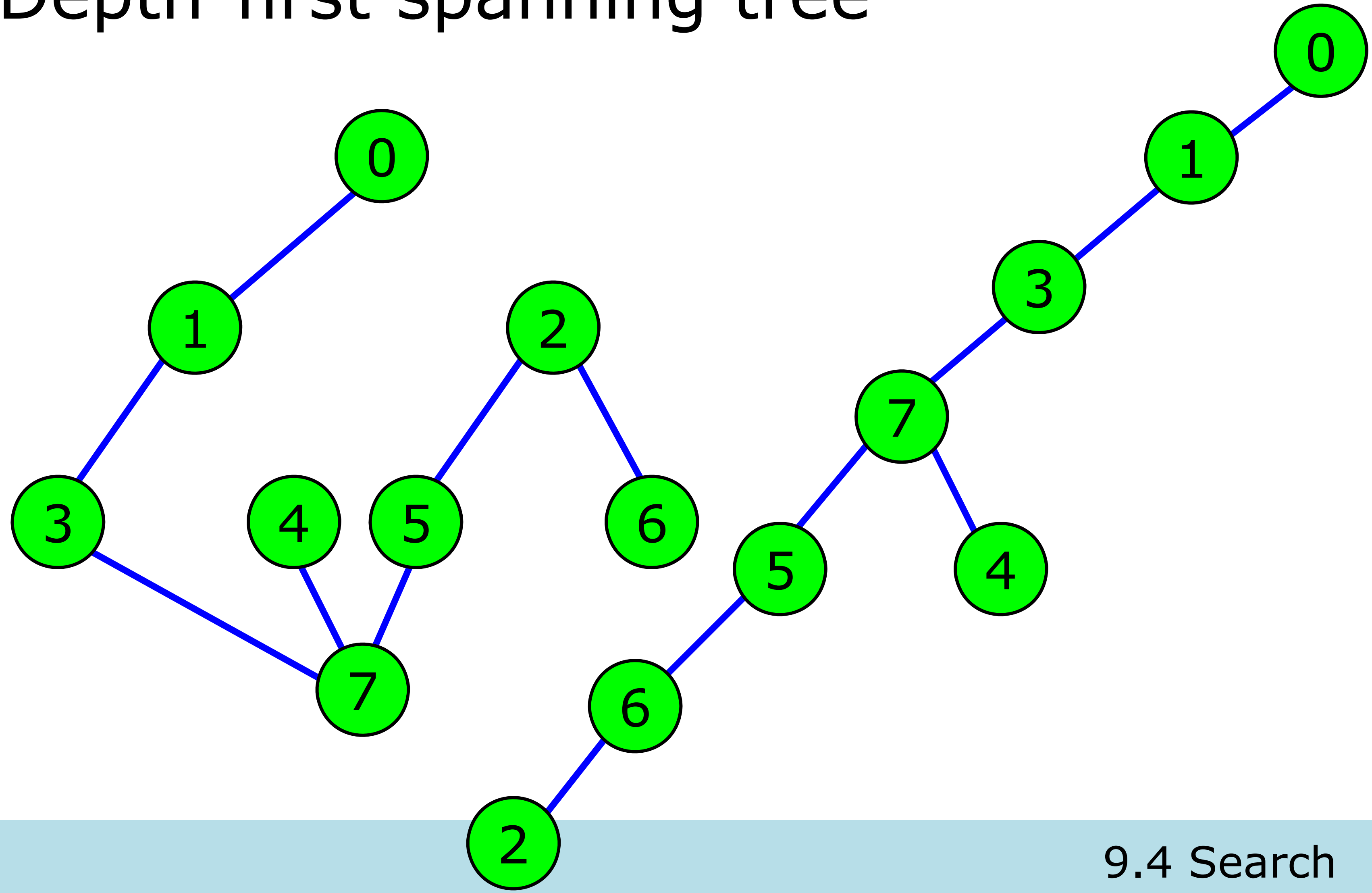
9.4.1 Depth-First Search

- Depth-first spanning tree



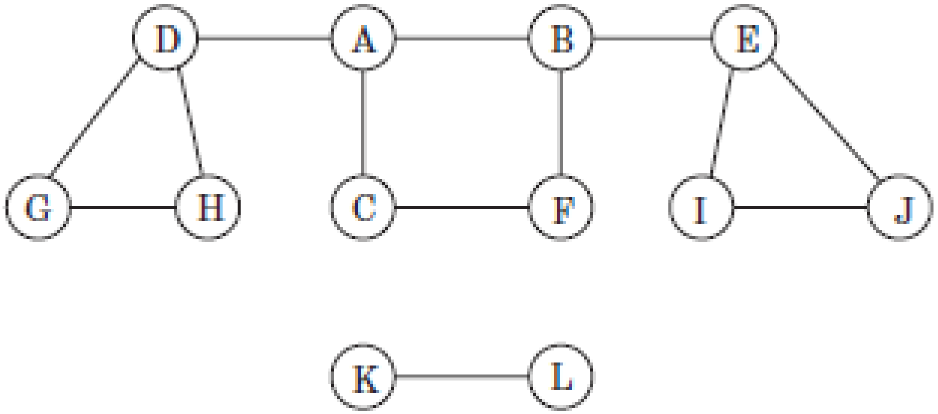
9.4.1 Depth-First Search

- Depth-first spanning tree



9.4.1 Depth-First Search

- Exercise



9.4.2 Breadth-First Search

- Breadth-first search
  - Use a queue
  - In visiting a vertex v,
    - mark all the adjacent vertices as visited
    - add the vertices to the queue

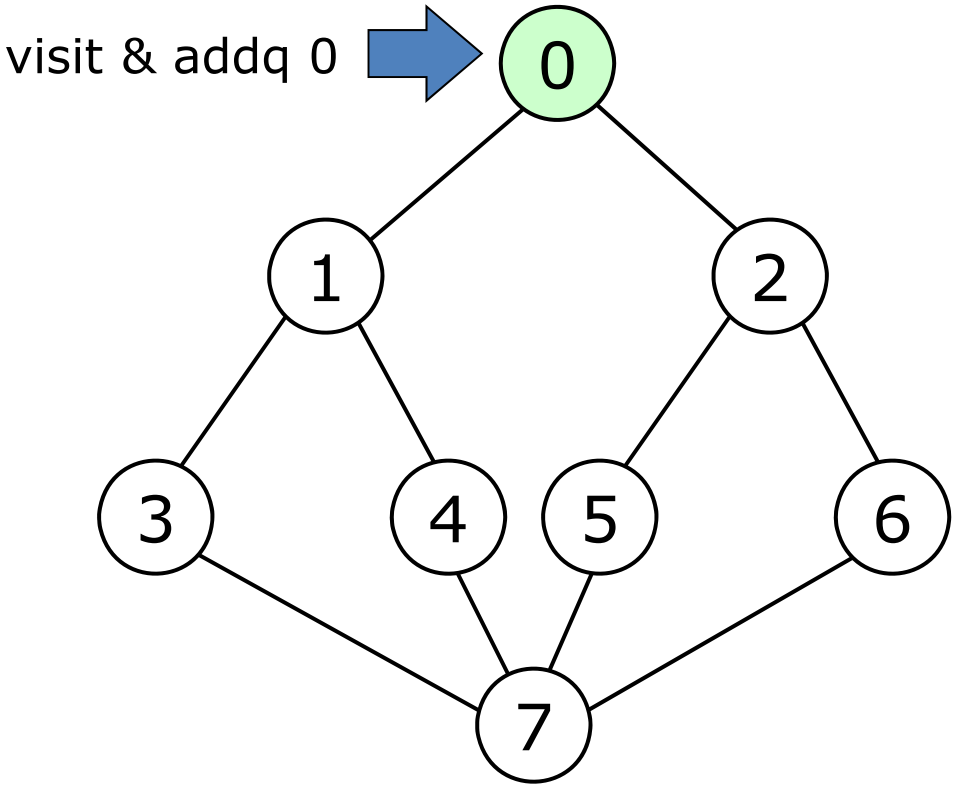
9.4.2 Breadth-First Search

```
void bfs ( int v )
{
    nodePointer w;
    front = rear = NULL;
    visit[v] = TRUE;
    addq ( v );
    while ( front ) {
       v = deleteq ( );
       for ( w = graph[v]; w; w = w->link )
           if ( !visit[w->vertex] ) {
              visit[w->vertex] = TRUE;
              addq ( w->vertex );
           }
    }
}
```

9.4.2 Depth-First Search

- Time complexity?
  - O(n + m)
    - O(m) for visiting all edges
    - O(n) for visiting all vertices
      - O(n) > O(m) for a disconnected graph
      - O(n) == O(m) for a sparse graph
      - O(n) < O(m) for a dense graph
  - It depends on the representation
    - A sparse graph on a adjacency matrix ?
    - A sparse graph on a adjacency list ?

9.4.2 Breadth-First Search

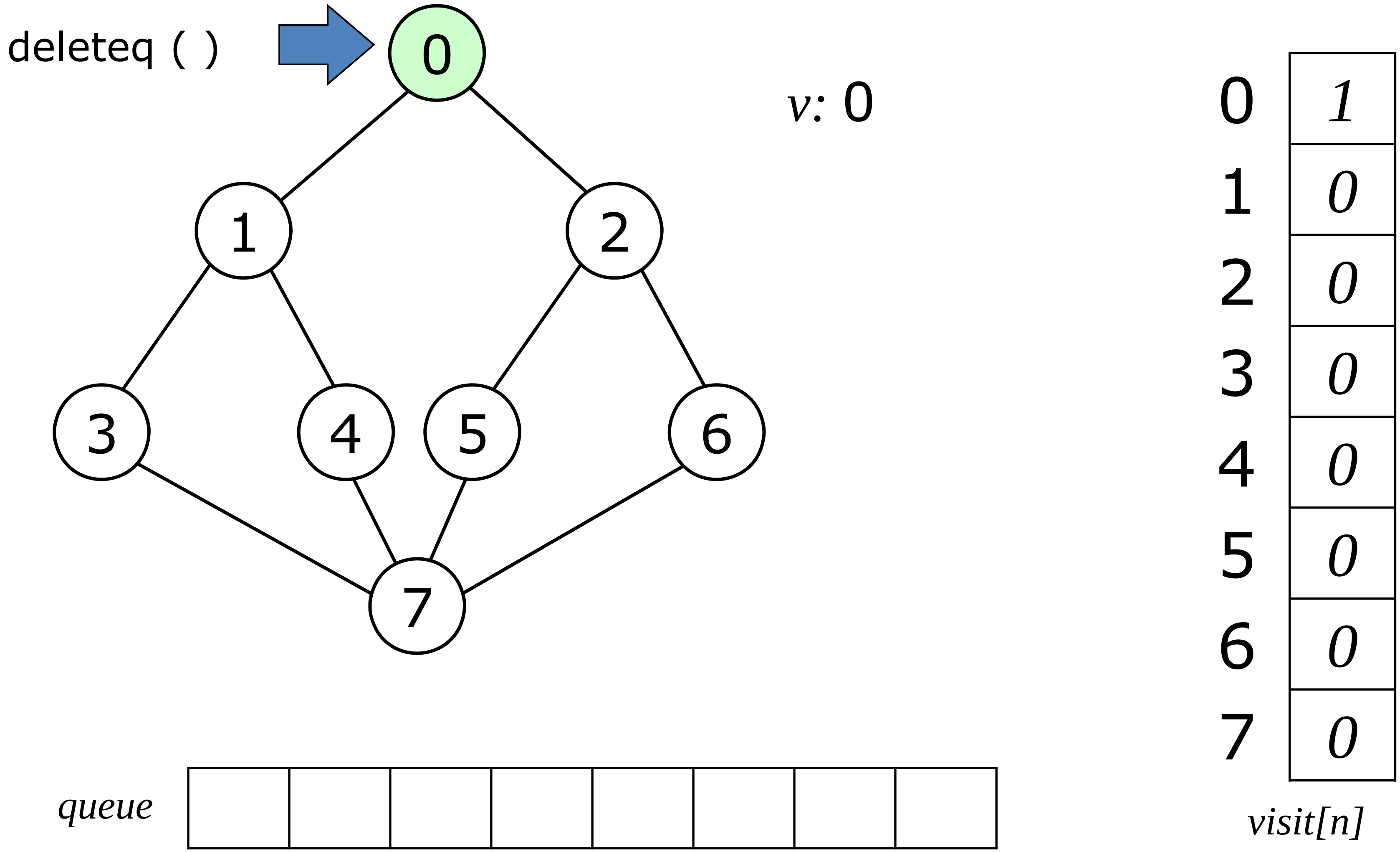


	visit[n]
0	*1*
1	*0*
2	*0*
3	*0*
4	*0*
5	*0*
6	*0*
7	*0*

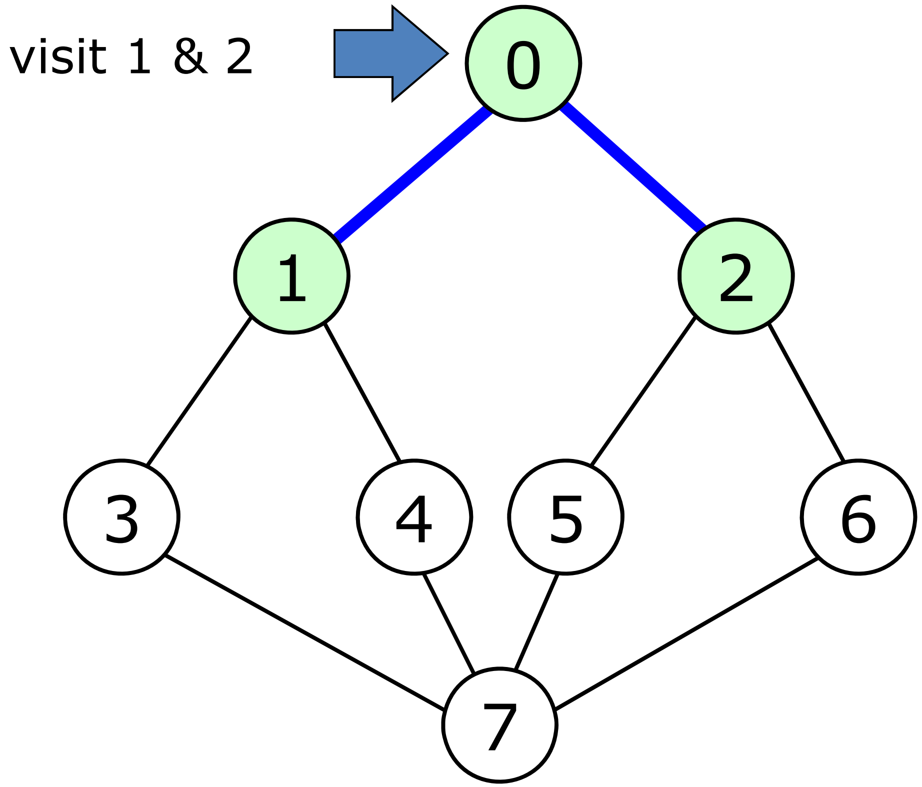
visit[n]

queue	0							

9.4.2 Breadth-First Search



9.4.2 Breadth-First Search



v: 0

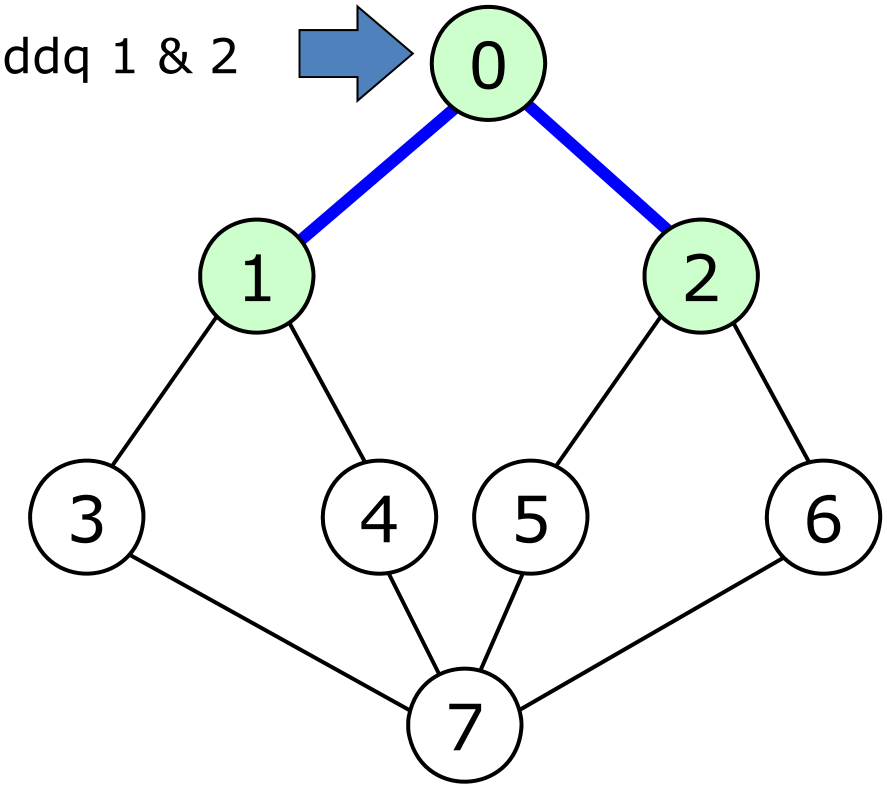
0	*1*
1	*1*
2	*1*
3	*0*
4	*0*
5	*0*
6	*0*
7	*0*

visit[n]

queue

9.4.2 Breadth-First Search

addq 1 & 2



v: 0

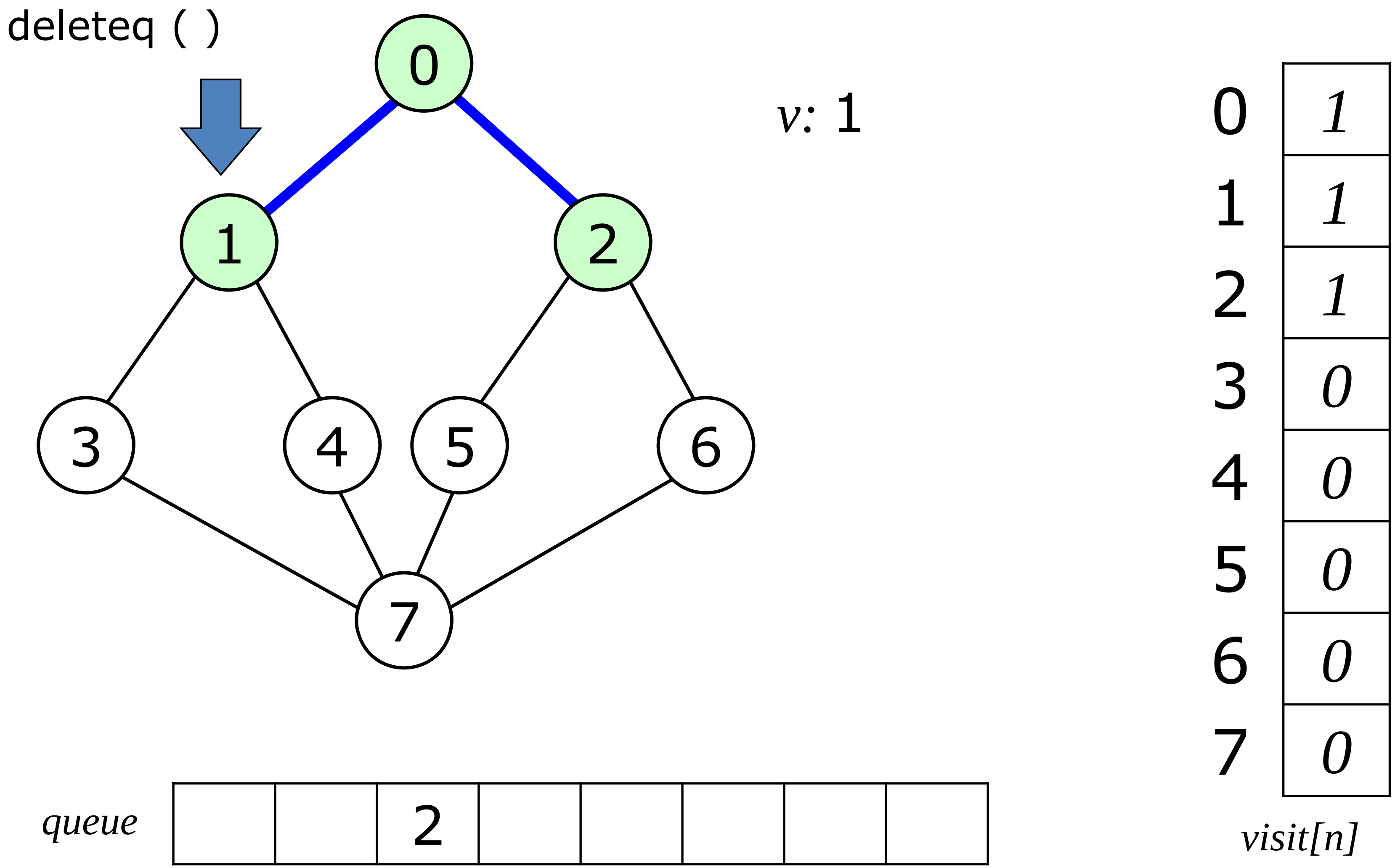
0	*1*
1	*1*
2	*1*
3	*0*
4	*0*
5	*0*
6	*0*
7	*0*

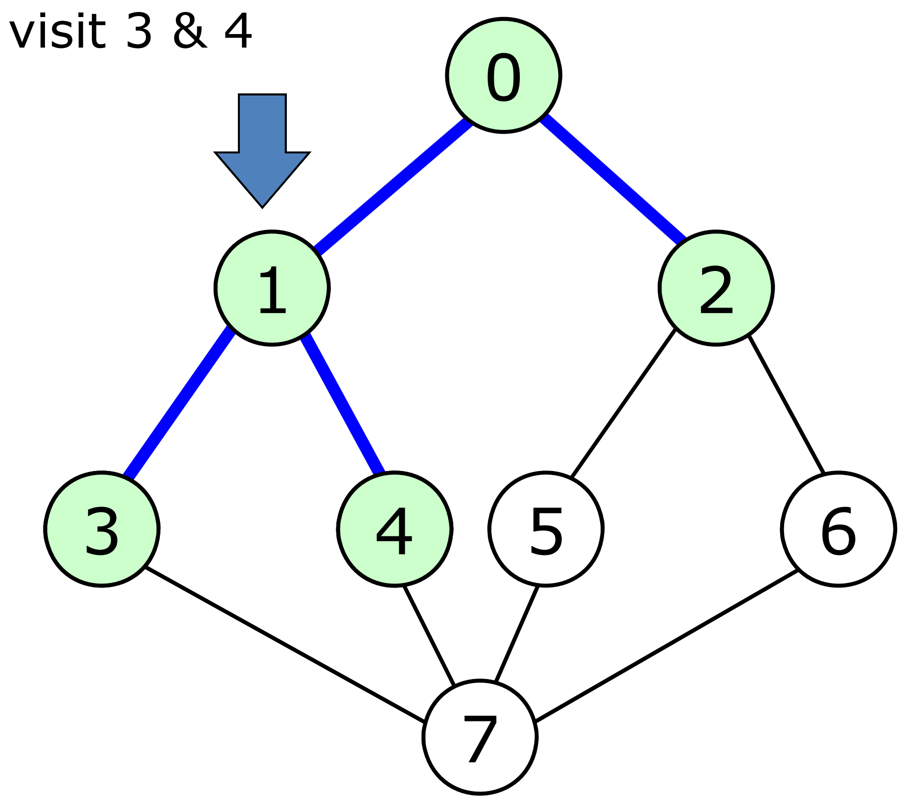
visit[n]

queue

	1	2					

9.4.2 Breadth-First Search





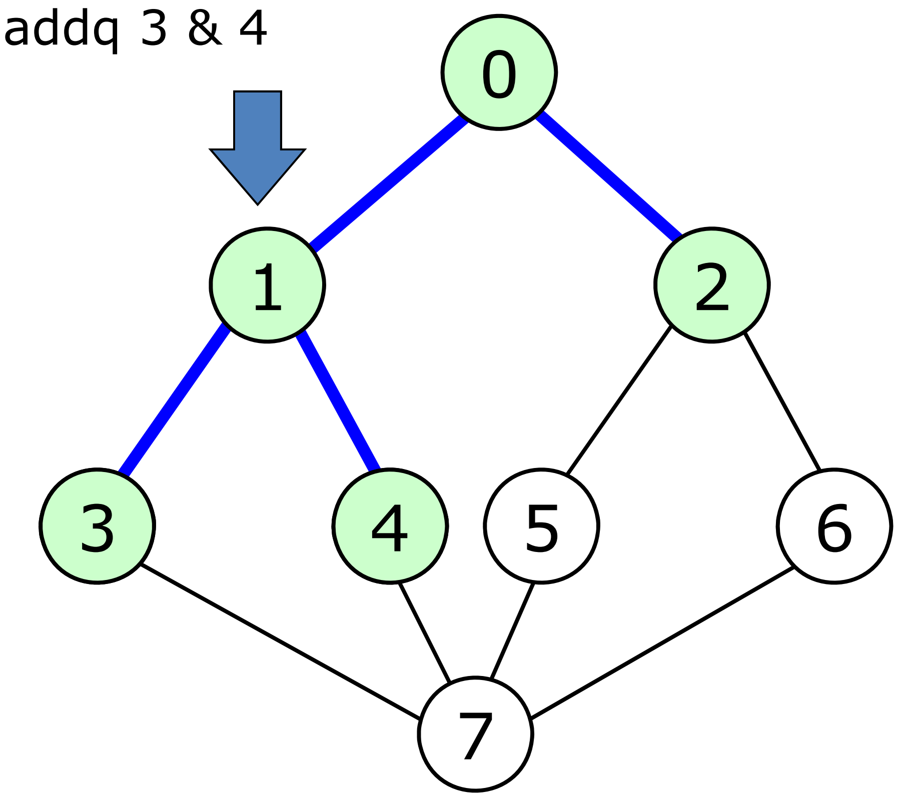
v: 1

	visit[n]
0	*1*
1	*1*
2	*1*
3	*1*
4	*1*
5	*0*
6	*0*
7	*0*

queue

		2					

9.4.2 Breadth-First Search



v: 1

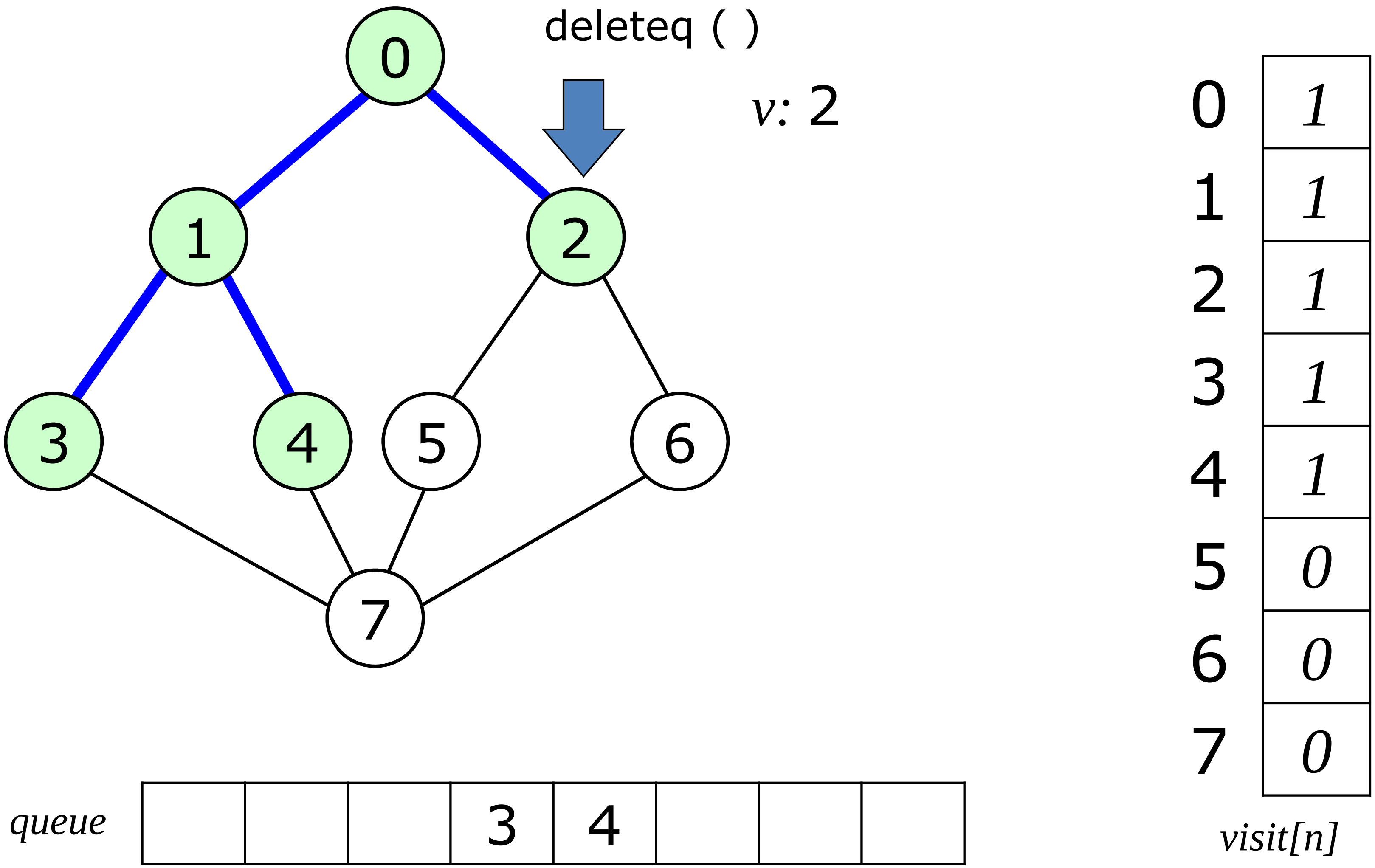
0	*1*
1	*1*
2	*1*
3	*1*
4	*1*
5	*0*
6	*0*
7	*0*

visit[n]

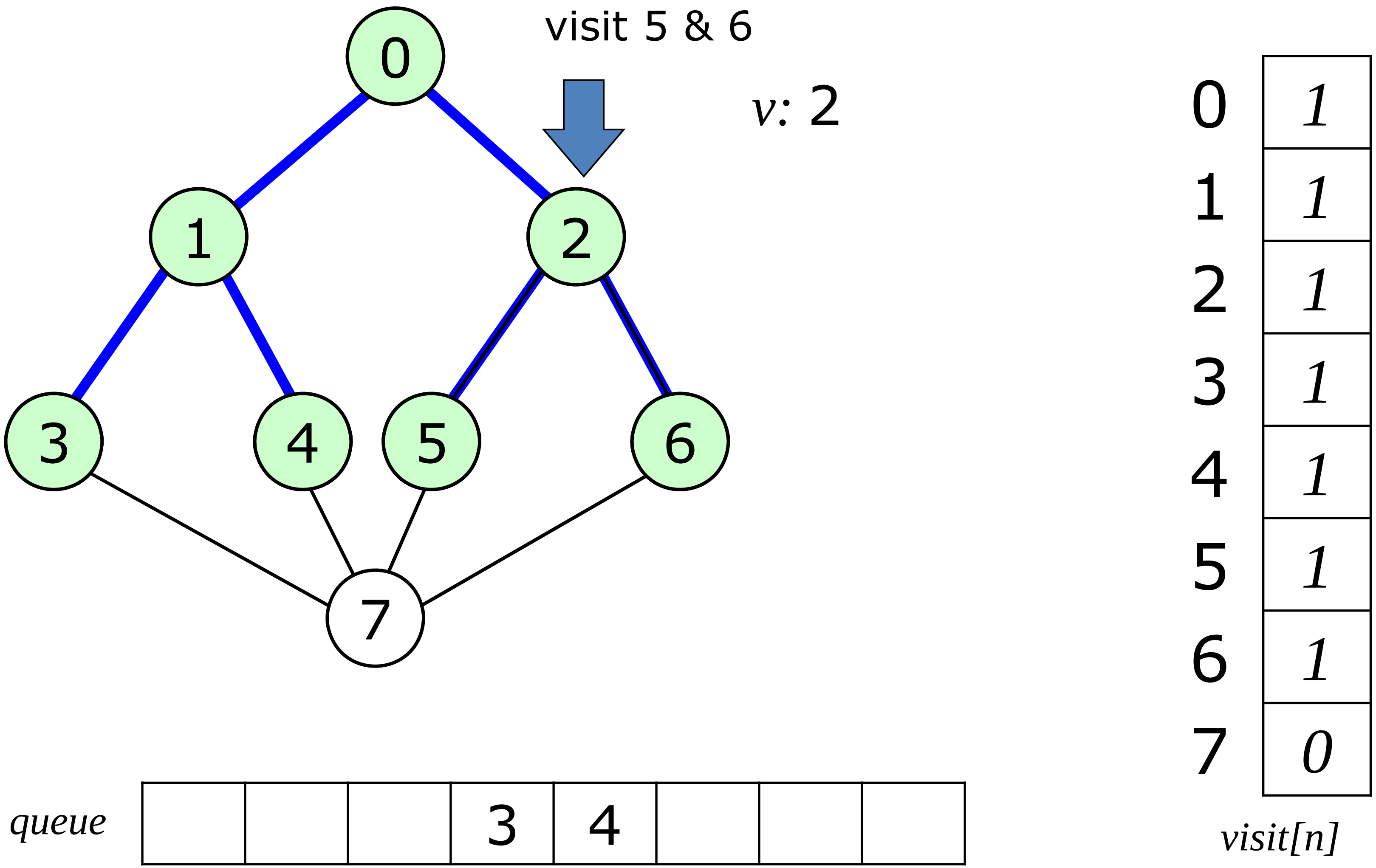
queue

		2	3	4			

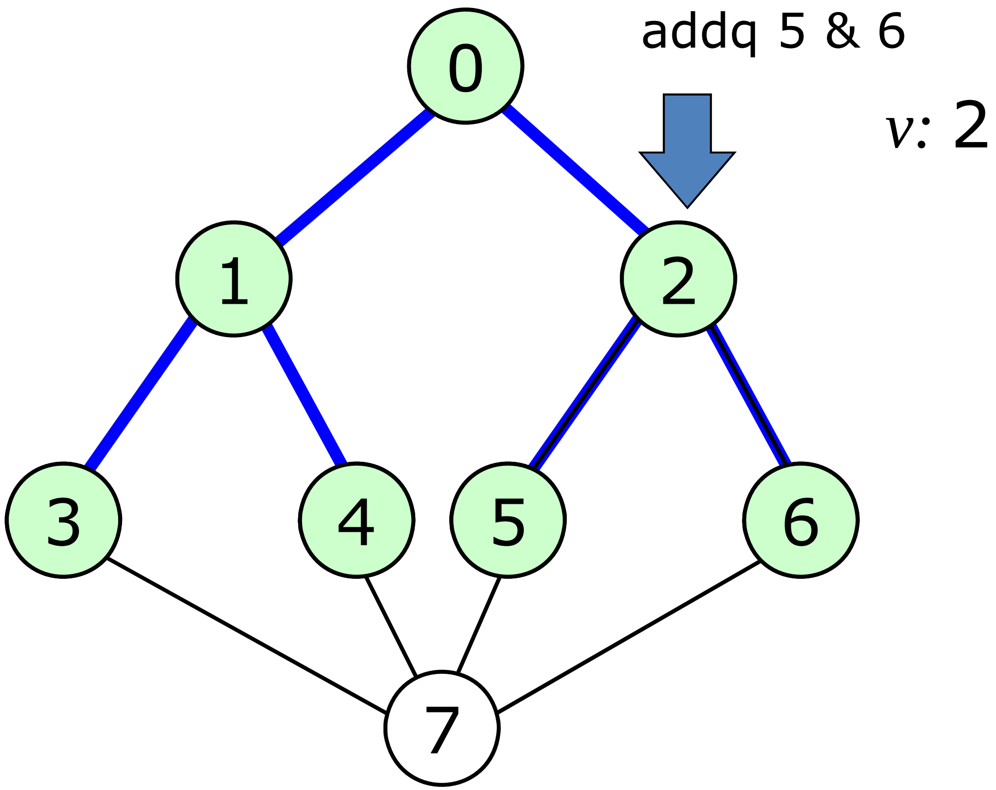
9.4.2 Breadth-First Search



9.4.2 Breadth-First Search



9.4.2 Breadth-First Search



0	*1*
1	*1*
2	*1*
3	*1*
4	*1*
5	*1*
6	*1*
7	*0*

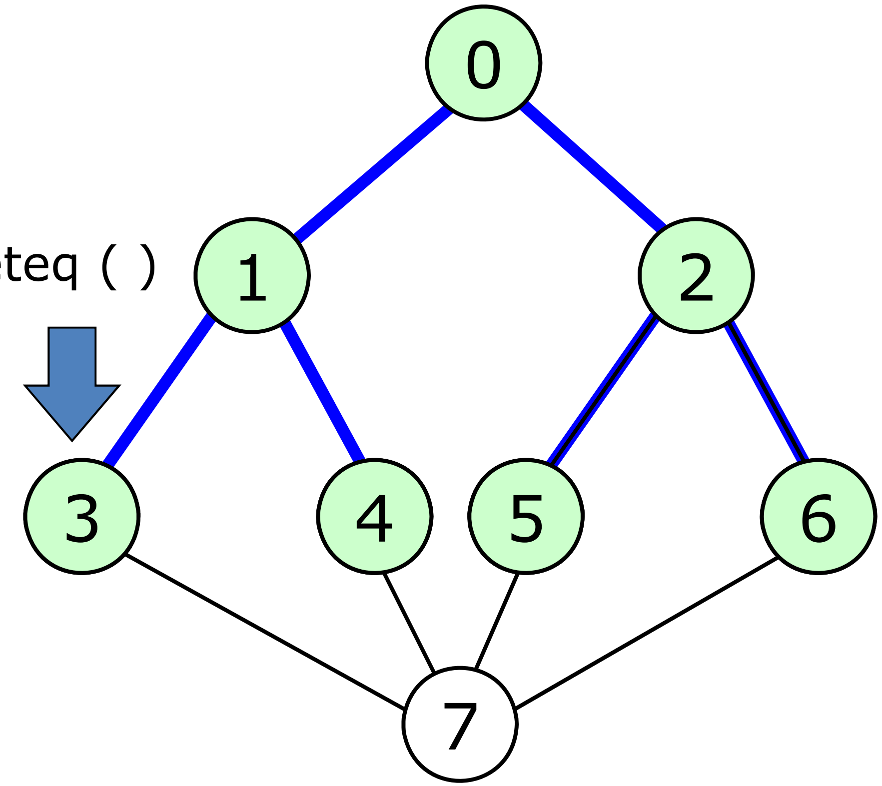
visit[n]

queue

			3	4	5	6	

9.4.2 Breadth-First Search

deleteq ( )



v: 3

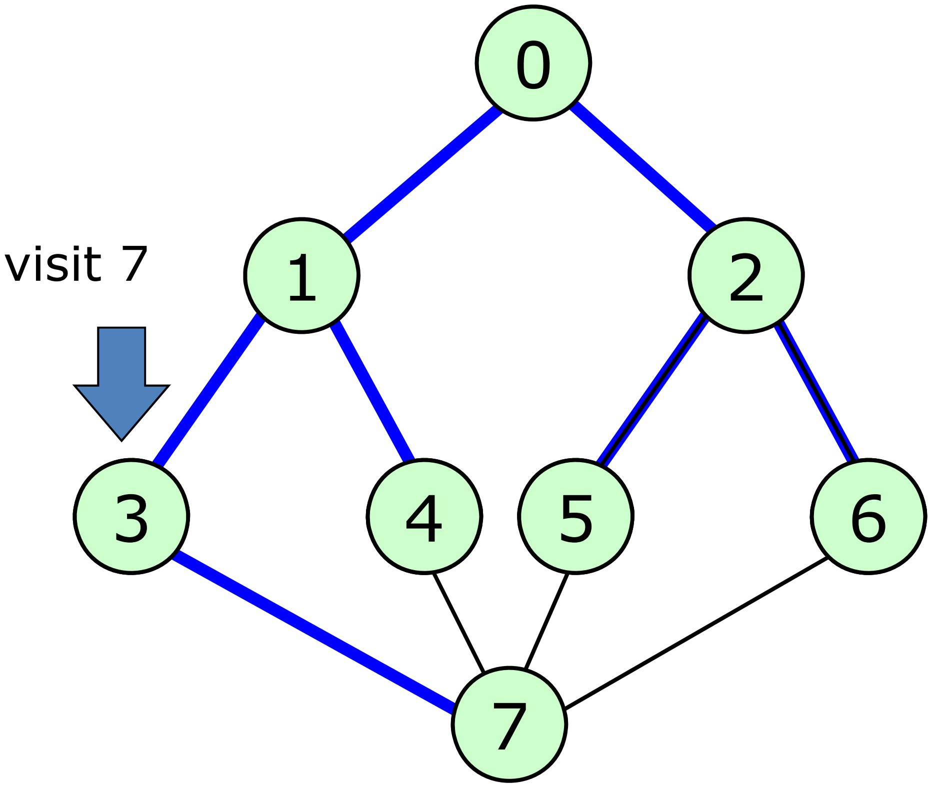
0	*1*
1	*1*
2	*1*
3	*1*
4	*1*
5	*1*
6	*1*
7	*0*

visit[n]

queue

				4	5	6	

9.4.2 Breadth-First Search



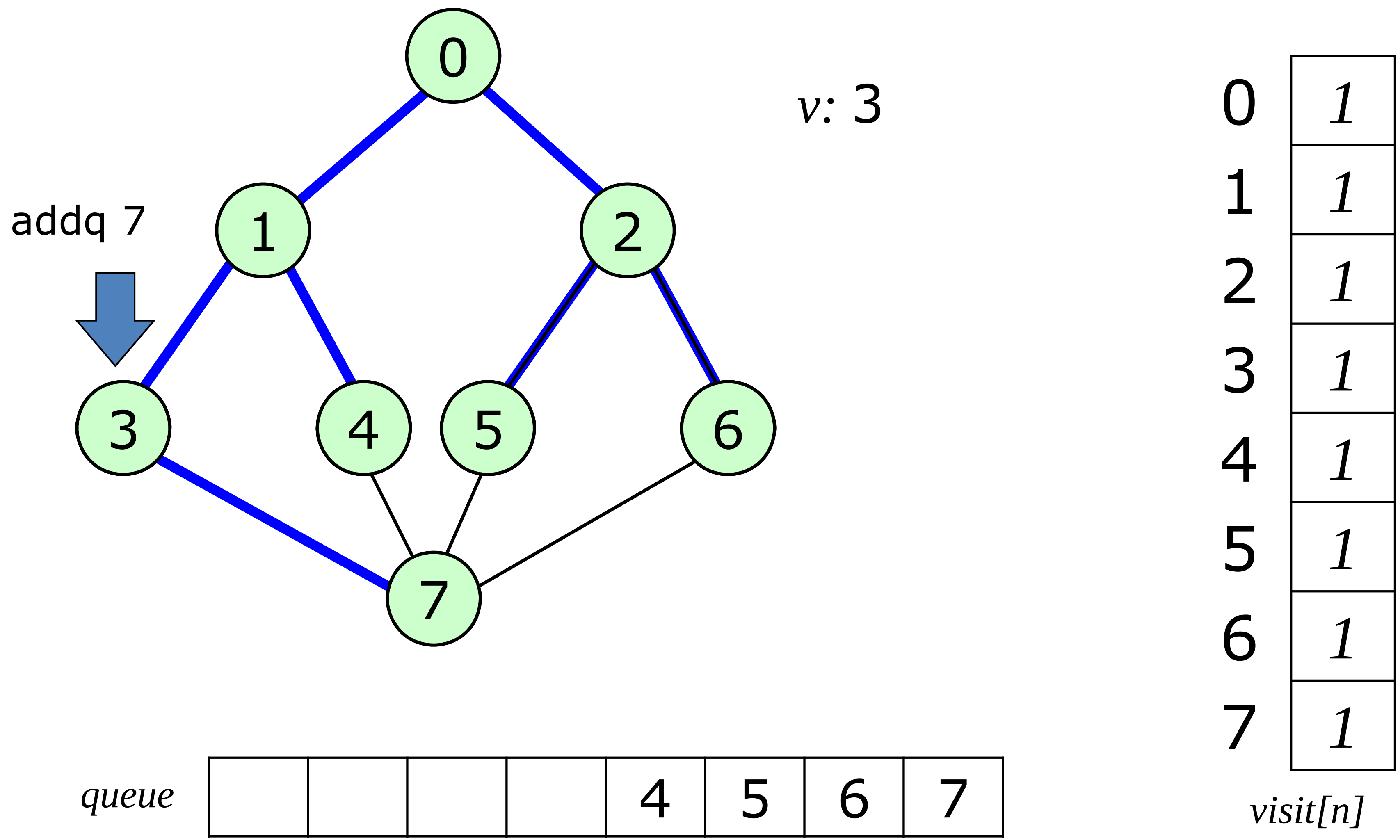
v: 3

	visit[n]
0	*1*
1	*1*
2	*1*
3	*1*
4	*1*
5	*1*
6	*1*
7	*1*

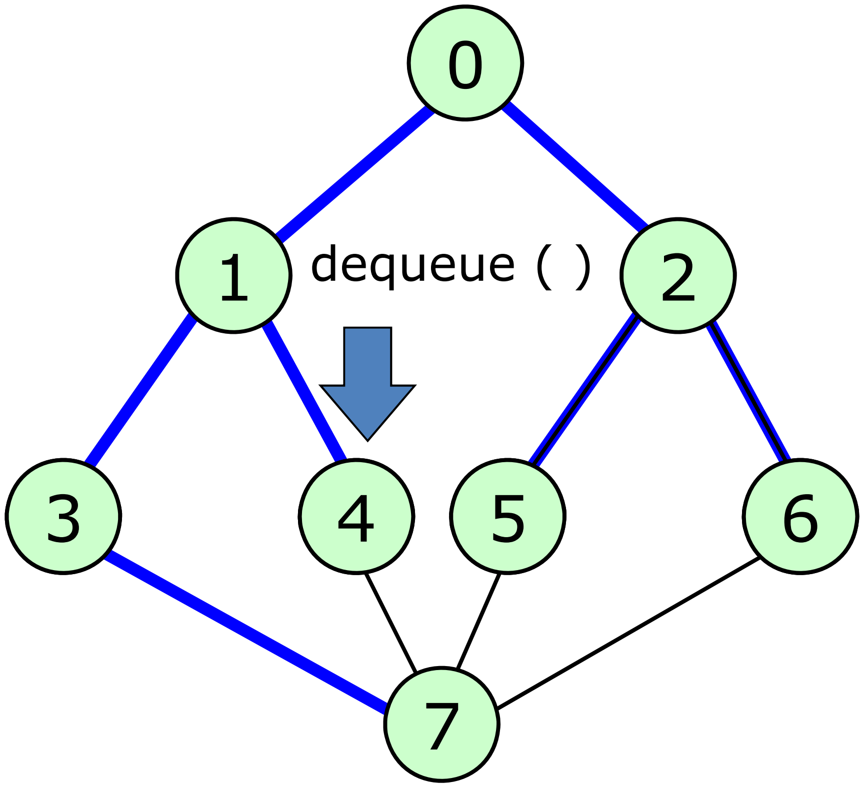
queue

				4	5	6	

9.4.2 Breadth-First Search



9.4.2 Breadth-First Search



v: 4

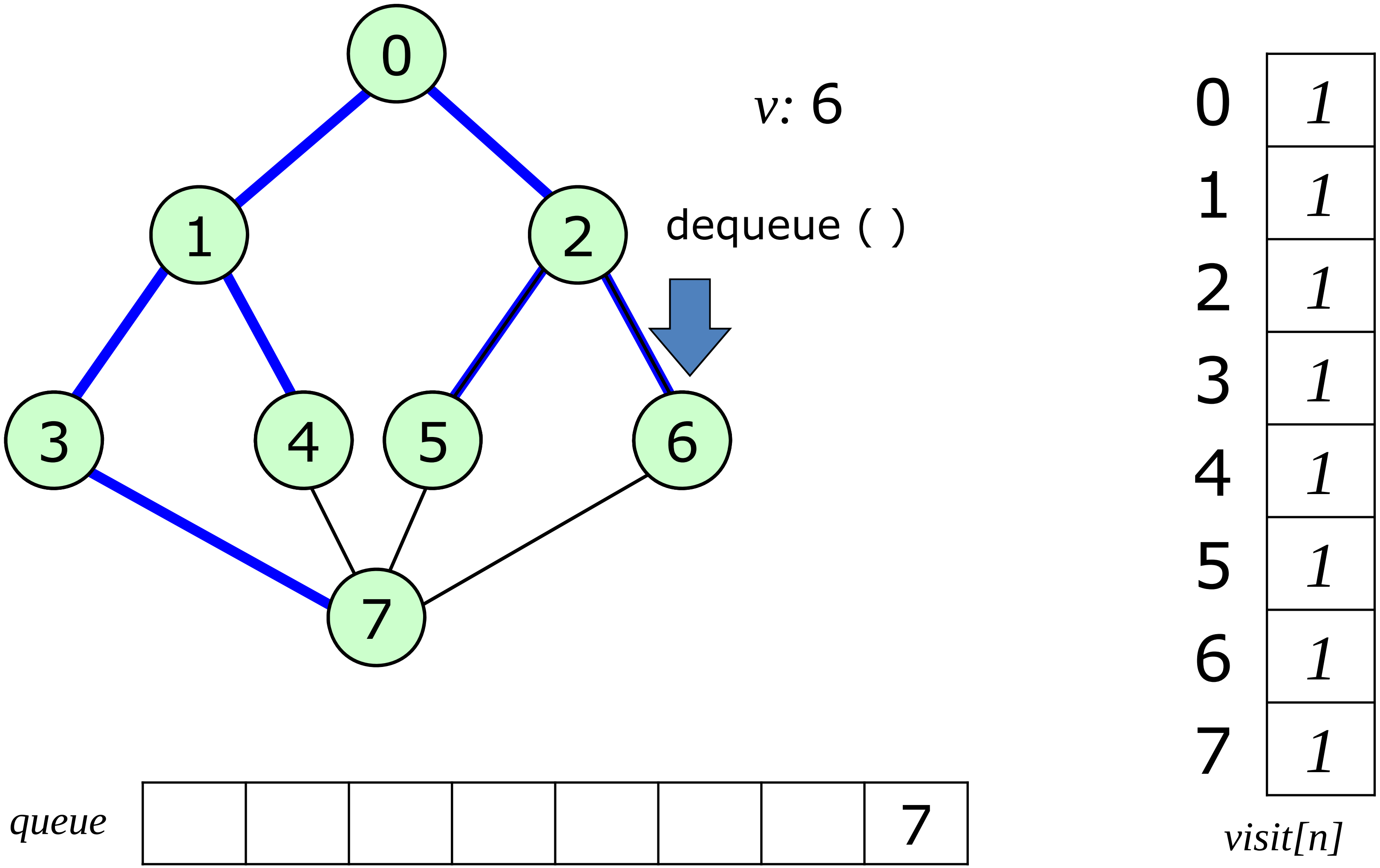
0	*1*
1	*1*
2	*1*
3	*1*
4	*1*
5	*1*
6	*1*
7	*1*

visit[n]

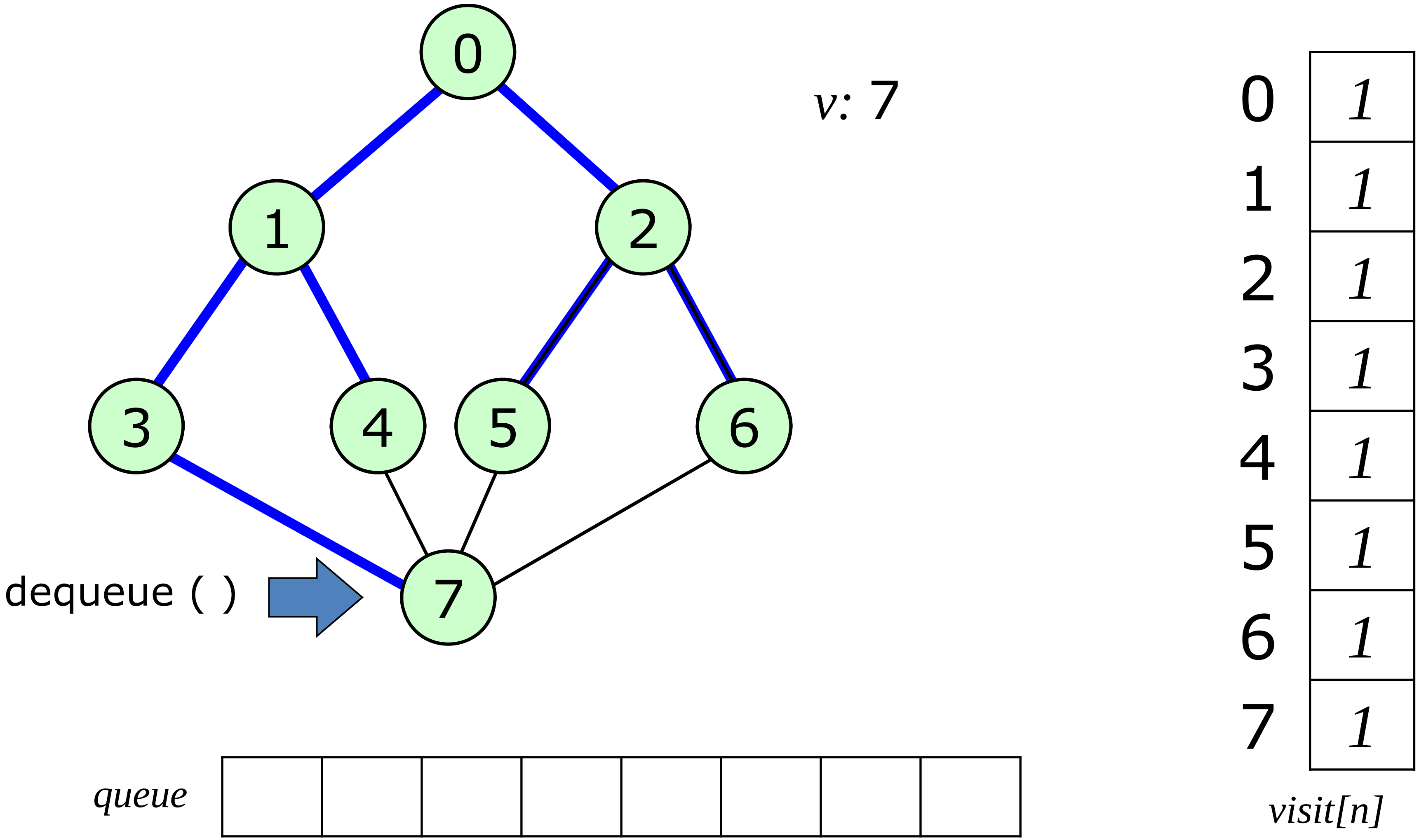
queue

					5	6	7

9.4.2 Breadth-First Search

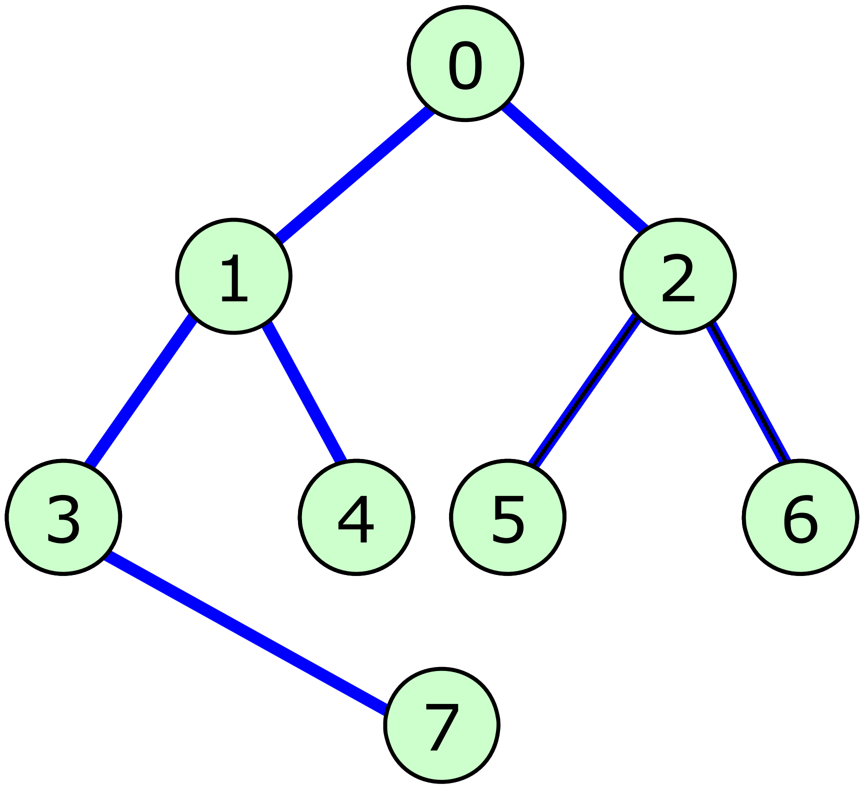


9.4.2 Breadth-First Search



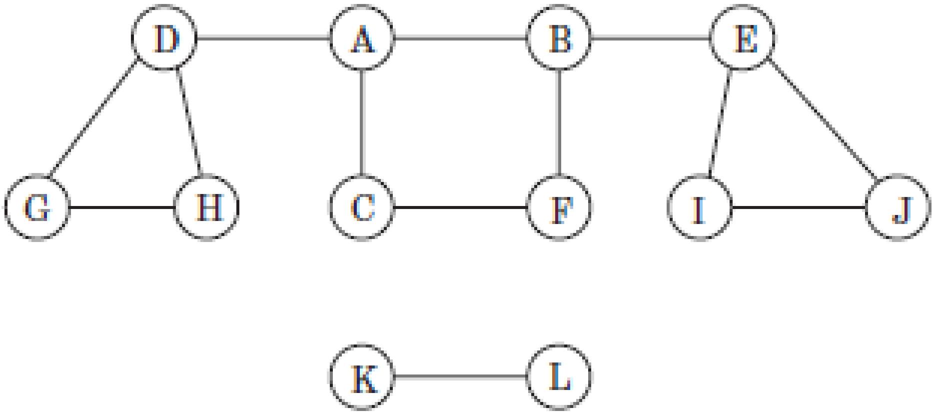
9.4.2 Breadth-First Search

- Breadth-first spanning tree



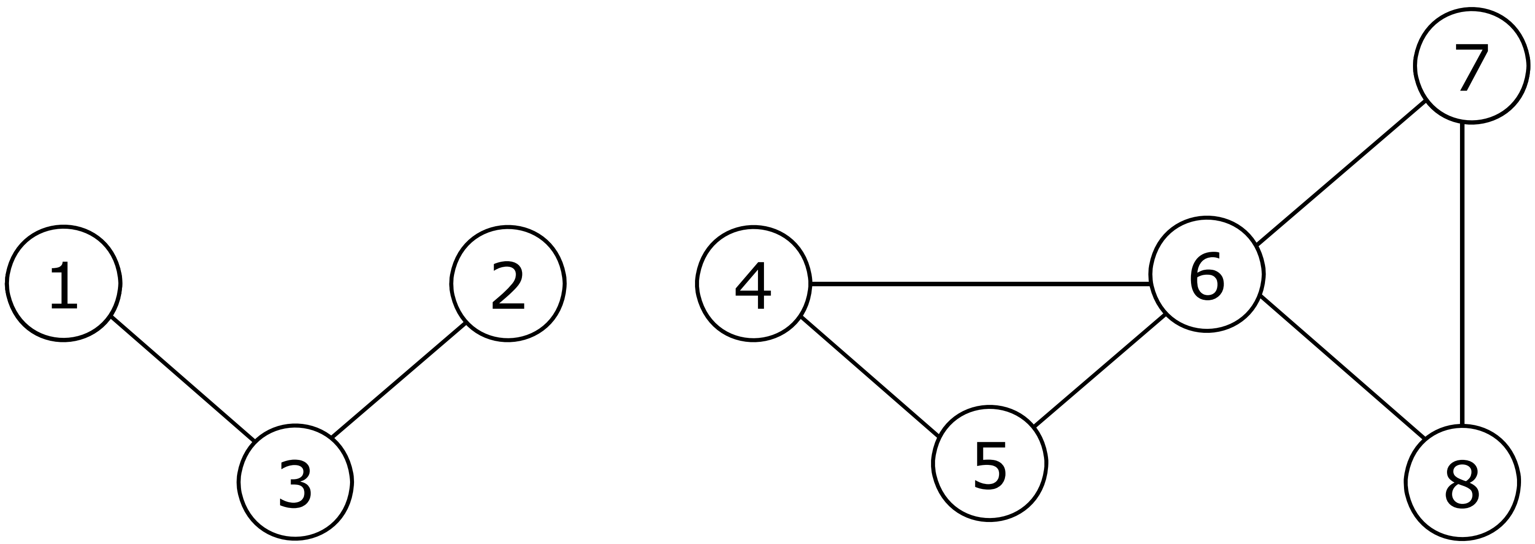
9.4.2 Breadth-First Search

- Exercise



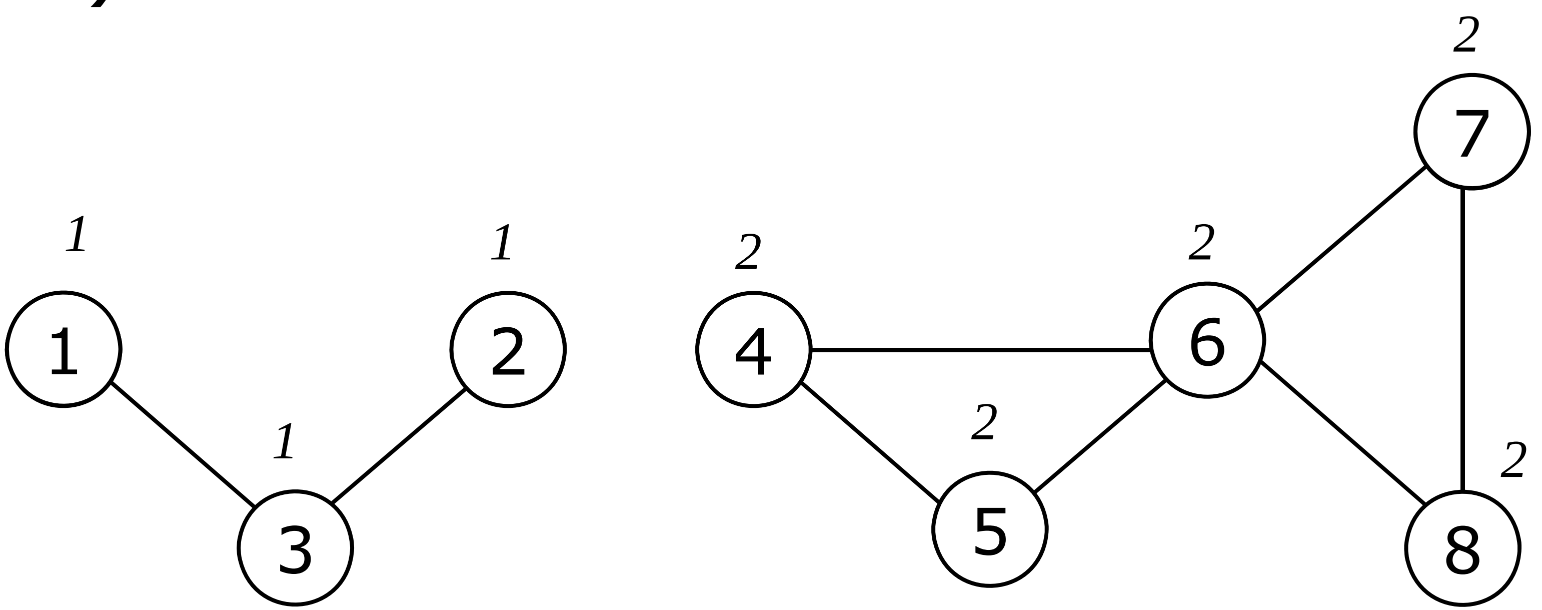
9.4.3 Connected Components

- Mark the connected components of a graph
- Ex)



9.4.3 Connected Components

- Mark the connected components of a graph
- Ex)



9.4.3 Connected Components

- Use a search algorithm to explore all the vertices in a connected component
- In calling dfs ( ) or bfs ( ), add an index for a connected component

9.4.3 Connected Components

```
void connected ( )
{
    int i;
    int idx = 1;
    for ( i = 0; i < n; i++ ) {
        if ( !visit[i] ) {
            dfs ( i, idx );
            idx++;
        }
    }
}
```

9.4.3 Connected Components

```
void dfs ( int v, int idx )
{
    nodePointer w;
    visit[v] = TRUE;
    v->label = idx;

    for ( w = graph[v]; w; w = w->link ) {
        if ( !visit[w->vertex] )
            dfs ( w->visit, idx );
    }
}
```

9.4.4 Spanning Trees

- Spanning tree
  - No. of vertices of G = n
  - No. of edges of the spanning tree of G → (n-1)

  - Use dfs ( ) to create depth-first spanning tree

9.4.4 Spanning Trees

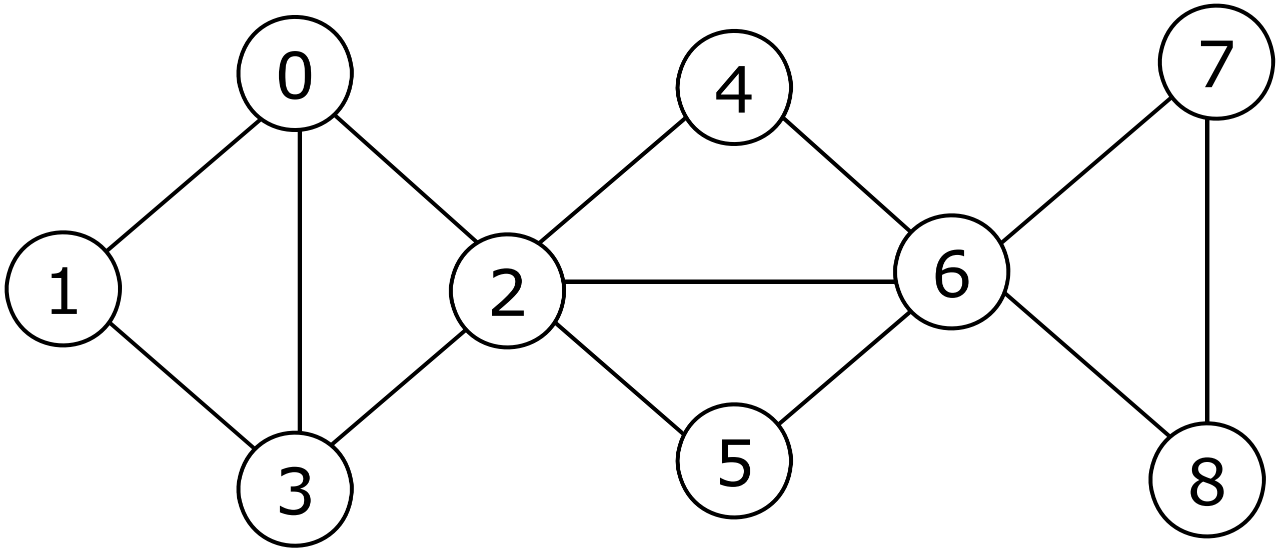
- Depth-first spanning tree

```
void dfst ( int v )
{
    nodePointer w;
    visit[v] = TRUE;

    for ( w = graph[v]; w; w = w->link ) {
        if ( !visit[w->vertex] ) {
            add (v, w) to the depth-first spanning tree;
            dfst ( w->visit );
        }
    }
}
```

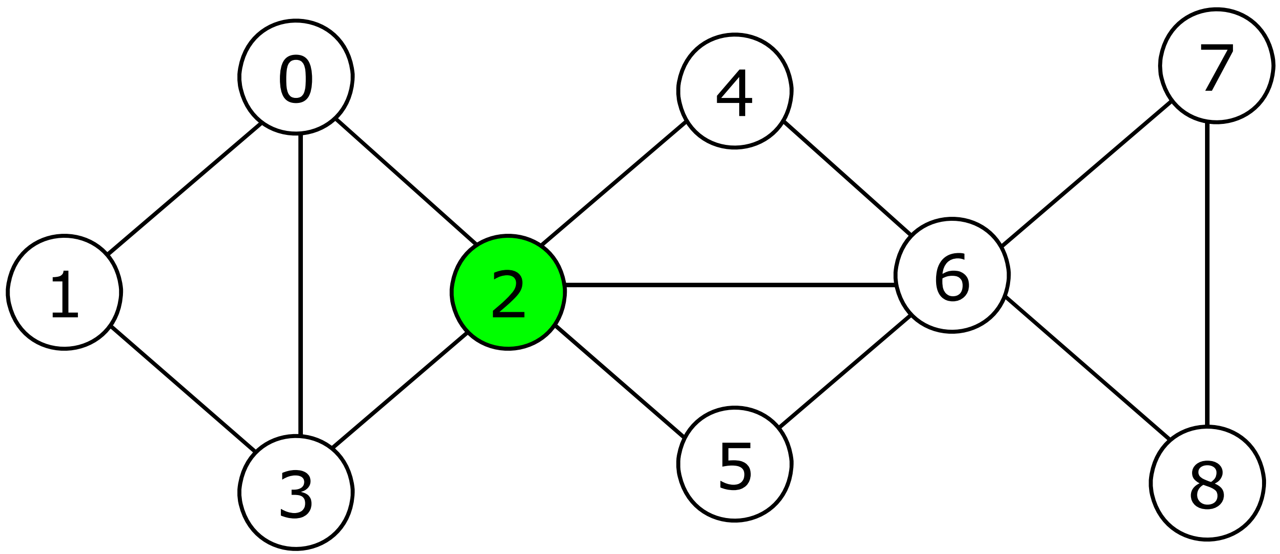
9.5 Biconnected Components

- Articulation point
  - A vertex v of G such that the deletion of v, together with all edges incident to v, produces a graph G′ that has at least two connected components



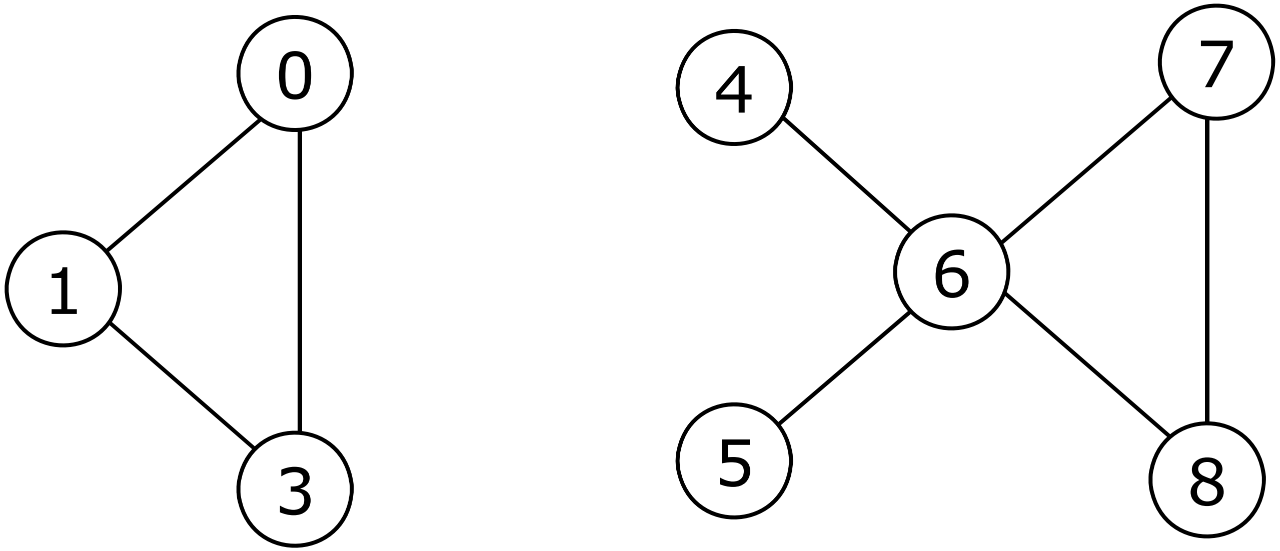
9.5 Biconnected Components

- Articulation point
  - A vertex v of G such that the deletion of v, together with all edges incident to v, produces a graph G' that has at least two connected components



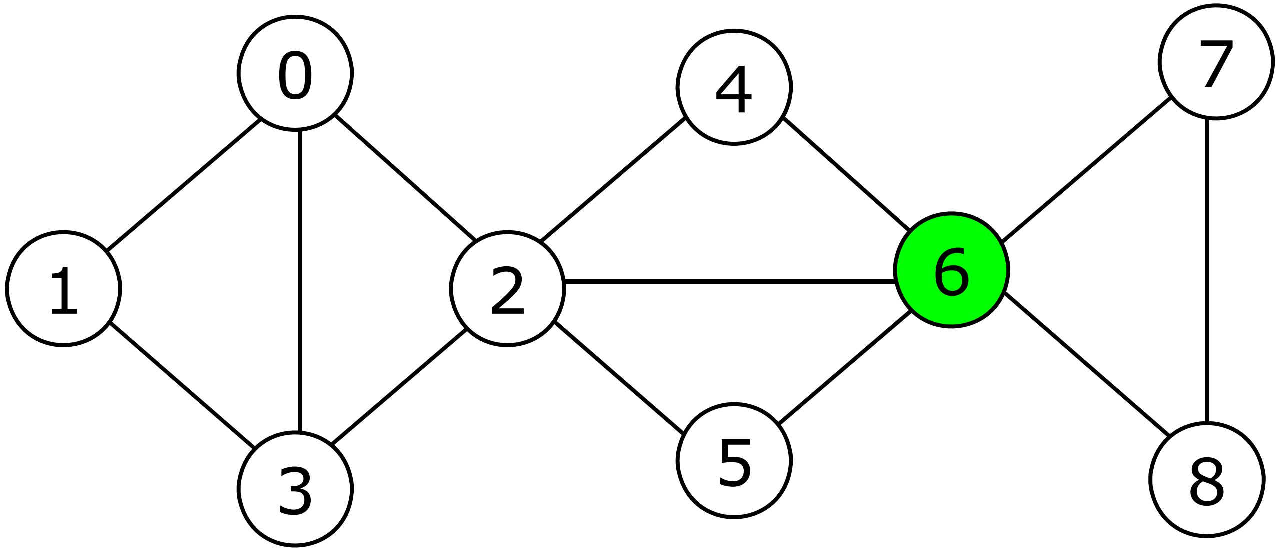
9.5 Biconnected Components

- Articulation point
  - A vertex v of G such that the deletion of v, together with all edges incident to v, produces a graph G' that has at least two connected components



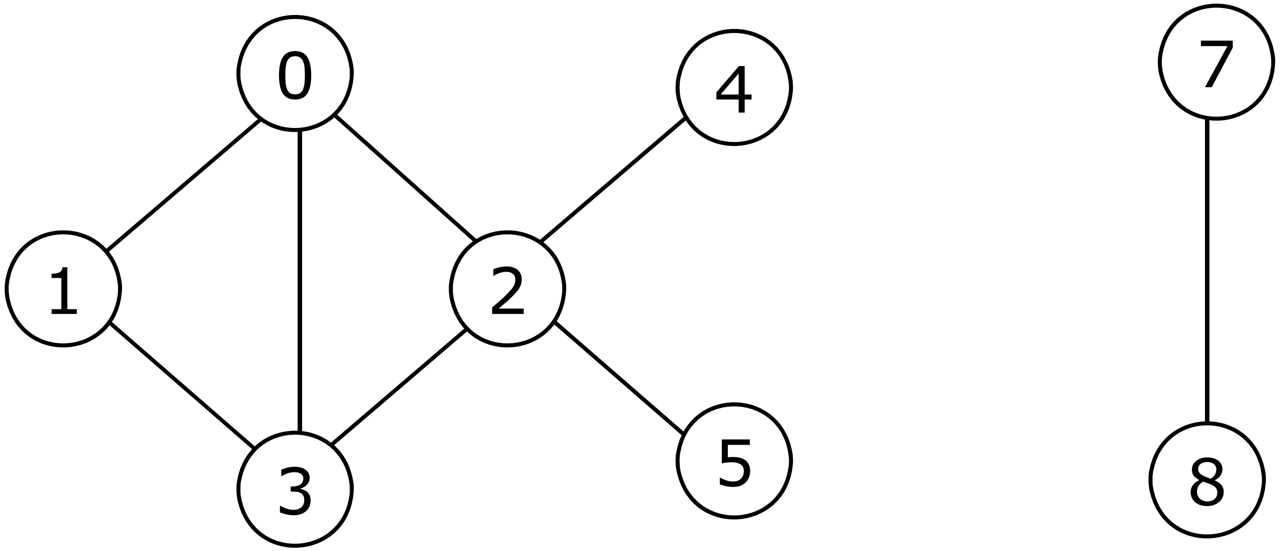
9.5 Biconnected Components

- Articulation point
  - A vertex v of G such that the deletion of v, together with all edges incident to v, produces a graph G′ that has at least two connected components



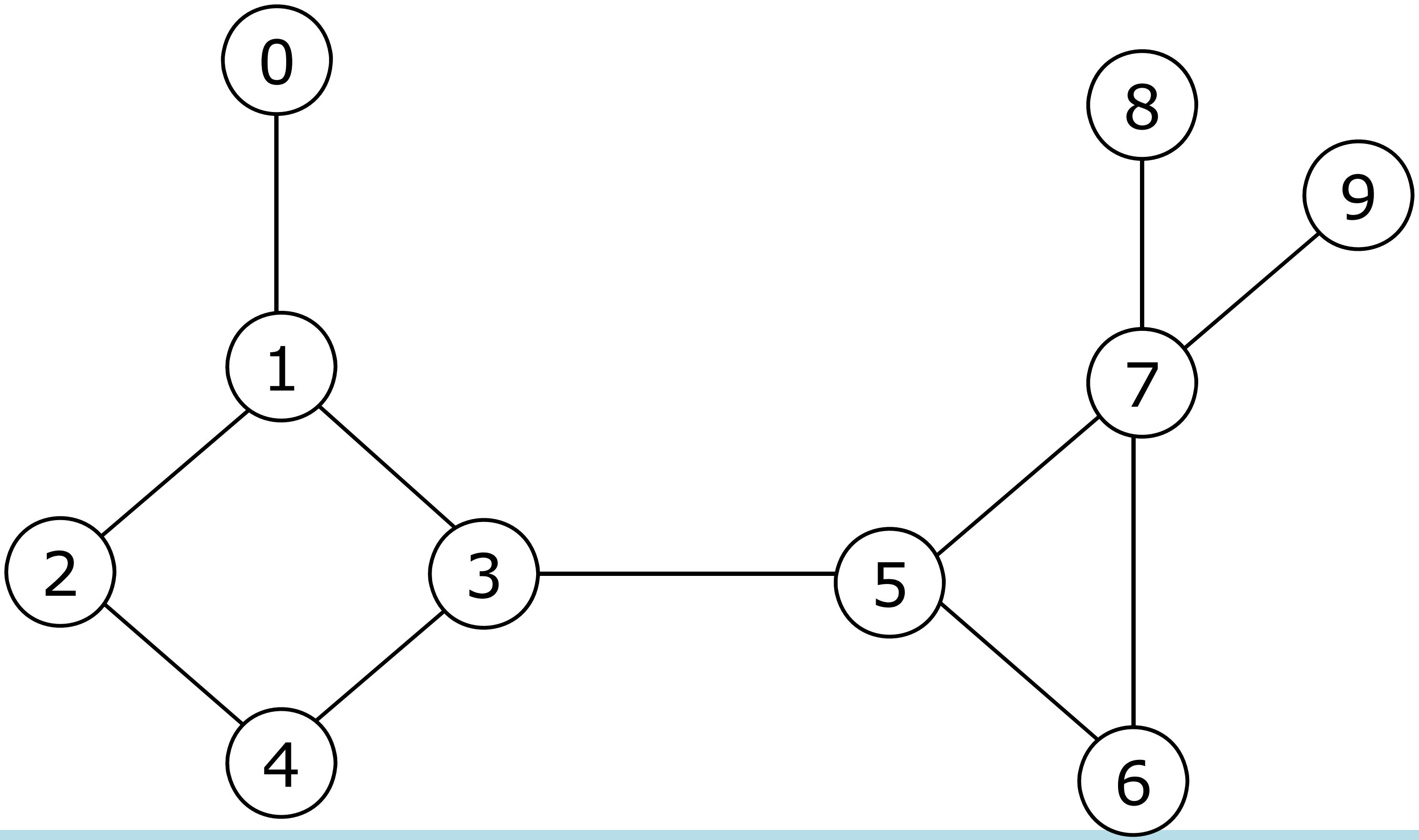
9.5 Biconnected Components

- Articulation point
  - A vertex v of G such that the deletion of v, together with all edges incident to v, produces a graph G′ that has at least two connected components



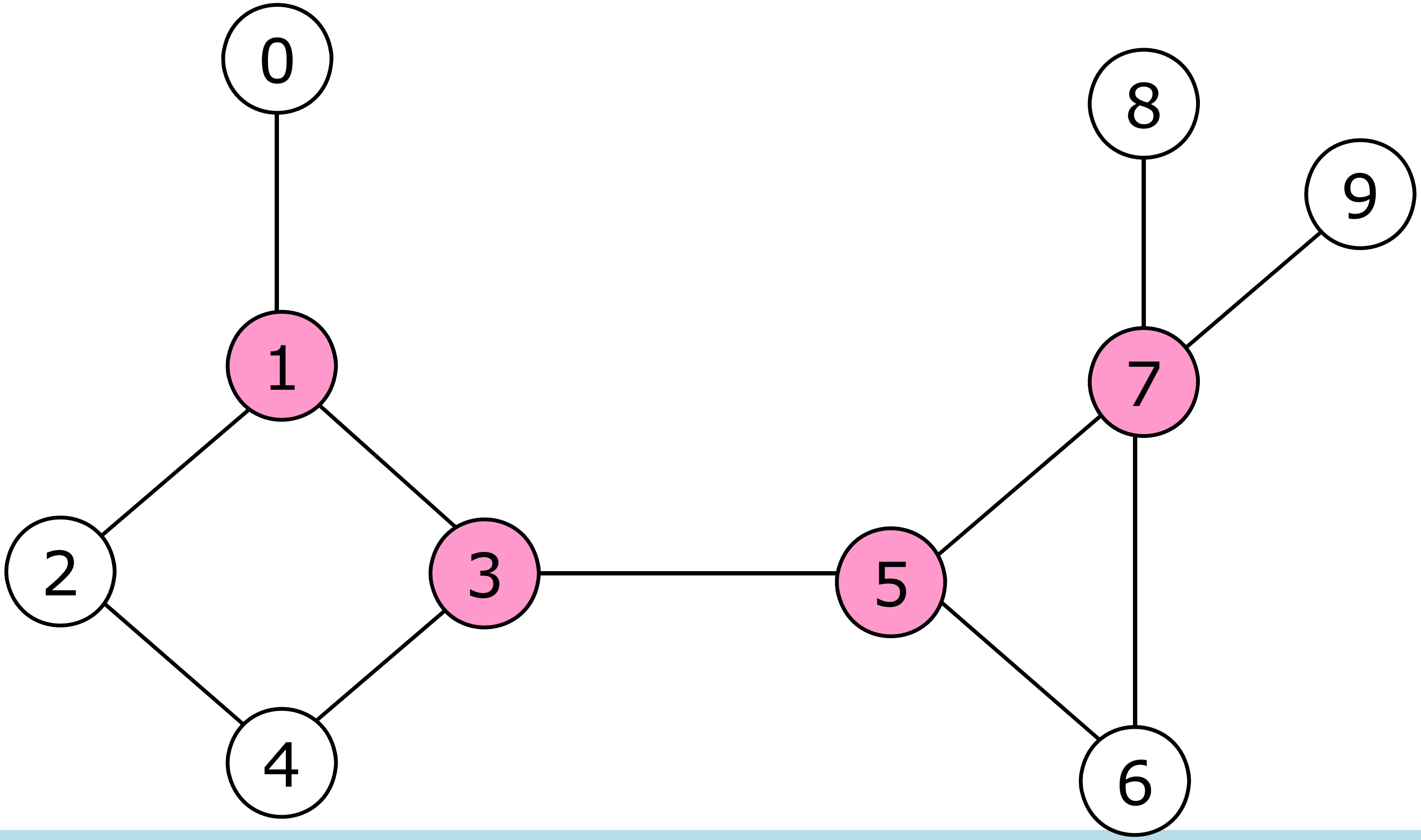
9.5 Biconnected Components

- Ex) What is the articulation point on this graph?



9.5 Biconnected Components

- Ex) What is the articulation point on this graph?

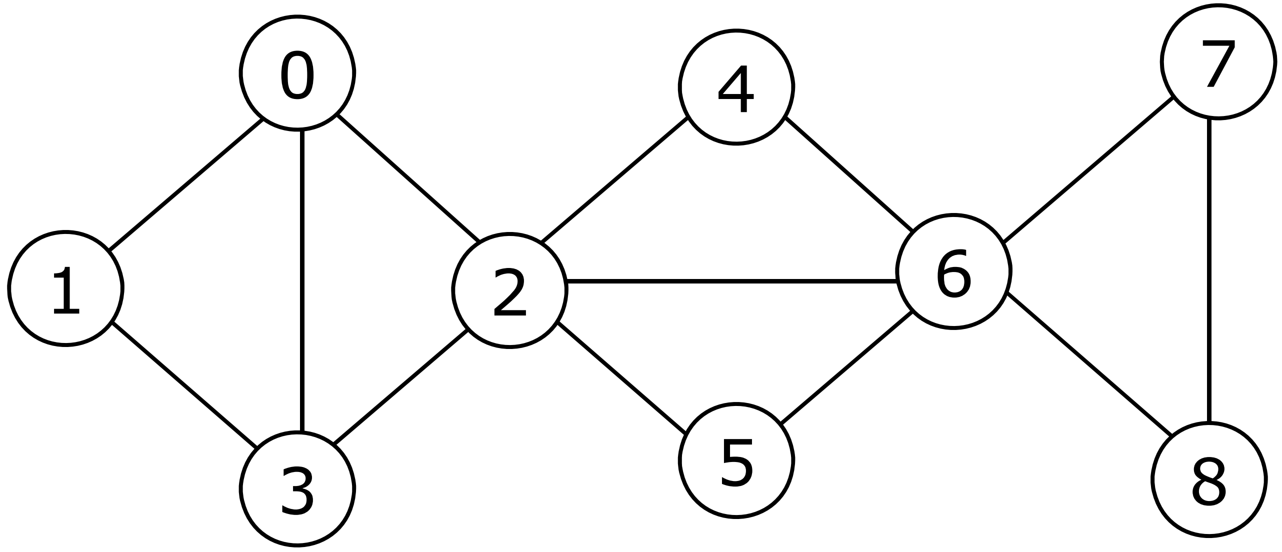


9.5 Biconnected Components

- Biconnected graph
  - A connected graph that has no articulation points

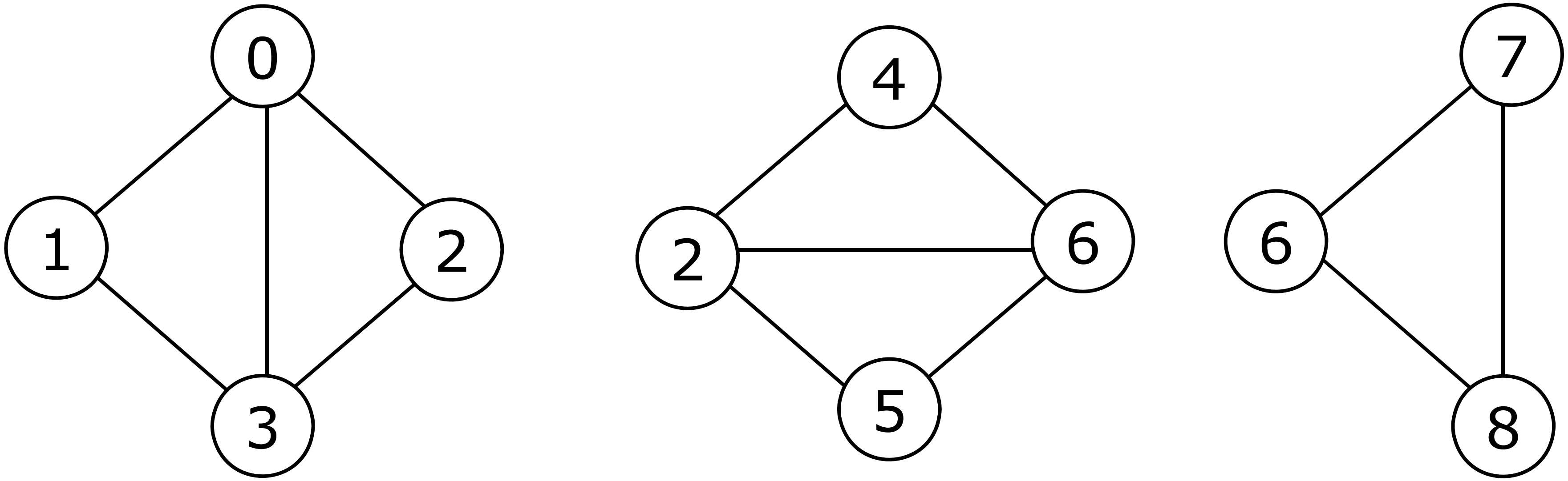
9.5 Biconnected Components

- Biconnected component
  - A maximal biconnected subgraph



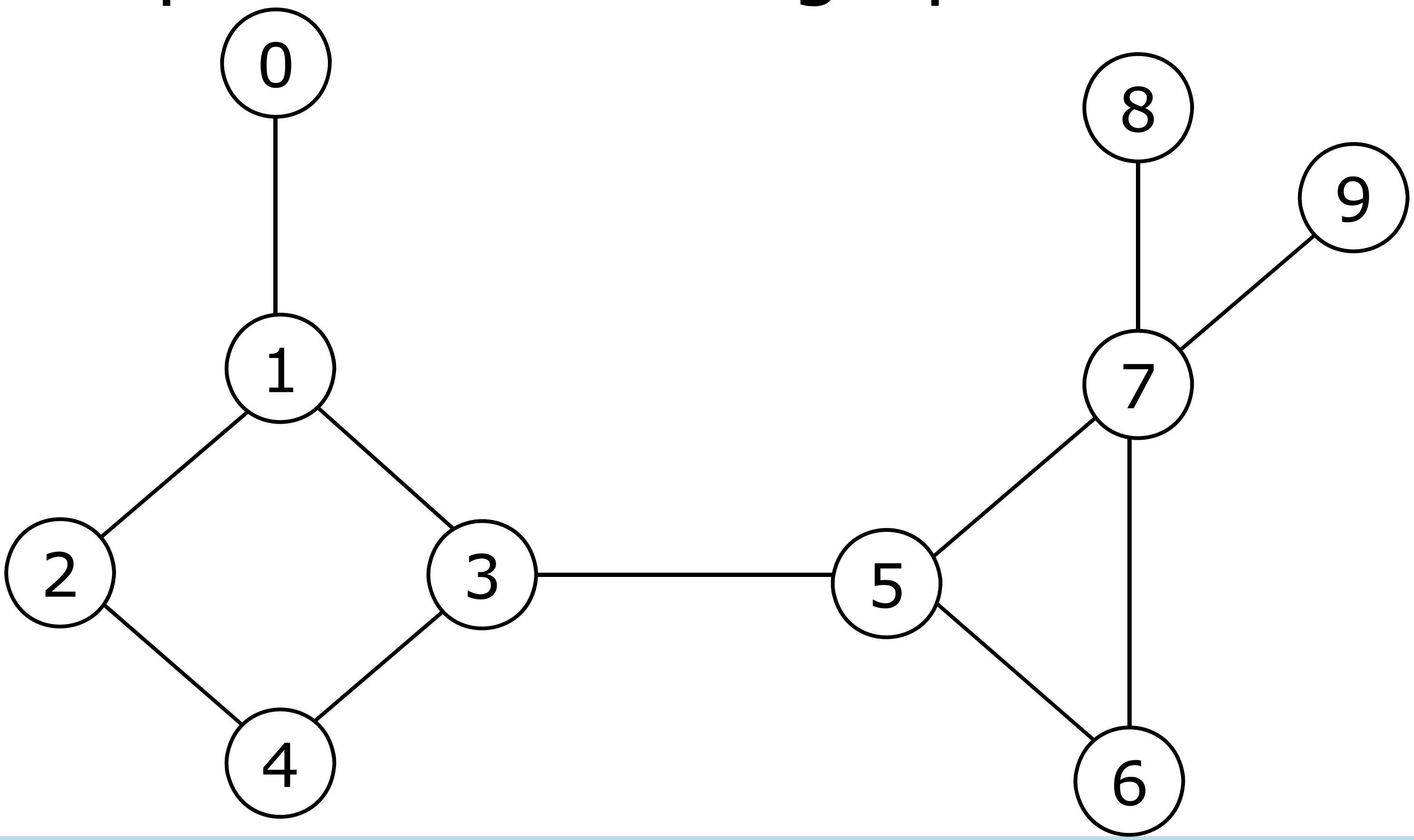
9.5 Biconnected Components

- Biconnected component
  - A maximal biconnected subgraph



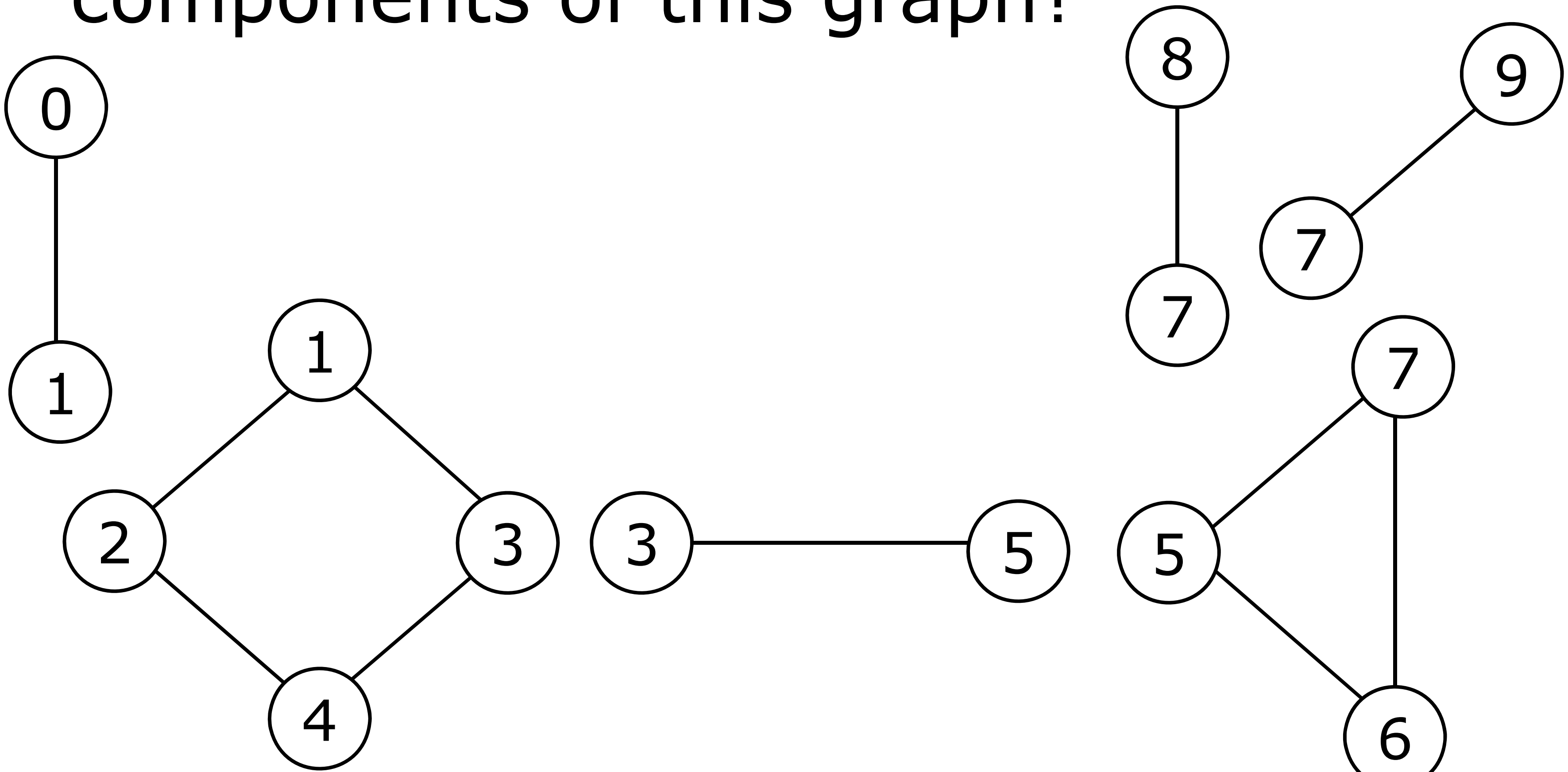
9.5 Biconnected Components

- Ex) What are the biconnected components of this graph?



9.5 Biconnected Components

- Ex) What are the biconnected components of this graph?



9.5 Biconnected Components

- Steps to compute biconnected components

(1) Computing articulation points

① computing dfn

② finding back edge

③ definition of articulation point

④ computing articulation points

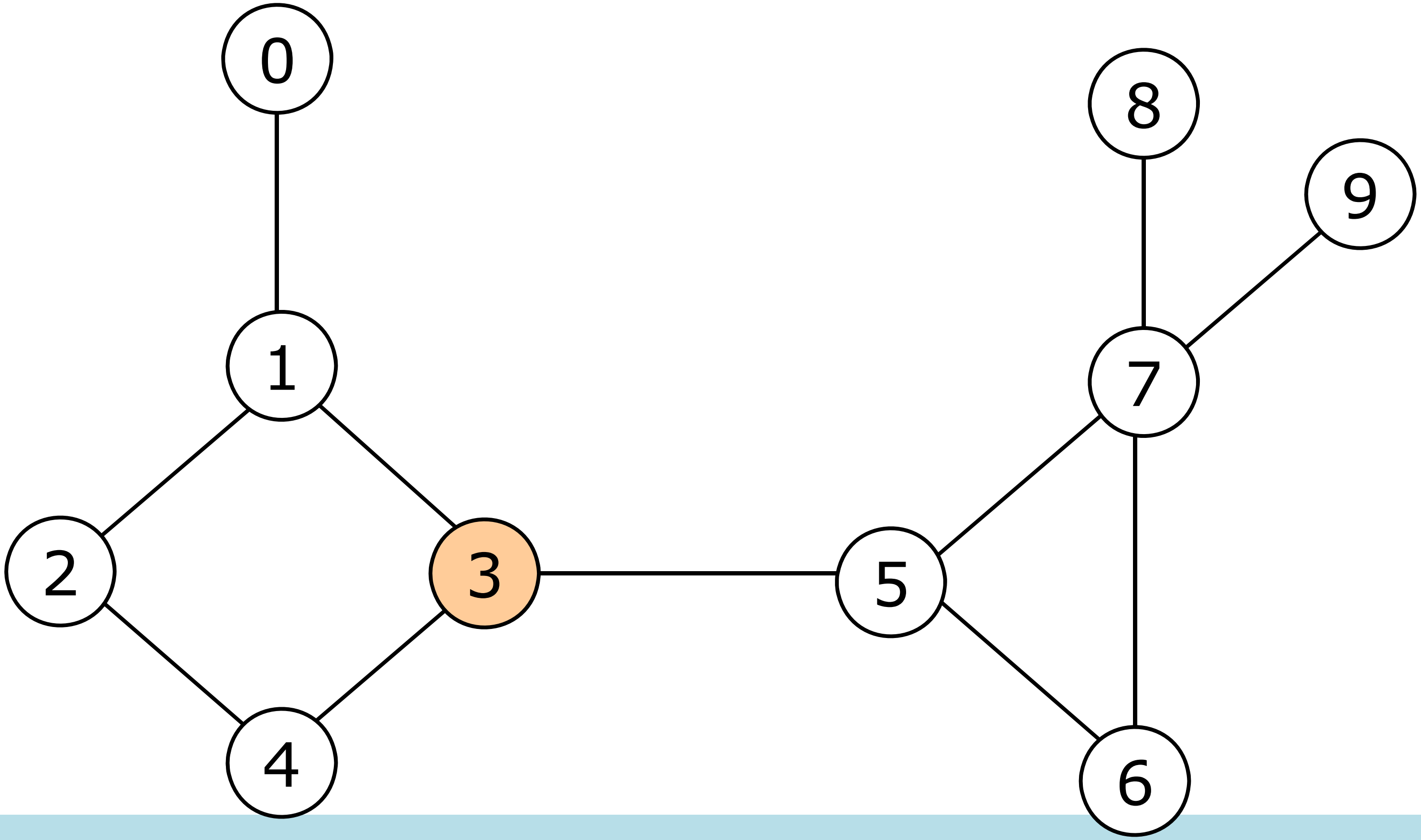
(2) Decomposition of a graph into biconnected components using articulation points

① computing dfn

- dfn: depth-first number
- computed by depth-first search
- dfn of a vertex
  - The sequence in which the vertices are visited during depth-first search

9.5 Biconnected Components

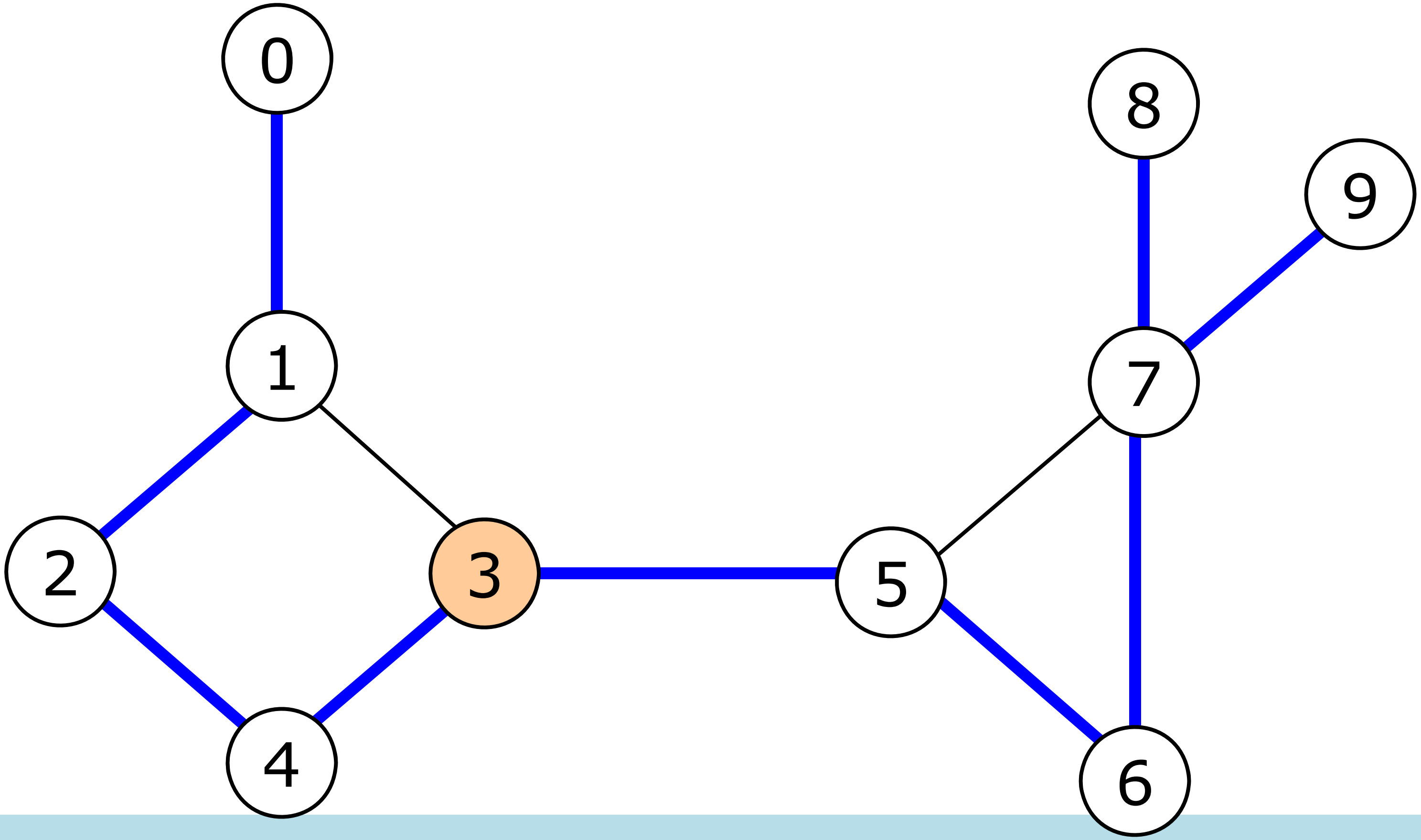
- Ex) What is dfn, if 3 is a root of depth-first spanning tree?



① computing dfn

9.5 Biconnected Components

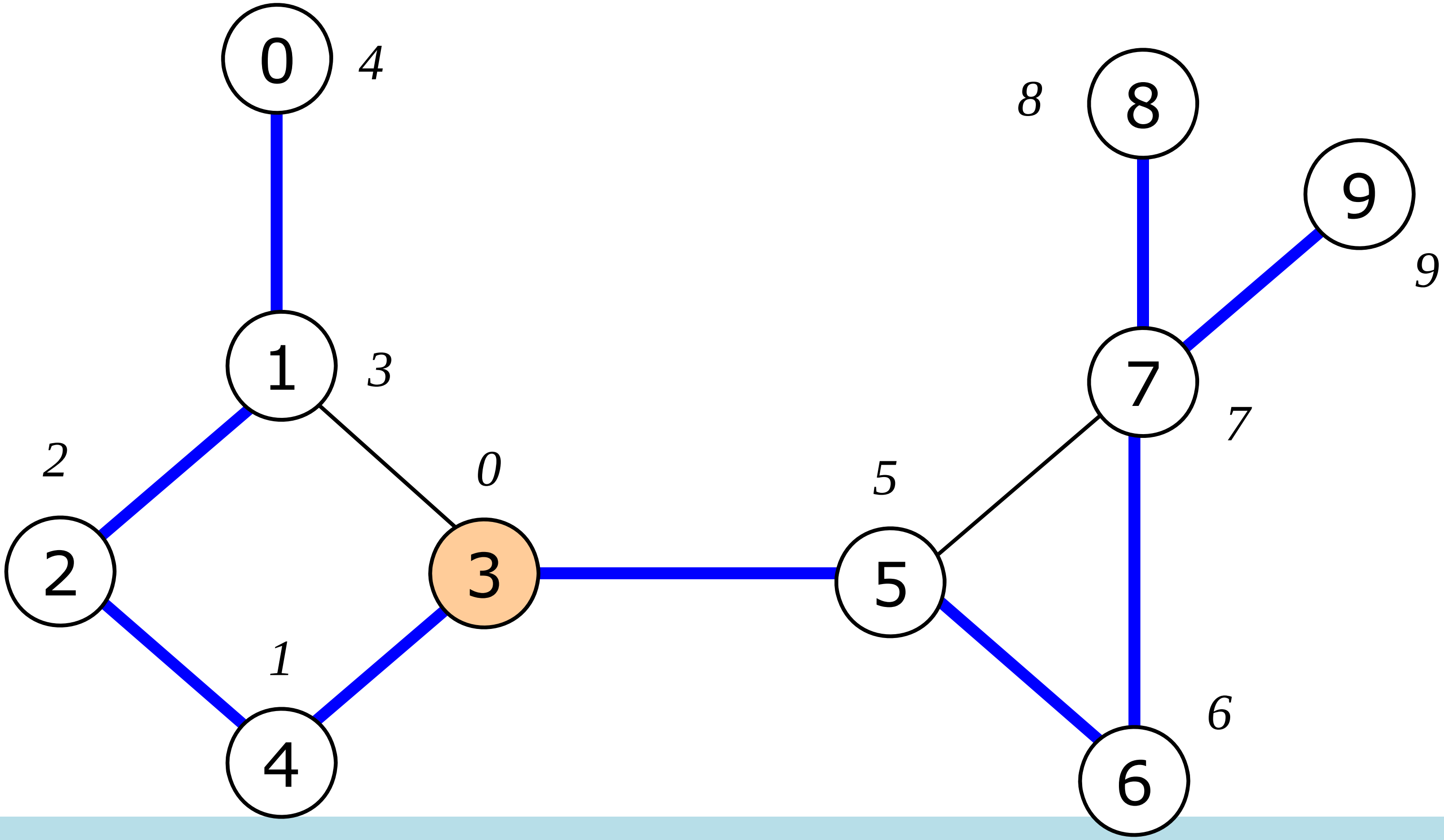
- Ex) What is dfn, if 3 is a root of depth-first spanning tree?



① computing dfn

9.5 Biconnected Components

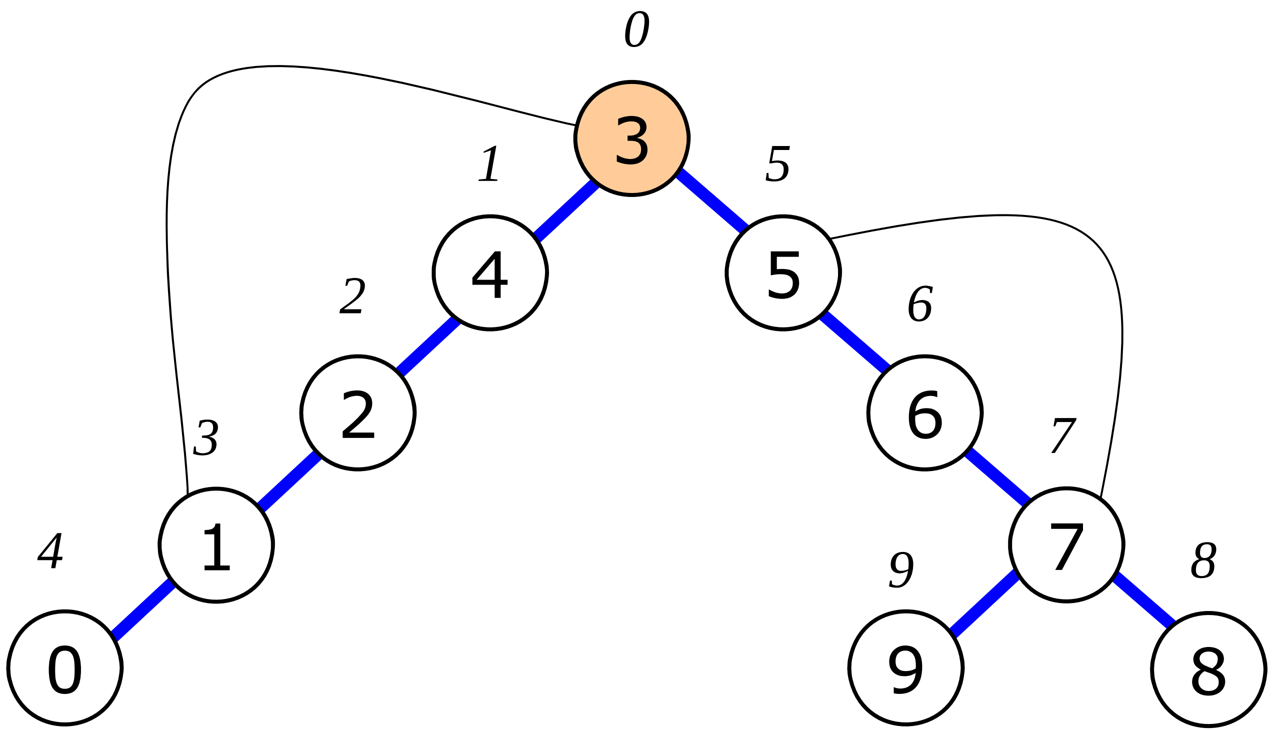
- Ex) What is dfn, if 3 is a root of depth-first spanning tree?



① computing dfn

9.5 Biconnected Components

- Ex) What is dfn, if 3 is a root of depth-first spanning tree?



① computing dfn

9.5 Biconnected Components

- Property of dfn
  - u is an ancestor of v in the depth-first spanning tree → dfn(u) < dfn(v)
  - Ex) Vertex 4 is the ancestor of vertex 1

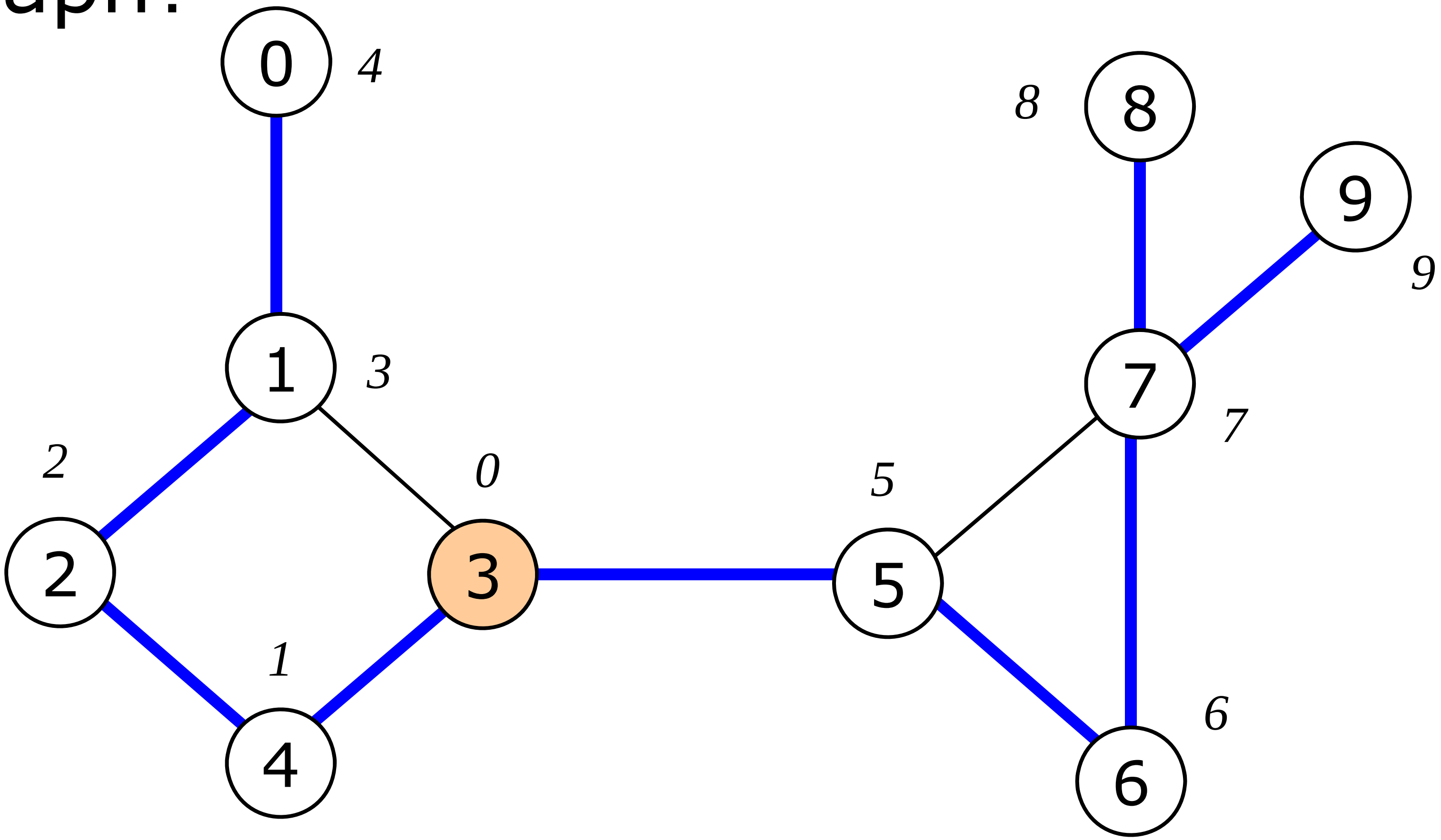
    → dfn(4) = 1 < dfn(1) = 3

② Back edge

- Edges in G = edges in the spanning tree + nontree edges
- Back edge:
  - A nontree edge (u, v), if u is an ancestor of v or vice versa
- In depth-first spanning tree, all the nontree edges are back edges

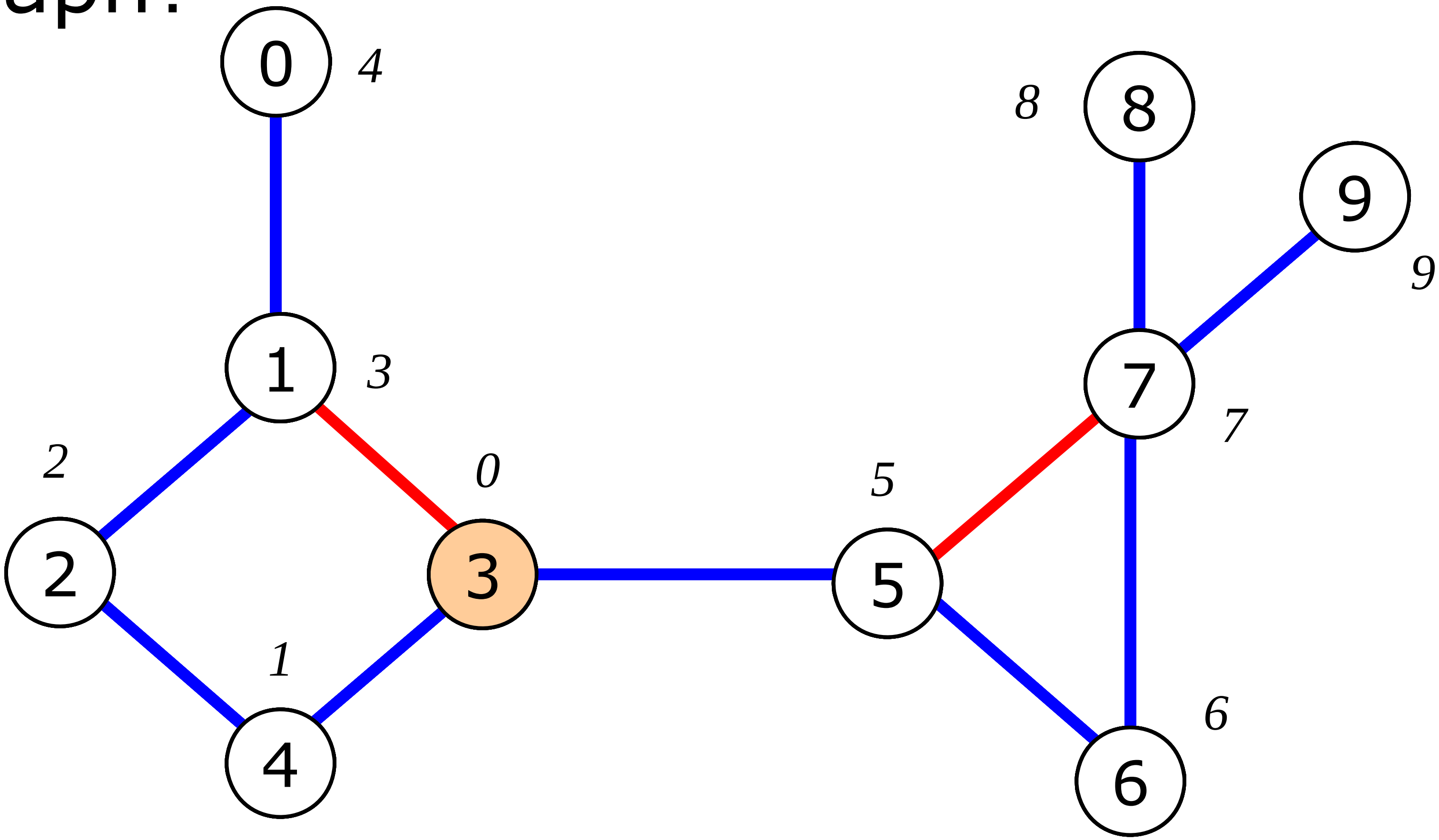
9.5 Biconnected Components

- Ex) What are the back edges of this graph?



② back edge

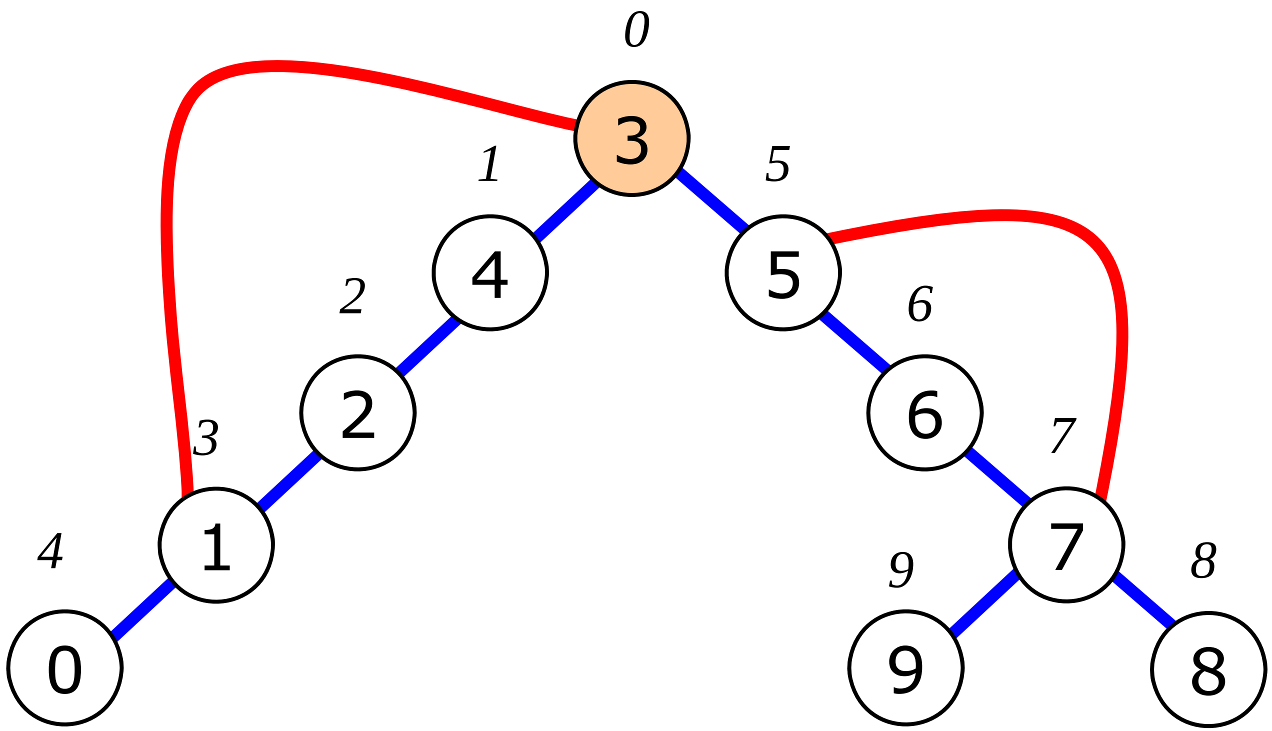
- Ex) What are the back edges of this graph?



② back edge

9.5 Biconnected Components

- Ex) What are the back edges of this graph?



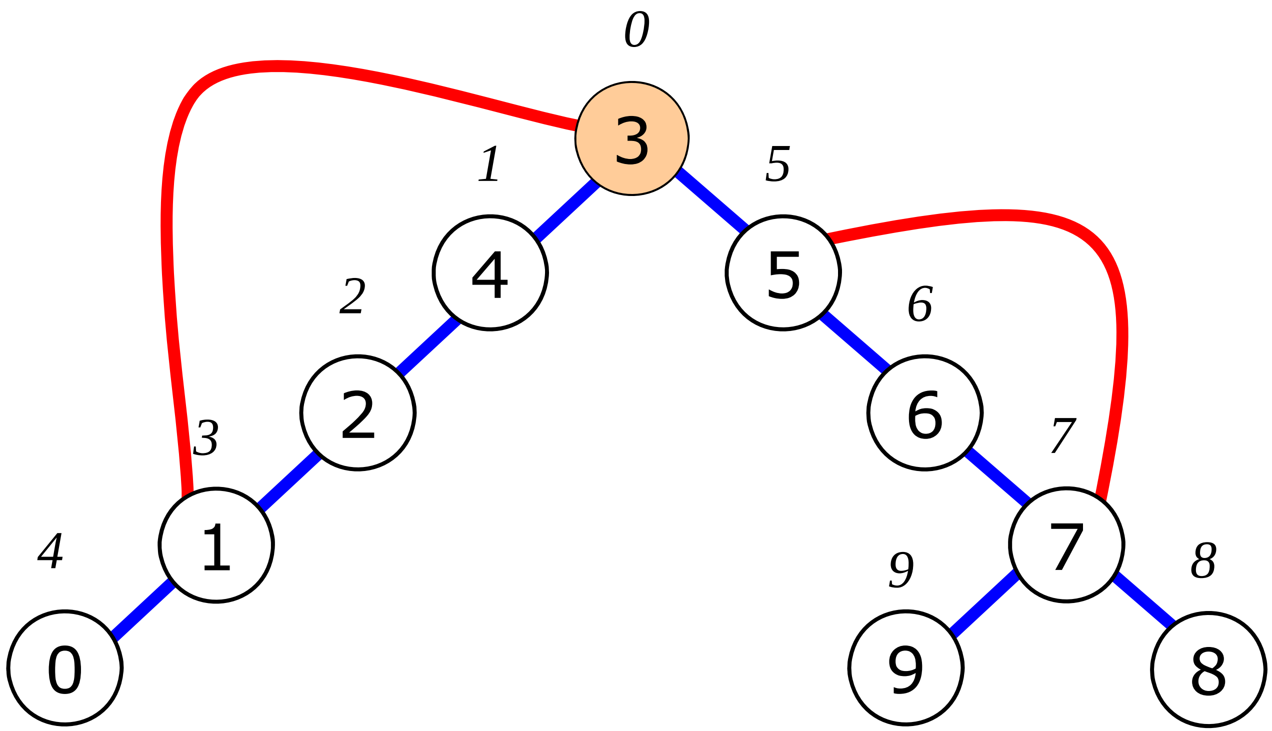
② back edge

③ Definition of articulation point (1)

– A root of a depth-first spanning tree is an articulation point, if it has at least two childs

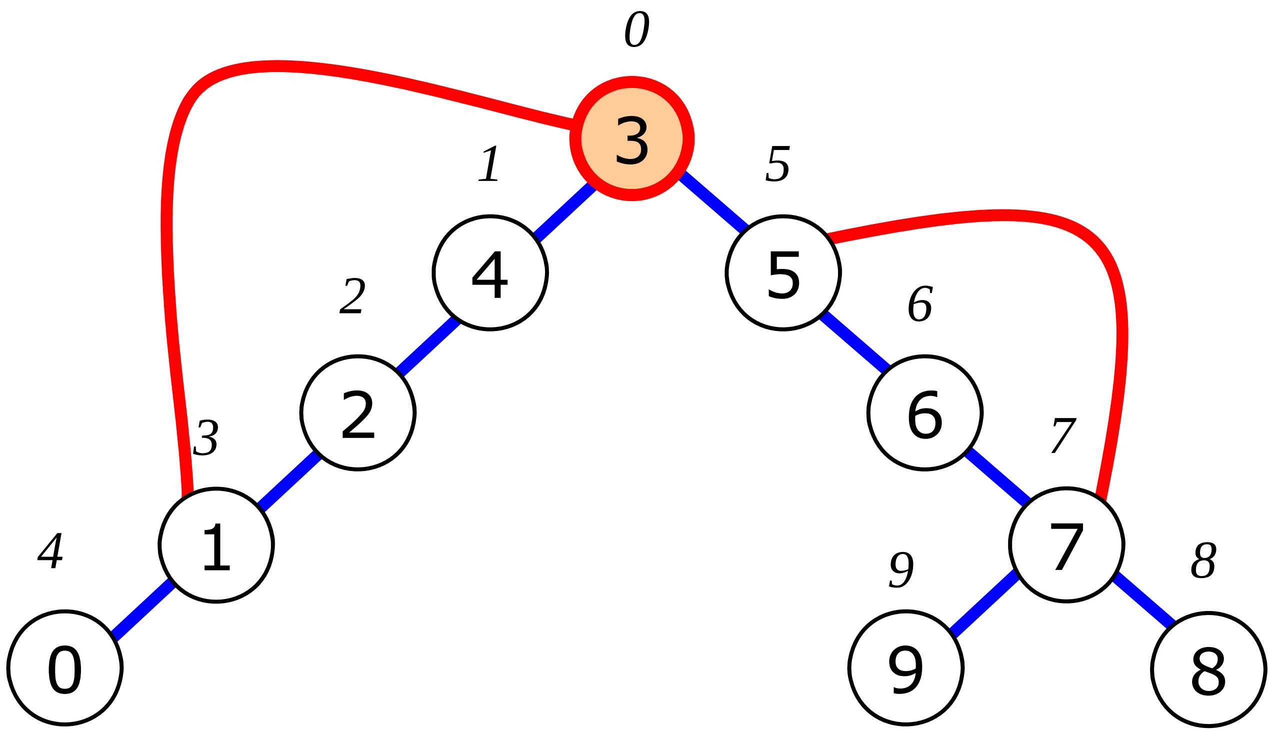
9.5 Biconnected Components

- Articulation points
  - Ex) What are the articulation points?



9.5 Biconnected Components

- Articulation points
  - Ex)



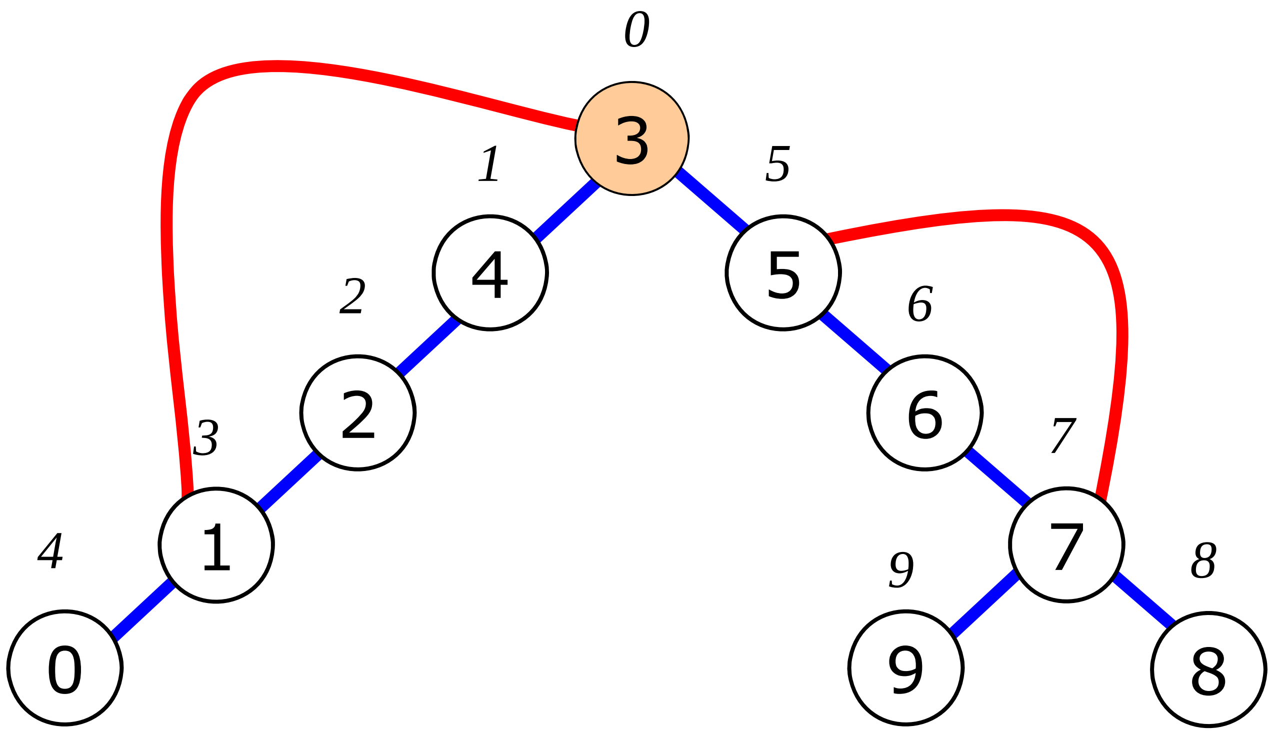
③ Definition of articulation point

③ Definition of articulation point (2)

- A vertex u, if it has at least one child w such that a path (w, descendants of w, and a single back edge, ancestor of u) does not exist

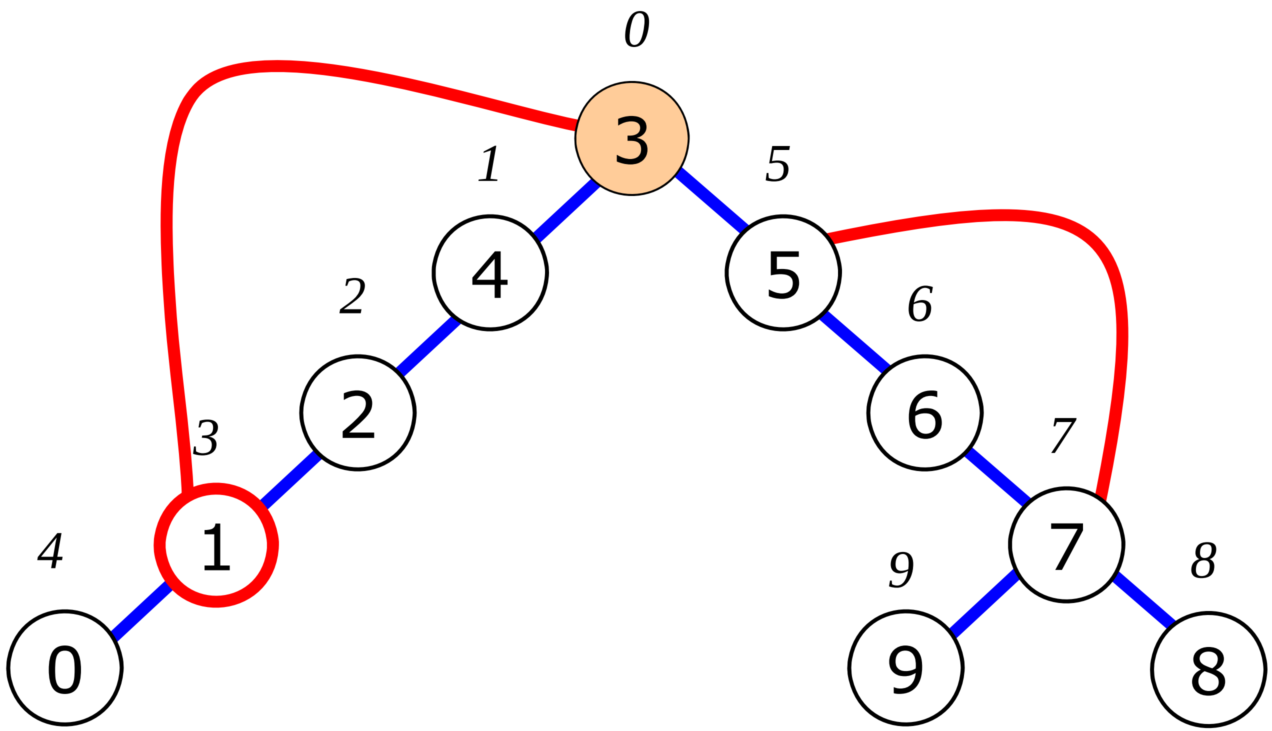
9.5 Biconnected Components

- Ex) Is vertex 4 an articulation point?



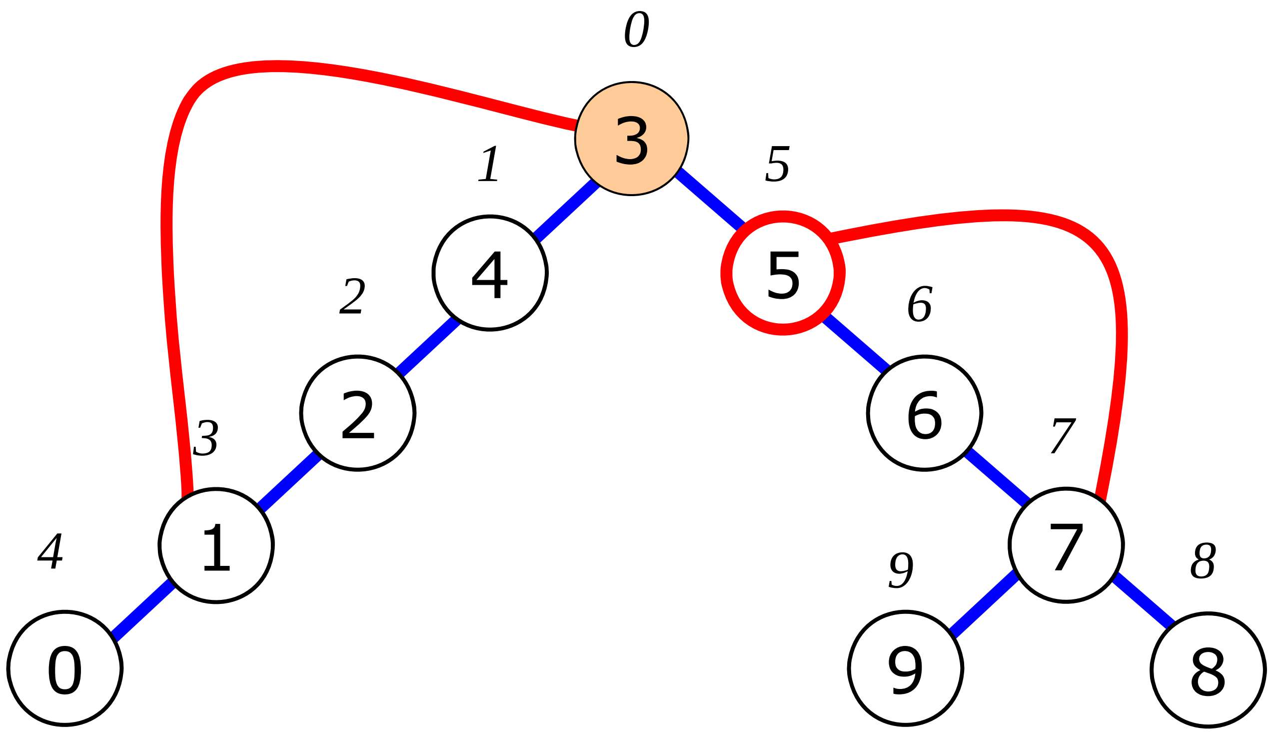
9.5 Biconnected Components

- Ex) Is vertex 1 an articulation point?



9.5 Biconnected Components

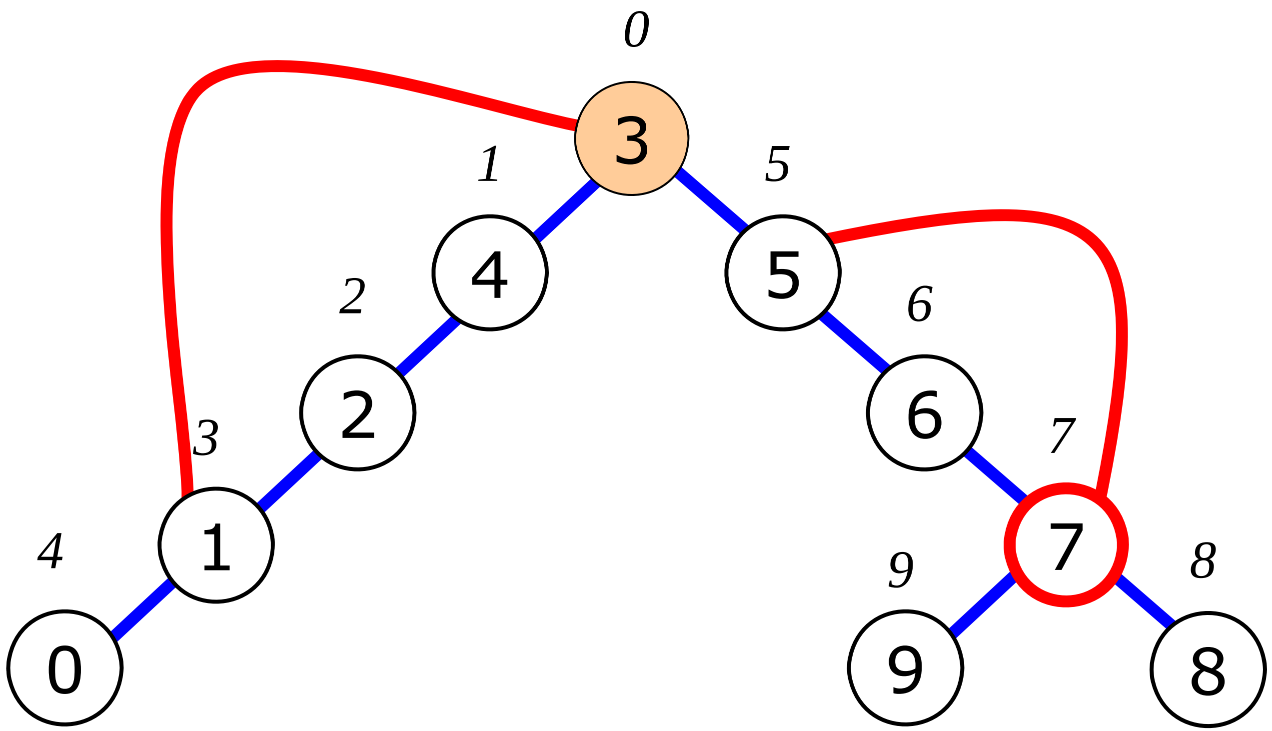
- Ex) Is vertex 5 an articulation point?



③ Definition of articulation point

9.5 Biconnected Components

- Ex) Is vertex 7 an articulation point?



9.5 Biconnected Components

④ computing articulation points

– Define a new value low for each vertex u, such as low(u)

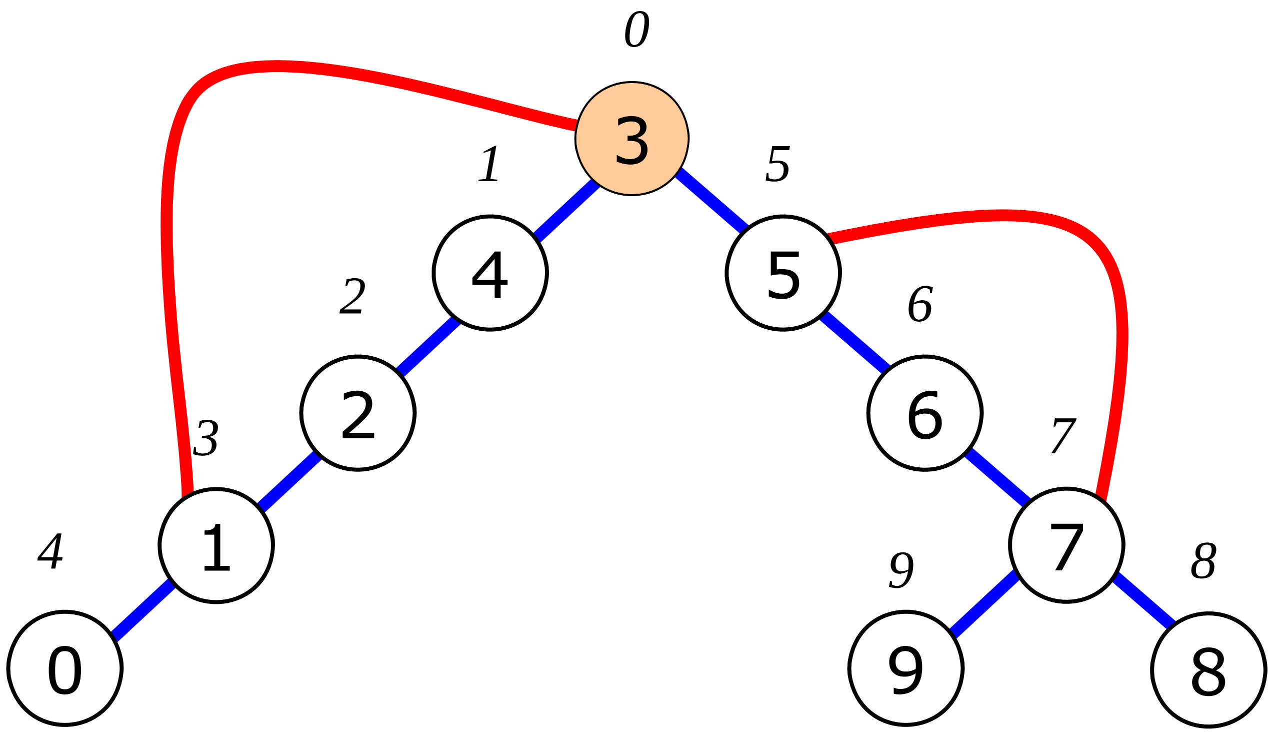
– low(u)

- The lowest depth first number that we can reach from u using a path of descendants followed by at most one back edge

$$
\begin{aligned}
low(u) = \min\{ & dfn(u), \\
& \min\{low(w) \mid w \text{ is a child of } u\}, \\
& \min\{dfn(v) \mid (u,v) \text{ is a back edge}\}\}
\end{aligned}
$$

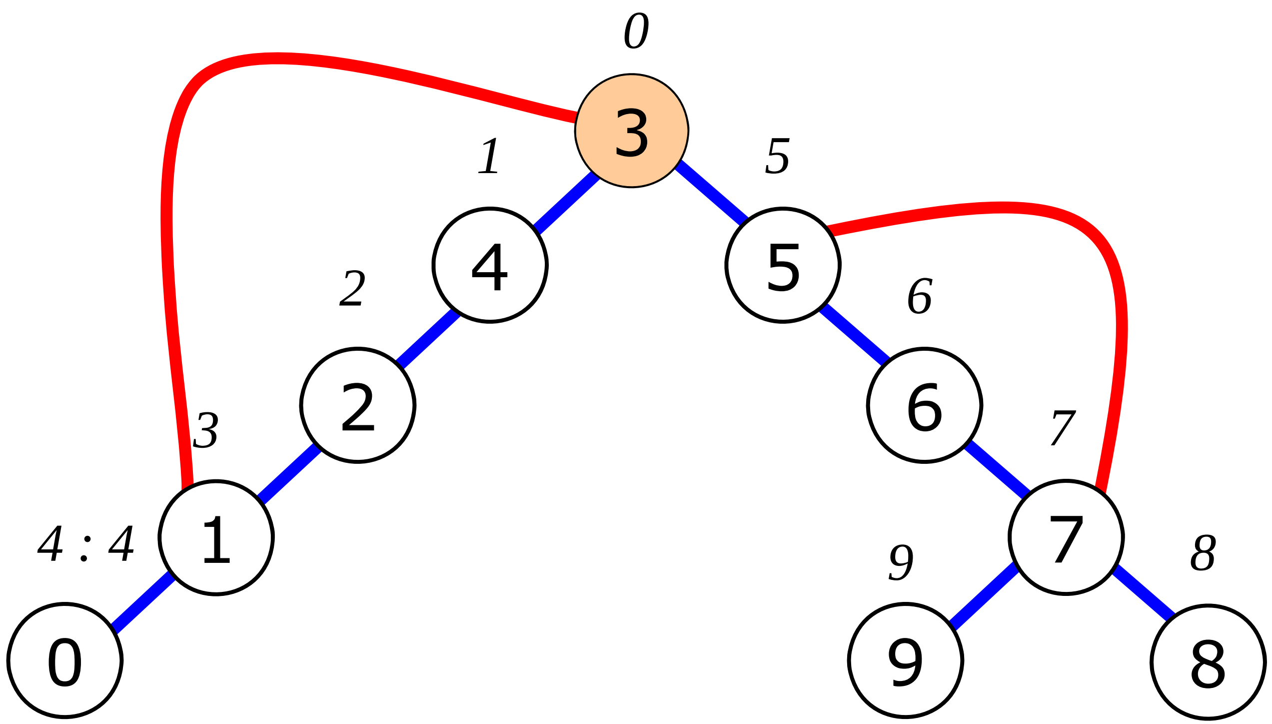
9.5 Biconnected Components

- low(u)
  - Ex) What are low(u)?



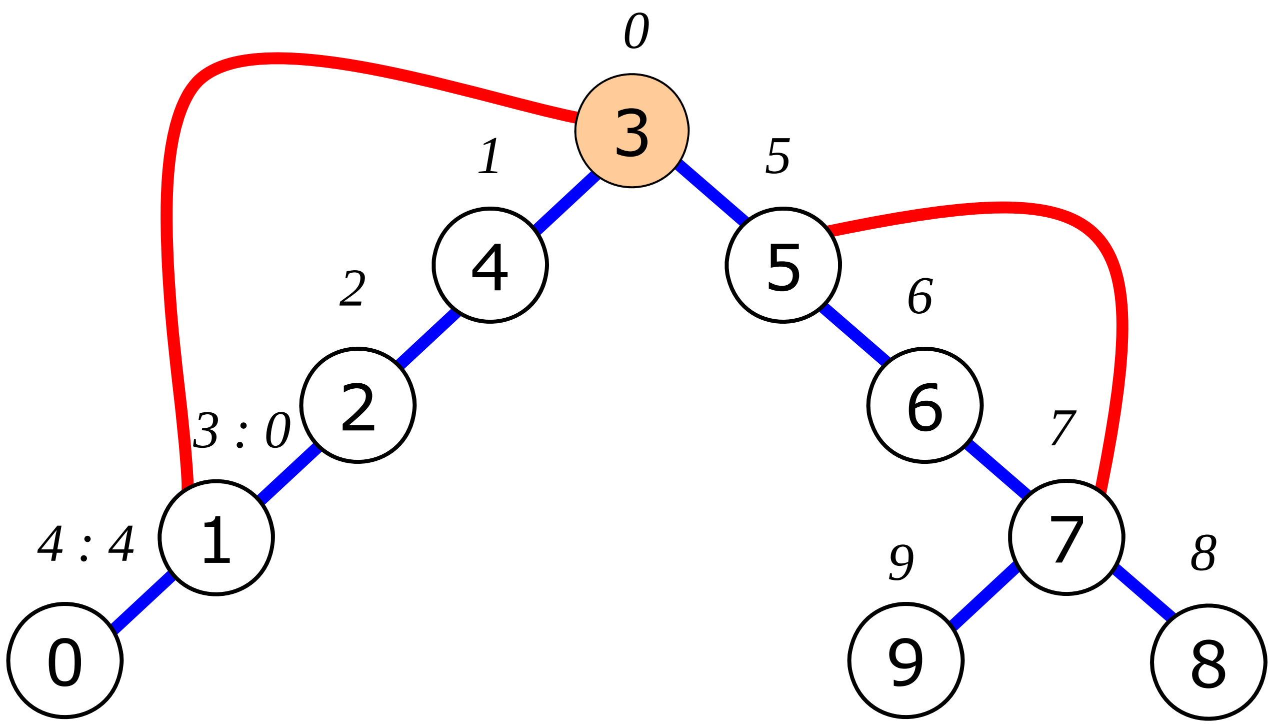
9.5 Biconnected Components

- low(u)
  - Ex) What are low(0)?



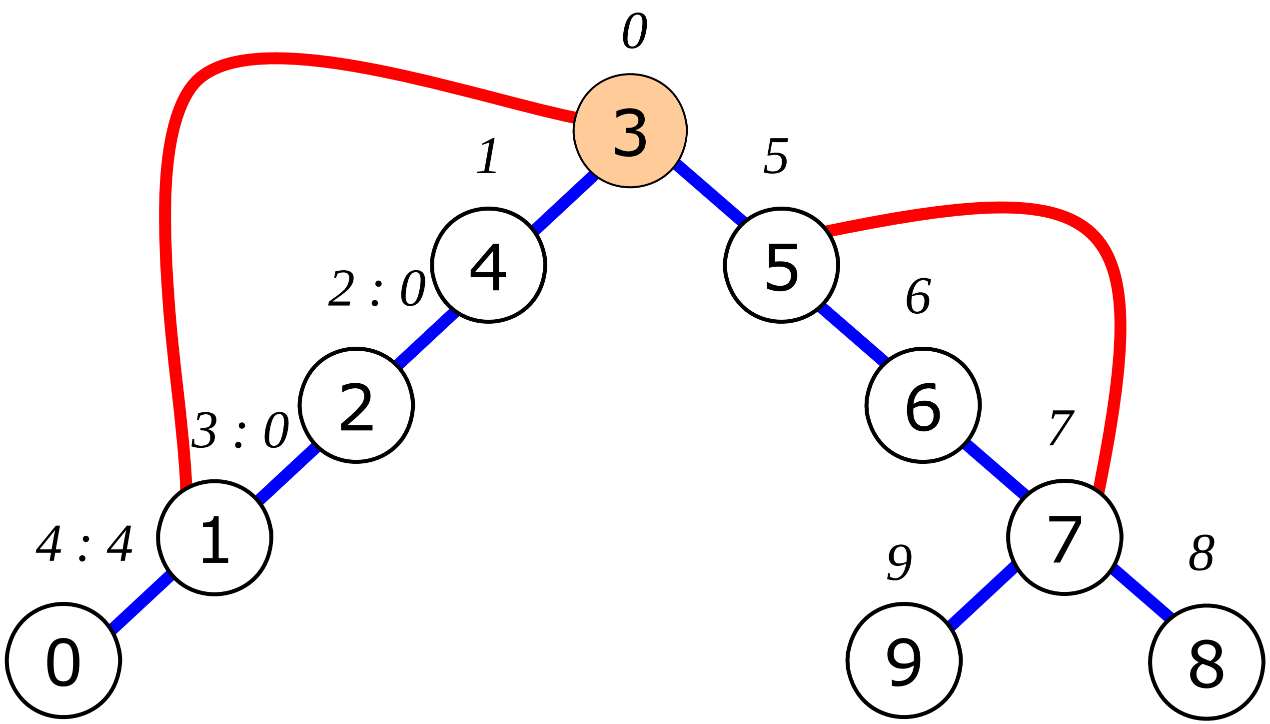
9.5 Biconnected Components

- low(u)
  - Ex) What are low(1)?



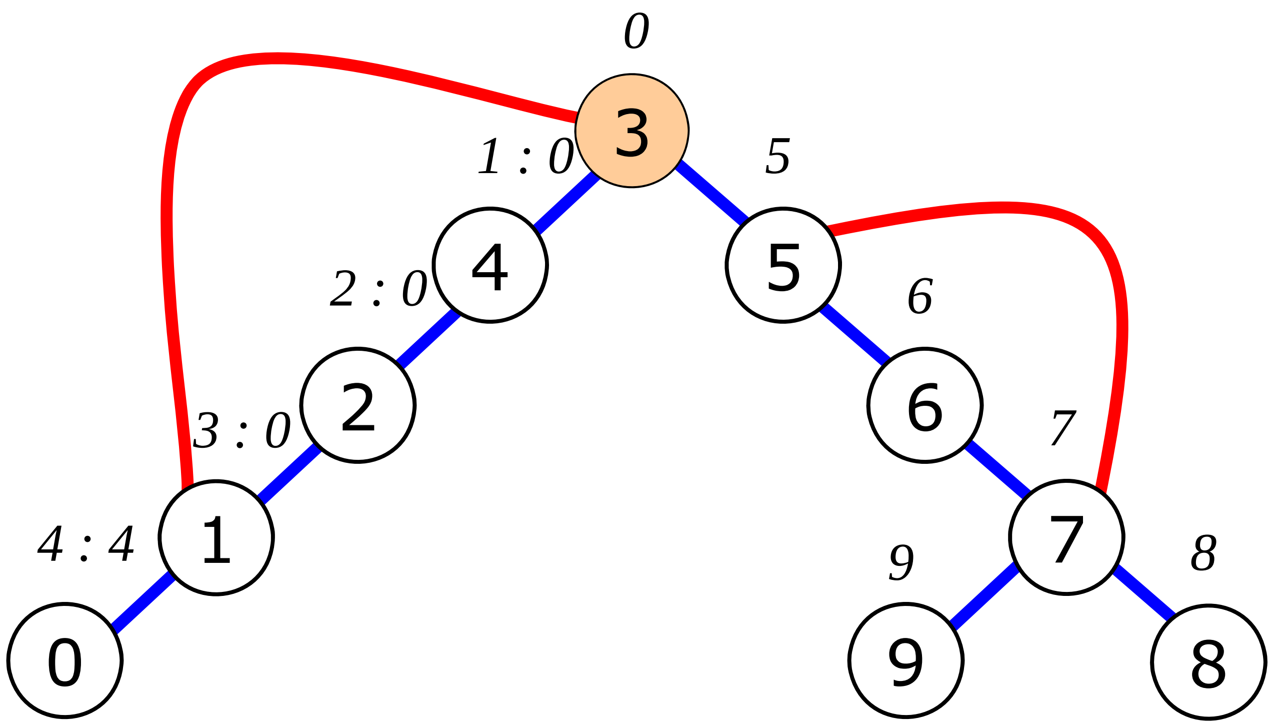
9.5 Biconnected Components

- low(u)
  - Ex) What are low(2)?

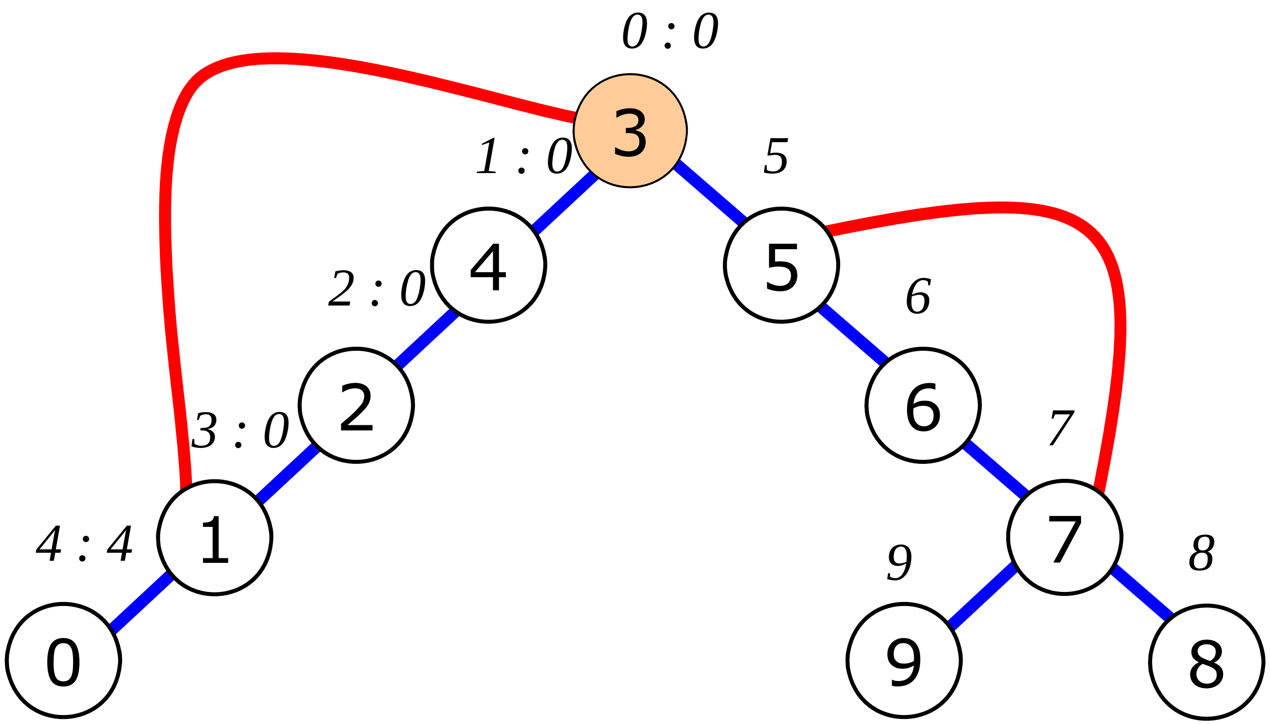


9.5 Biconnected Components

- low(u)
  - Ex) What are low(4)?

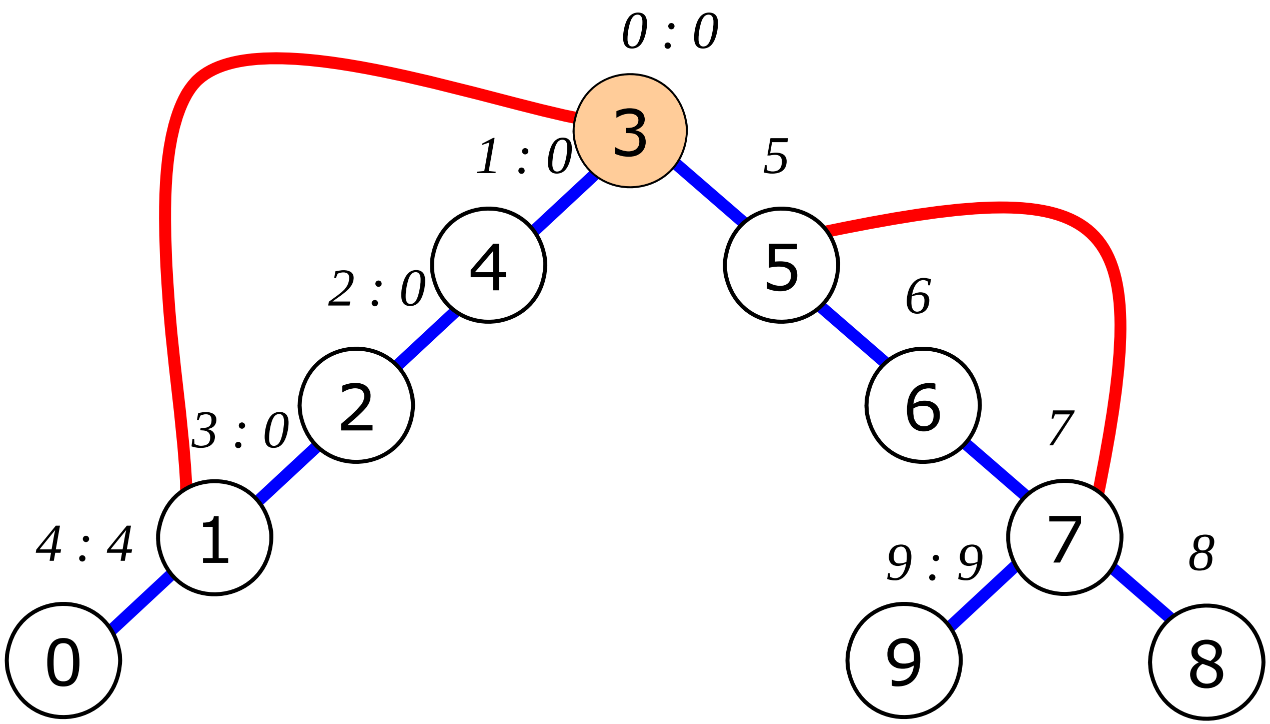


- low(u)
  - Ex) What are low(3)?



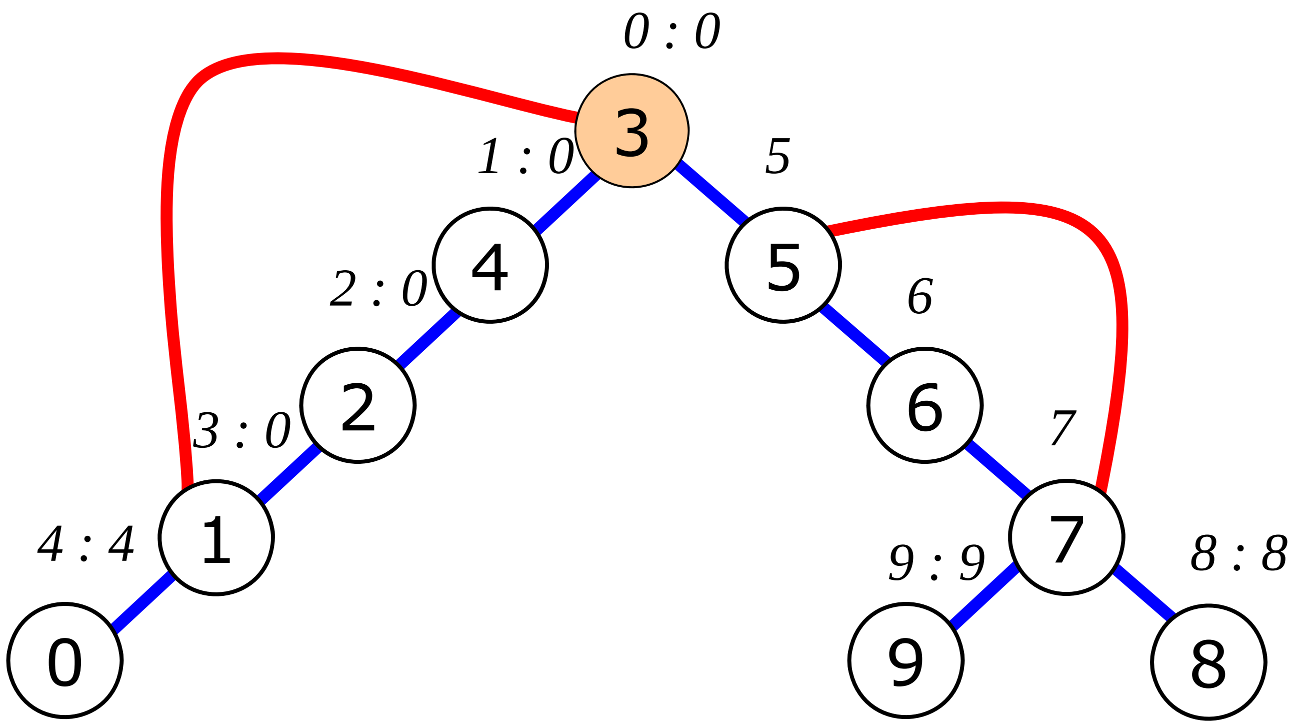
9.5 Biconnected Components

- low(u)
  - Ex) What are low(9)?



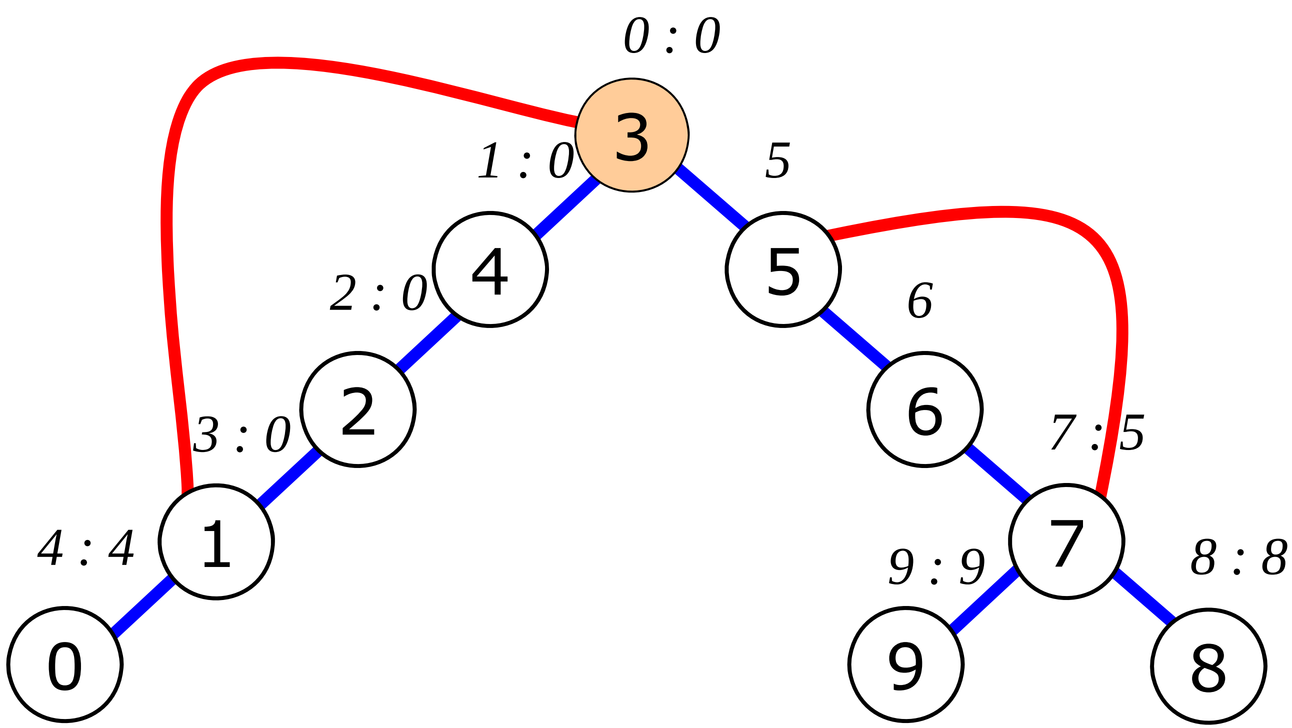
9.5 Biconnected Components

- low(u)
  - Ex) What are low(8)?



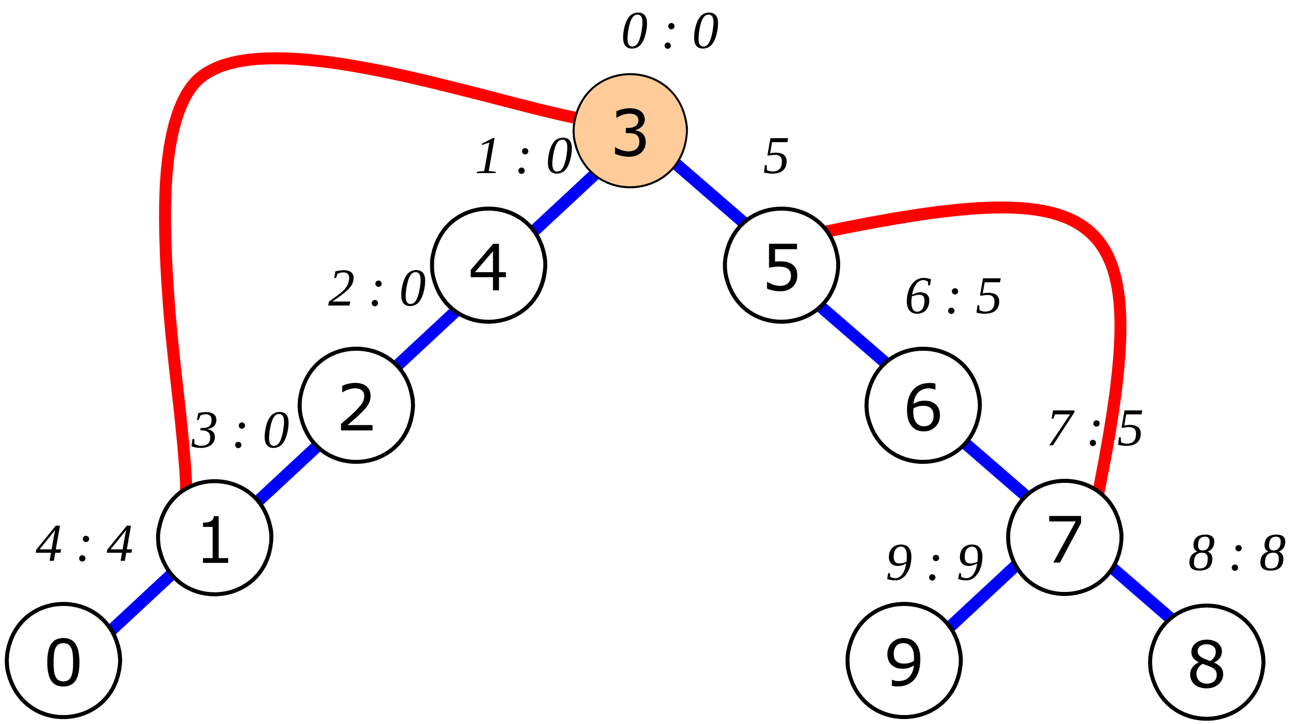
9.5 Biconnected Components

- low(u)
  - Ex) What are low(7)?



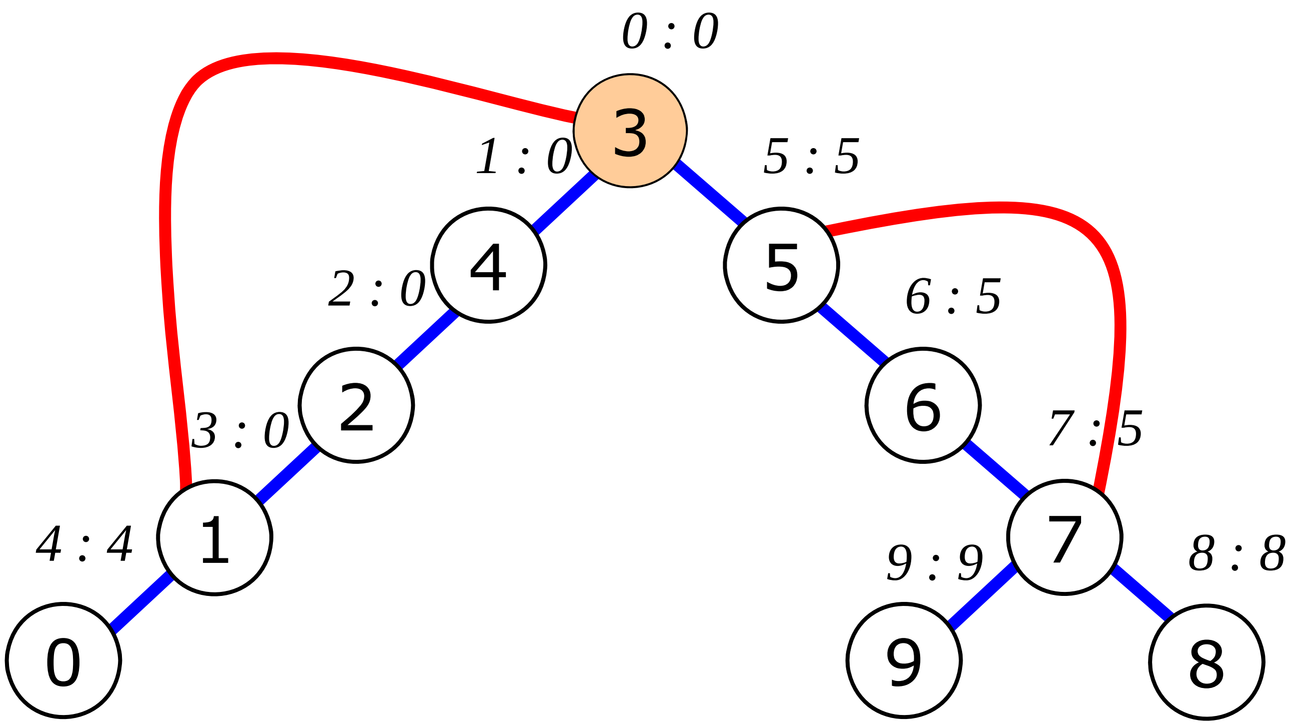
9.5 Biconnected Components

- low(u)
  - Ex) What are low(6)?



9.5 Biconnected Components

- low(u)
  - Ex) What are low(5)?



9.5 Biconnected Components

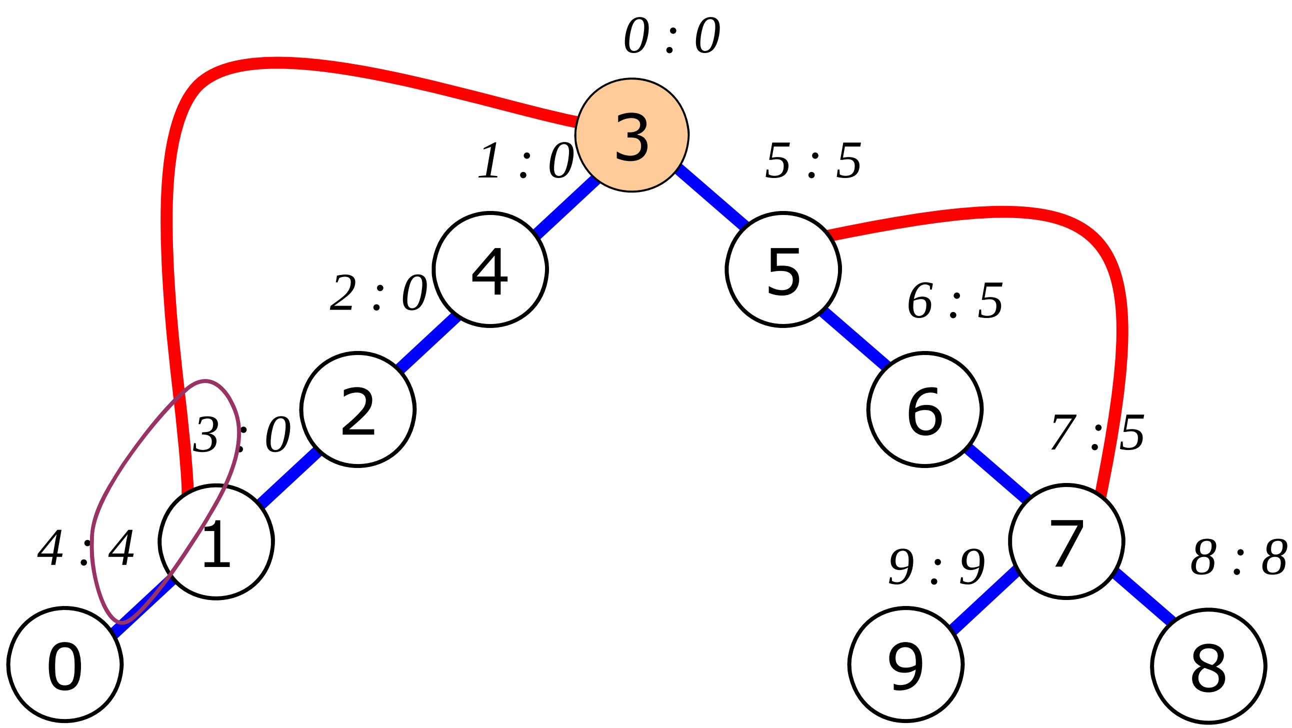
- Articulation points
  - u is an articulation point,
    - if u is either the root of the spanning tree with two or more childs,
    - or u is not a root and has a child w such that low(w) ≥ dfn(u)

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

④ computing articulation points

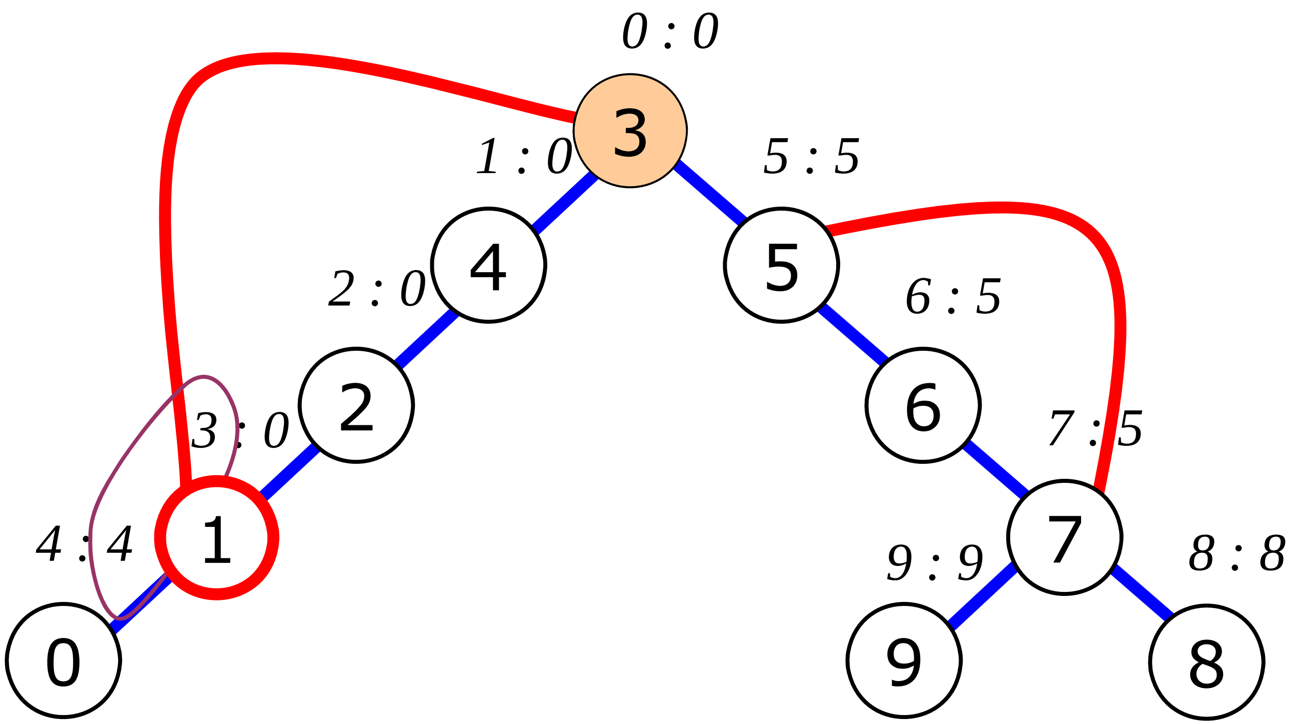
9.5 Biconnected Components

- Determine articulation points
  - At 1, low(w) ≥ dfn(u)?

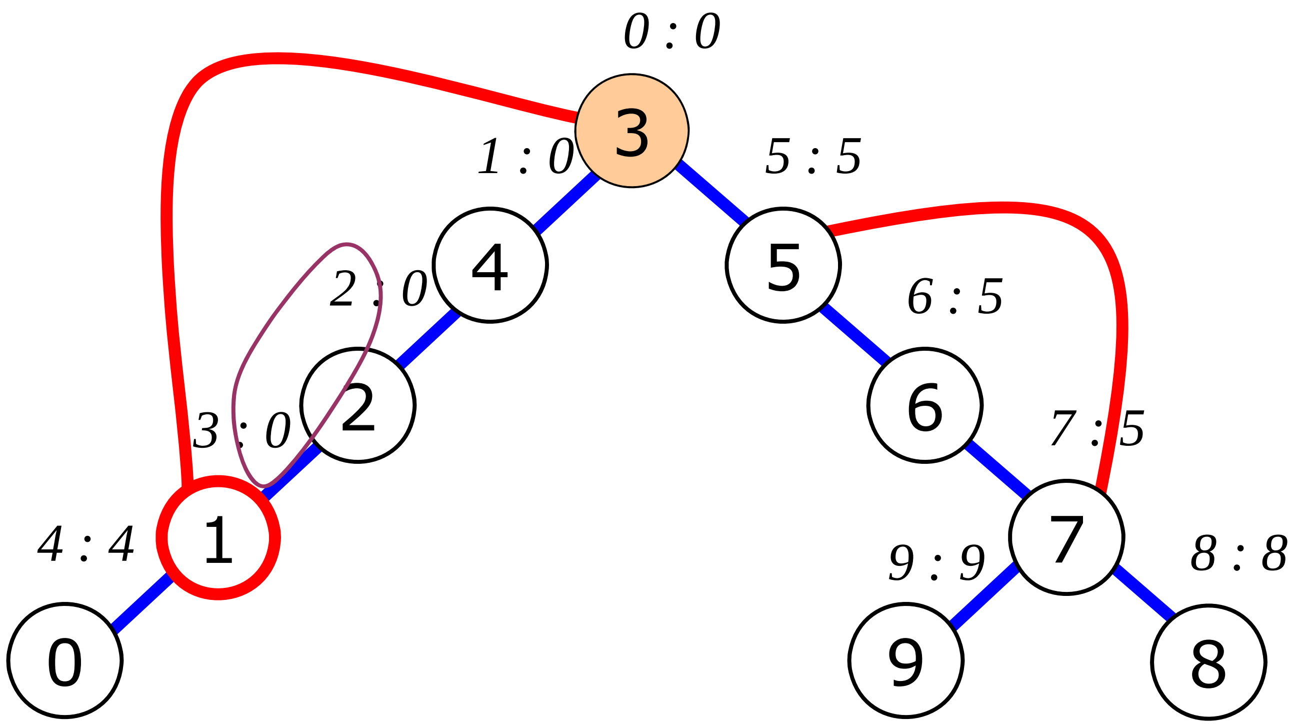


9.5 Biconnected Components

- Determine articulation points
  - At 1, low(w) ≥ dfn(u)?

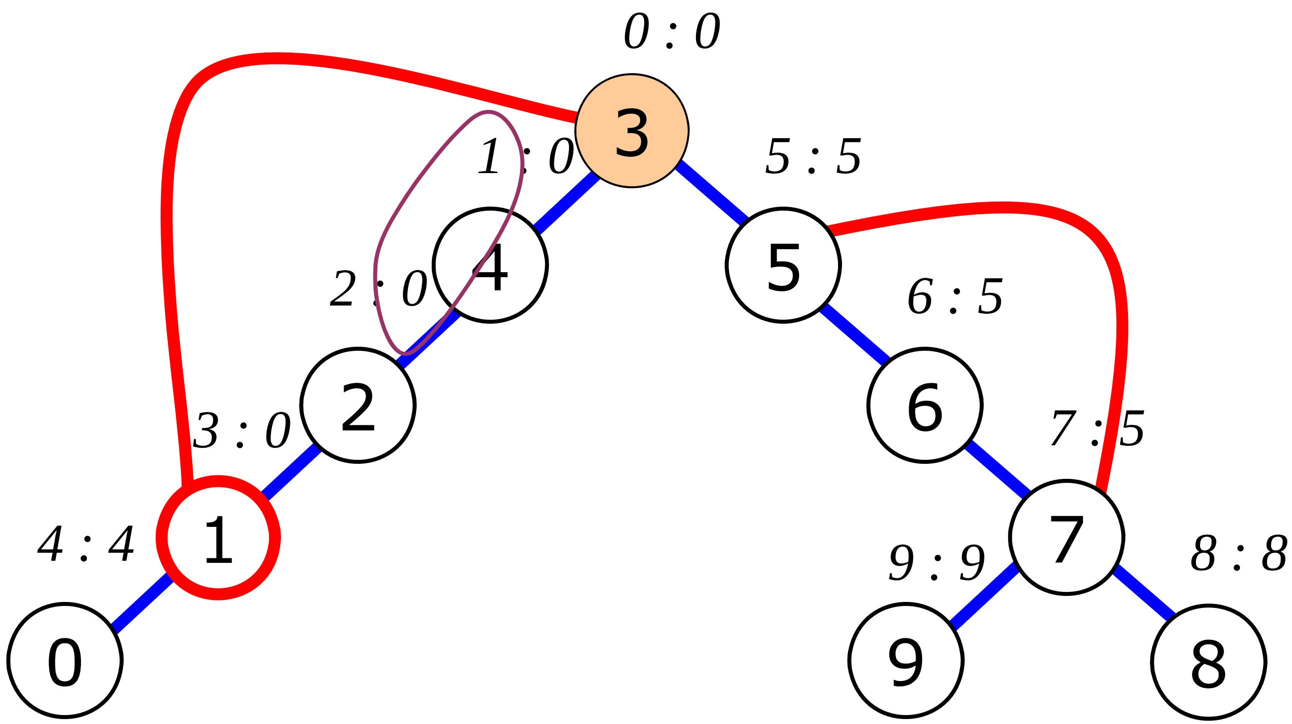


- Determine articulation points
  - At 2, low(w) $\geq$ dfn(u)?



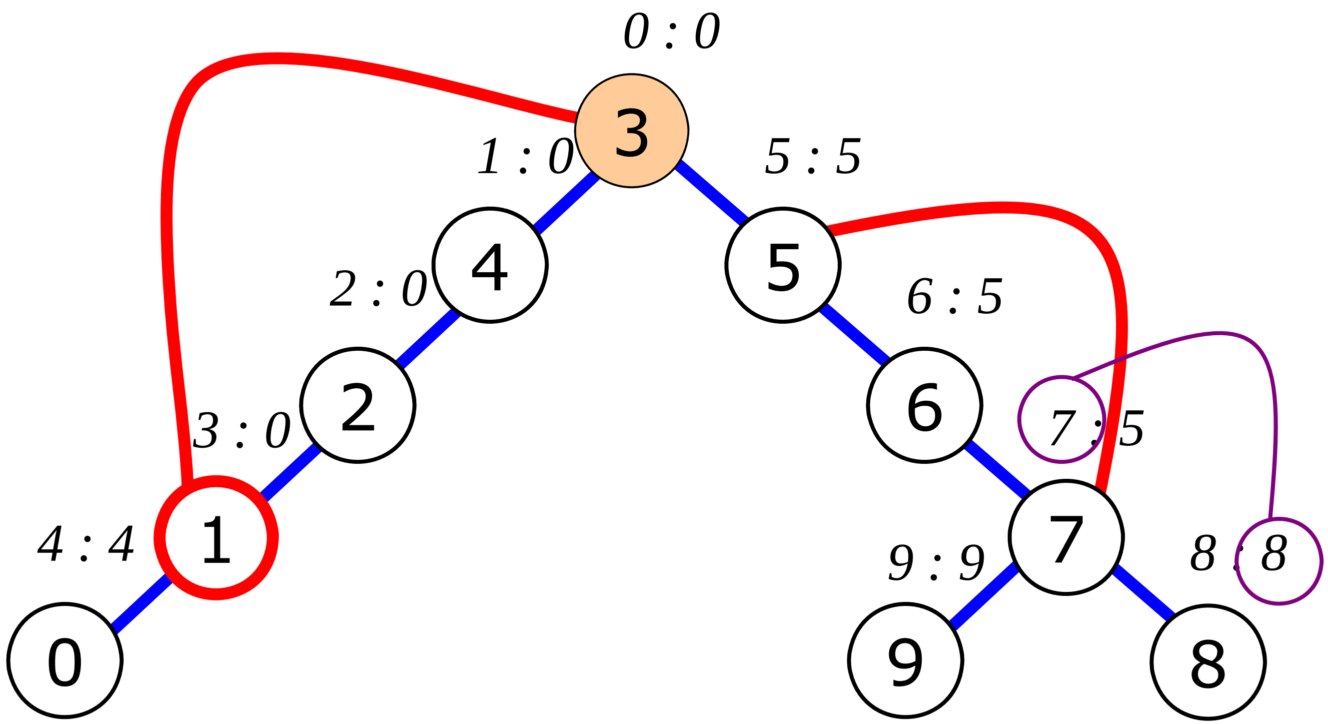
9.5 Biconnected Components

- Determine articulation points
  - At 4, low(w) ≥ dfn(u)?



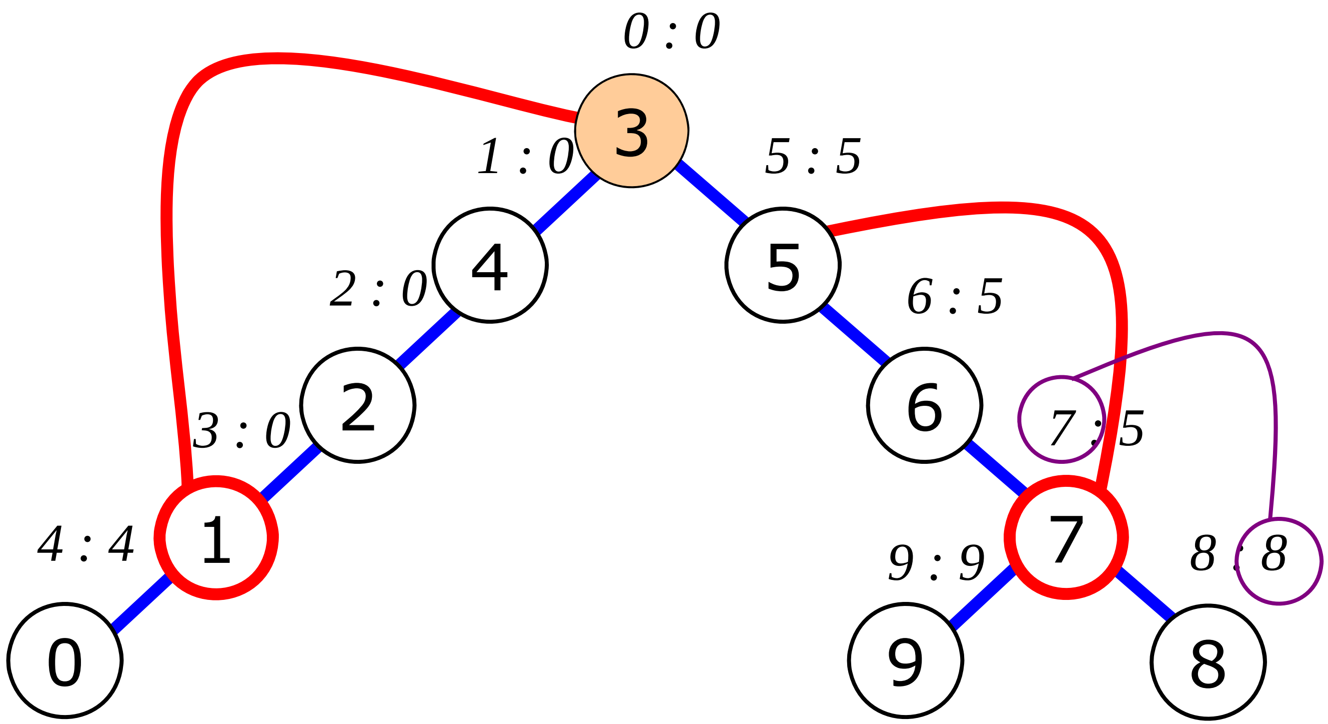
9.5 Biconnected Components

- Determine articulation points
  - At 7, low(w) ≥ dfn(u)?



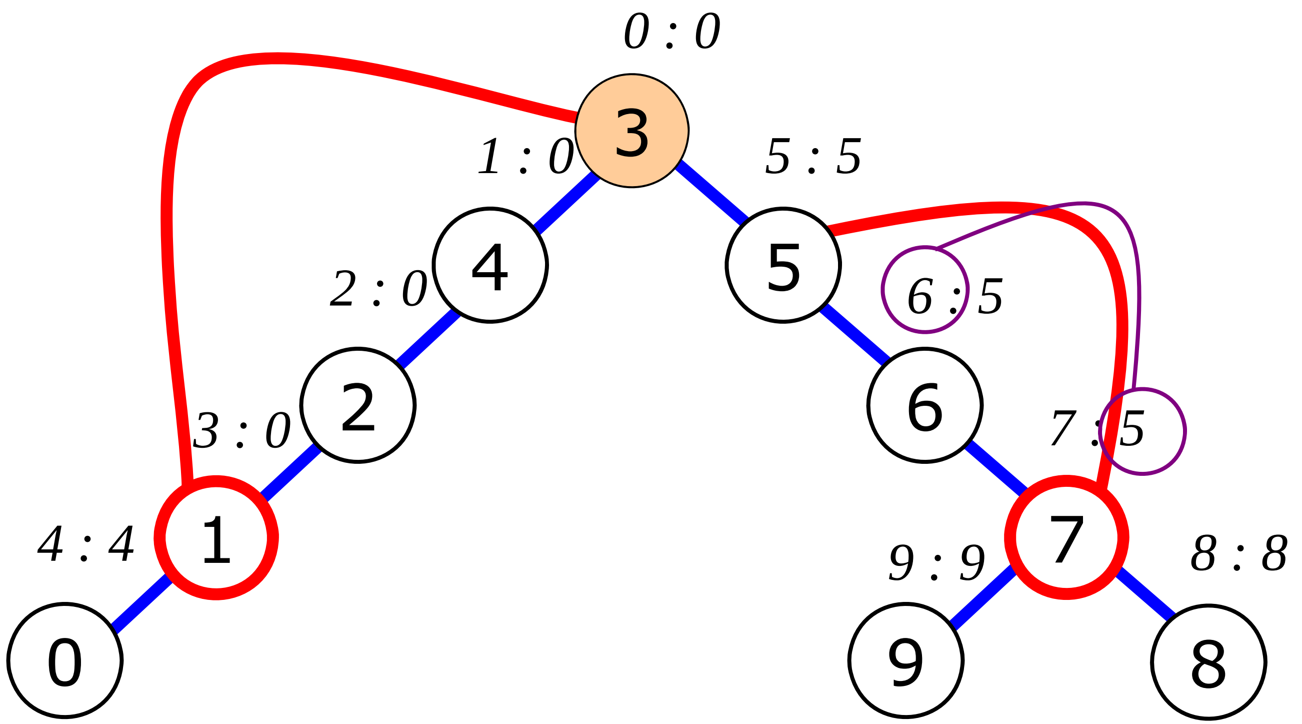
9.5 Biconnected Components

- Determine articulation points
  - At 7, low(w) ≥ dfn(u)?



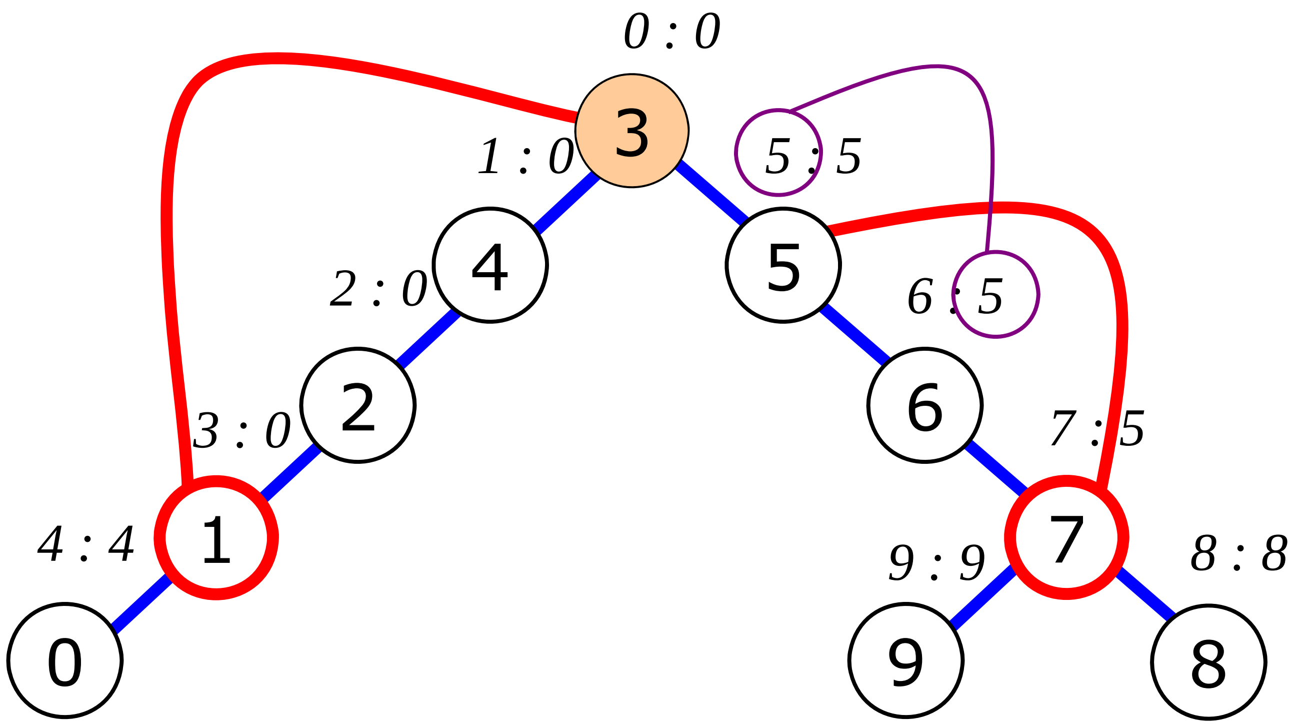
9.5 Biconnected Components

- Determine articulation points
  - At 6, low(w) ≥ dfn(u)?



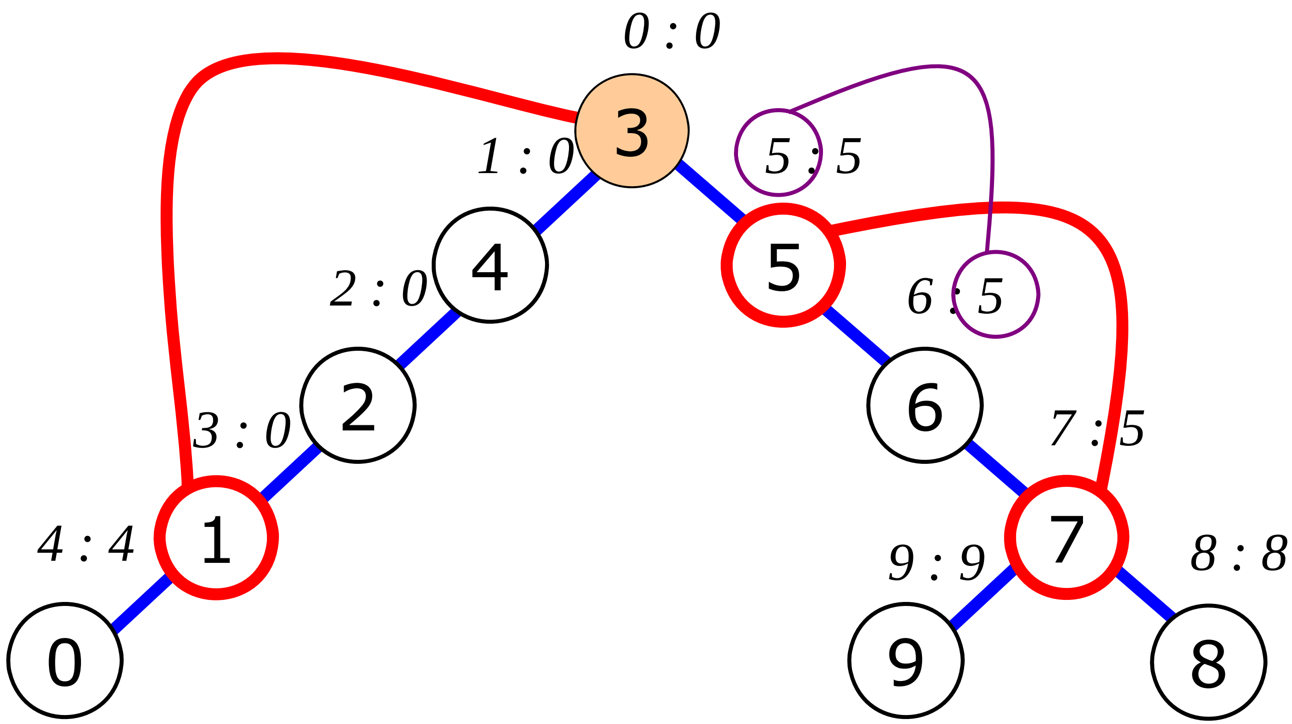
9.5 Biconnected Components

- Determine articulation points
  - At 5, low(w) ≥ dfn(u)?



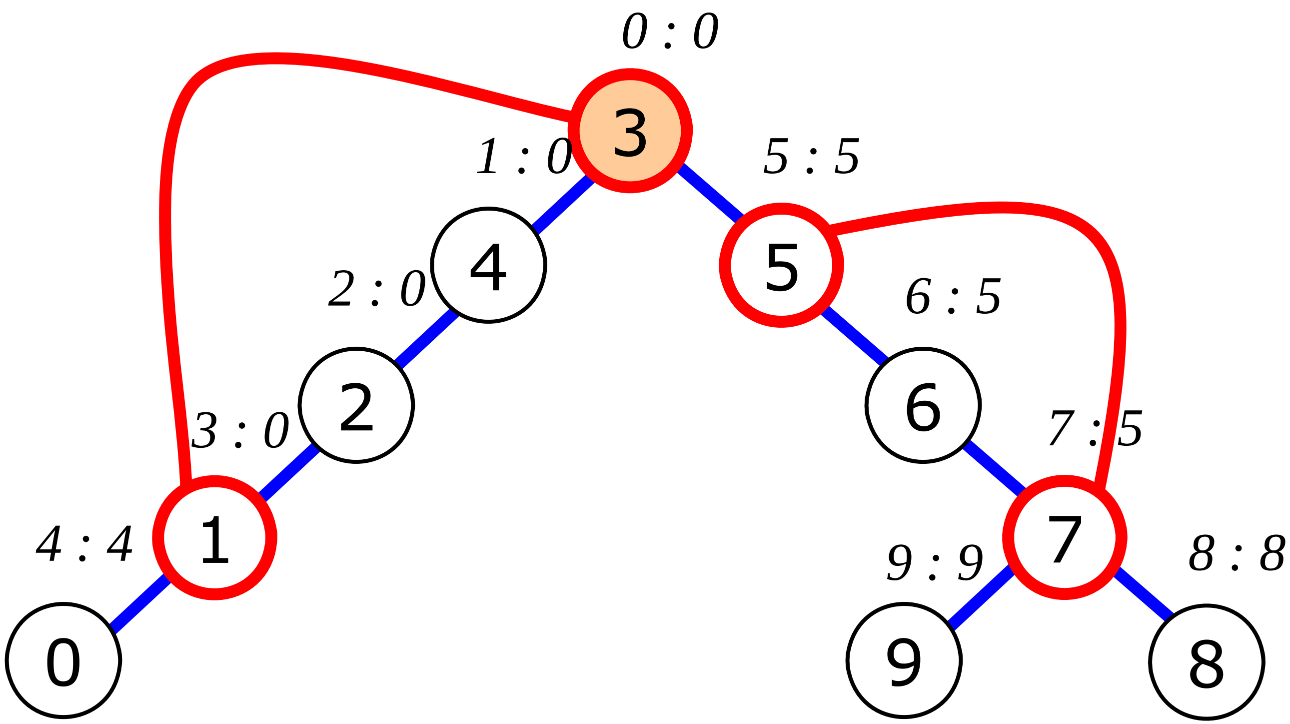
9.5 Biconnected Components

- Determine articulation points
  - At 5, low(w) ≥ dfn(u)?



9.5 Biconnected Components

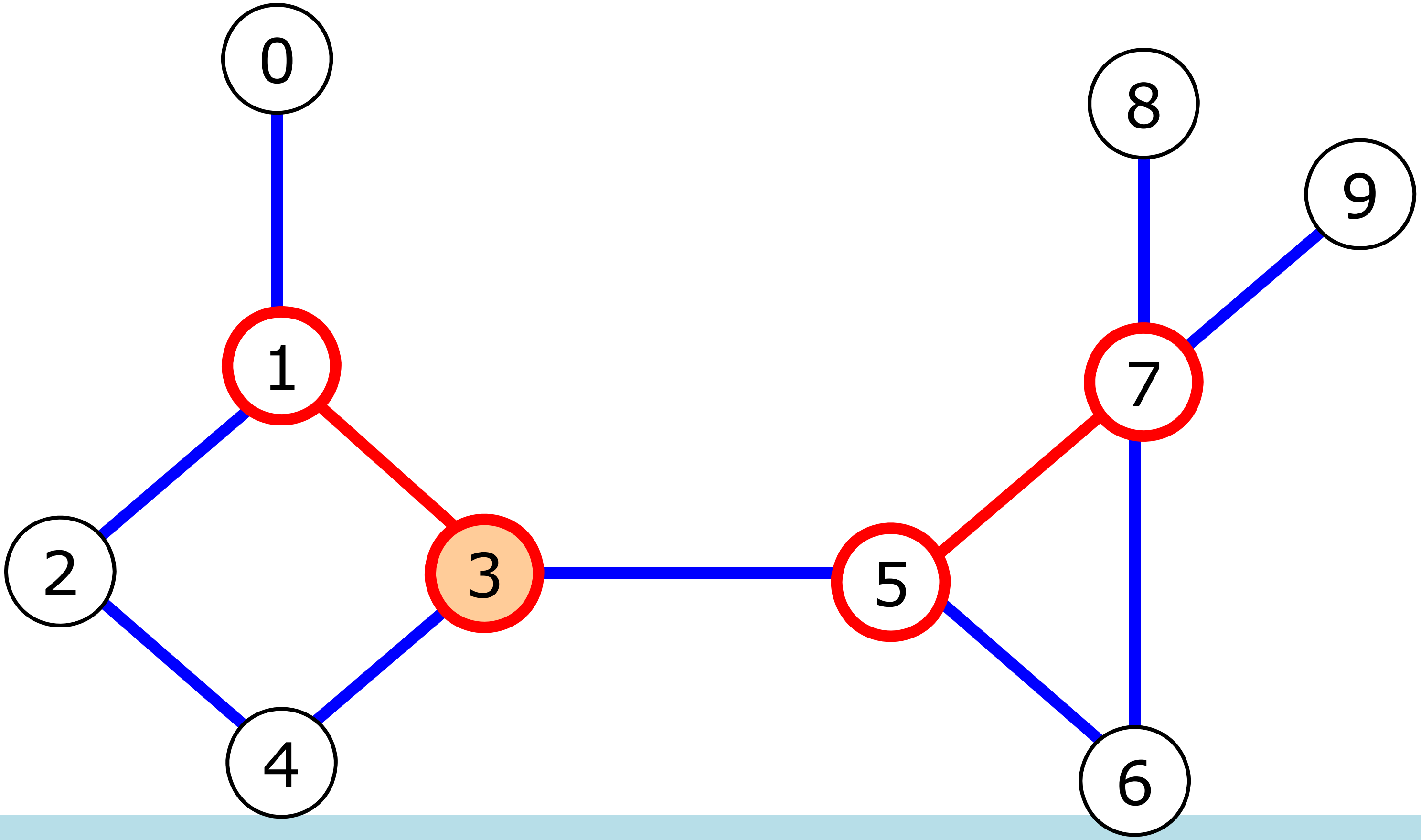
- Total articulation points
  - $\{u \mid low(w) \geq dfn(u)\}$ + root



④ computing articulation points

9.5 Biconnected Components

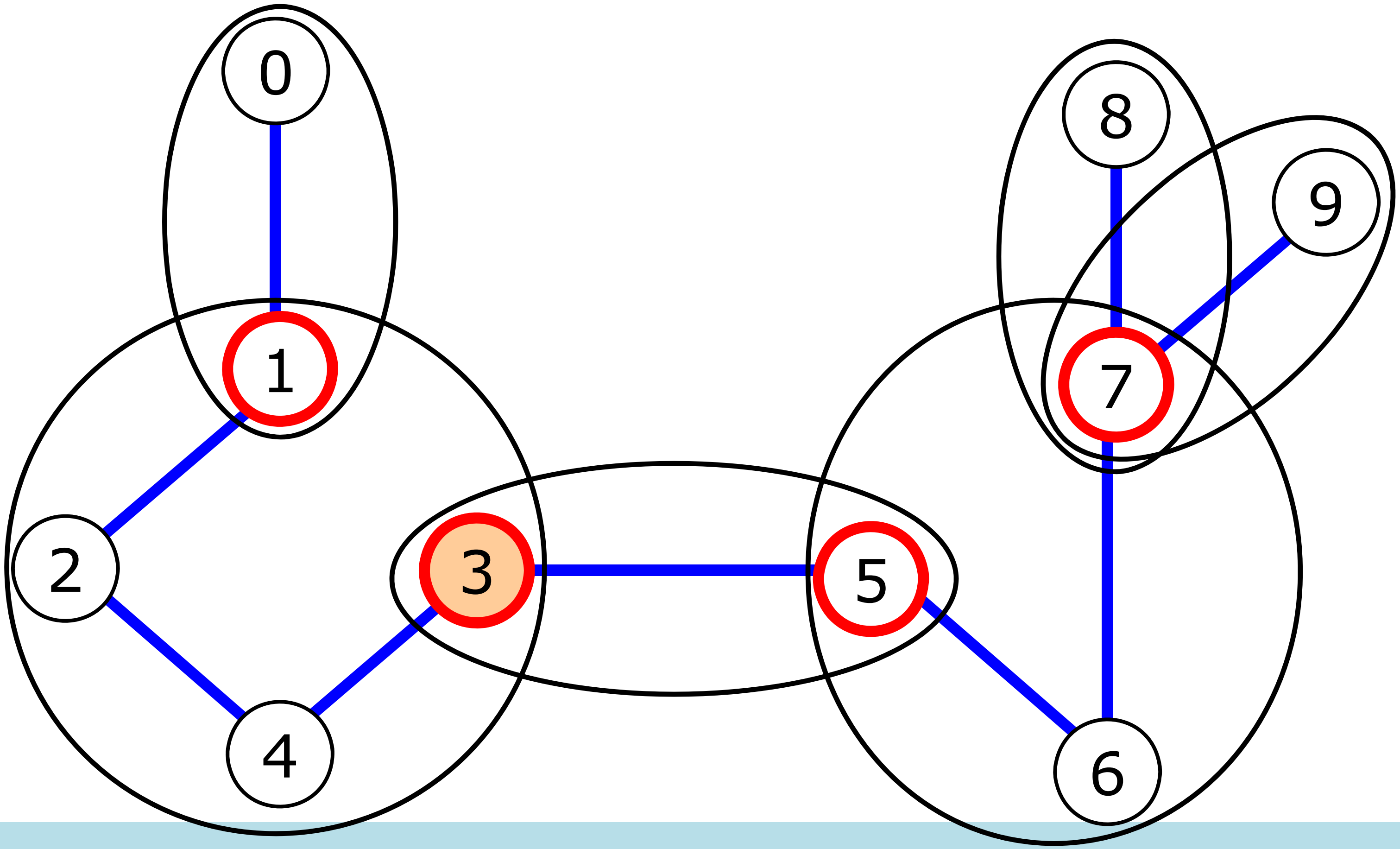
- Total articulation points
  - $\{u \mid low(w) \geq dfn(u)\}$ + root



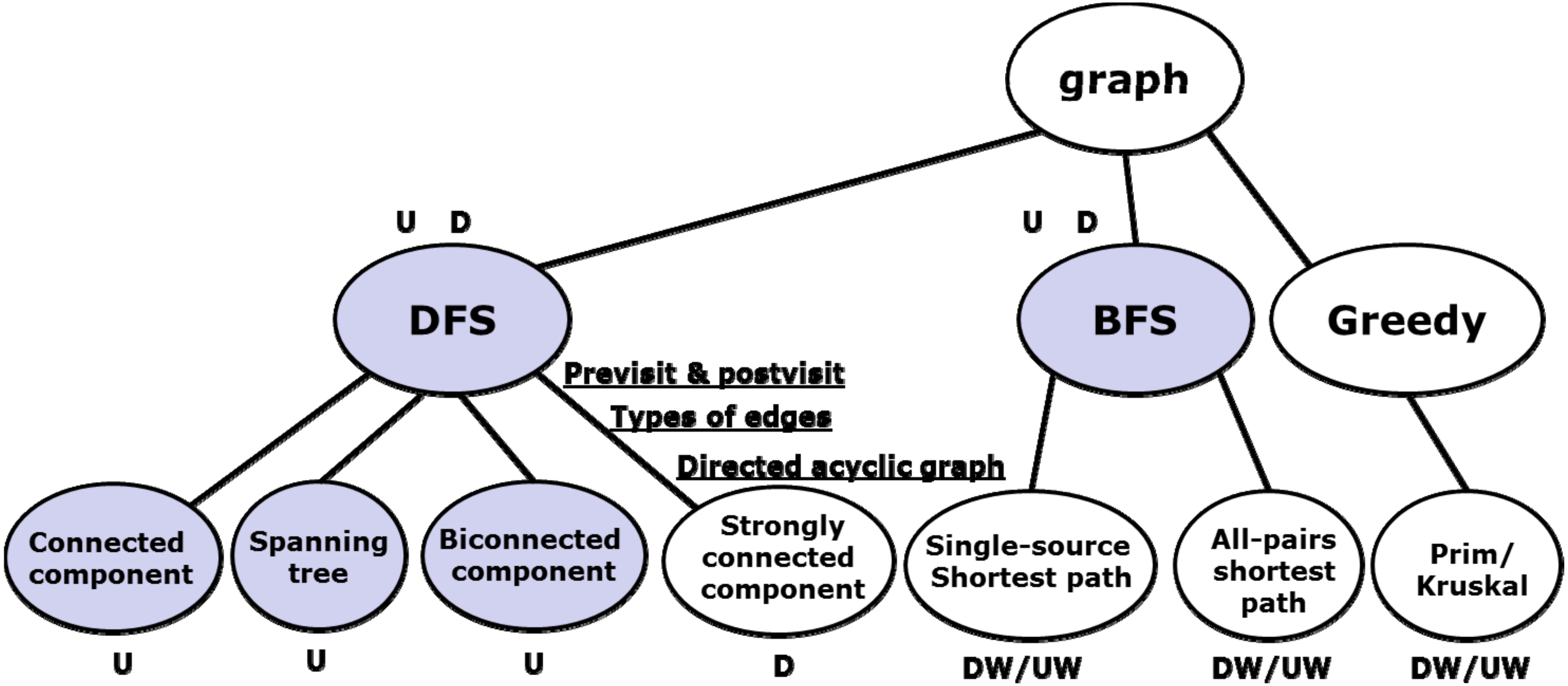
④ computing articulation points

9.5 Biconnected Components

(2) Decomposition of a graph into biconnected components using articulation points



9.5 Biconnected Components



Contents