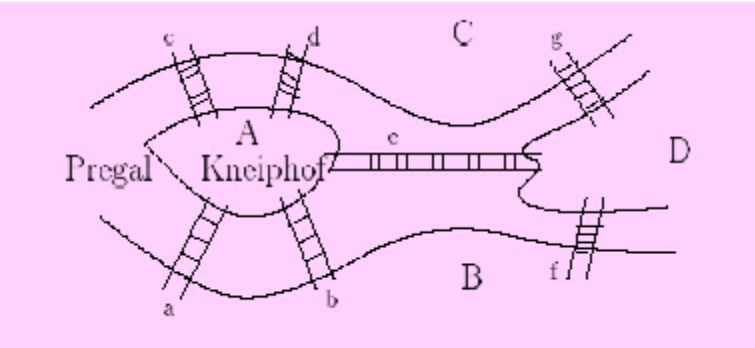
그래프

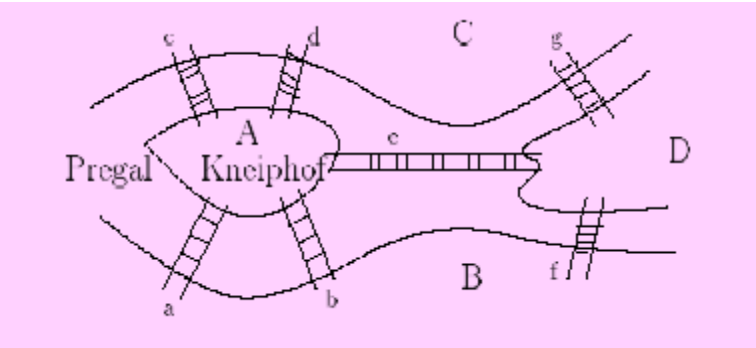
강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터공학과



Graph ?

Koenigsberg Bridge Problem

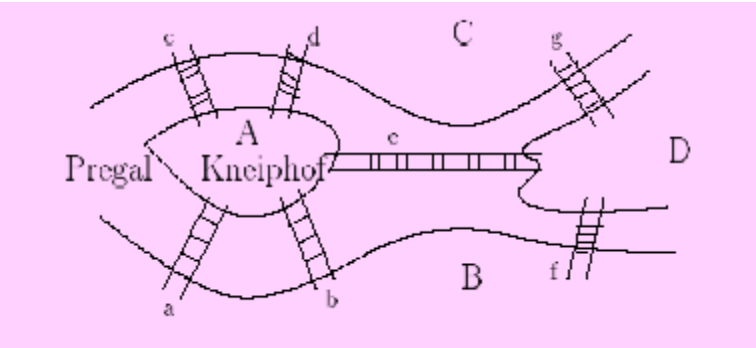
- The first recorded evidence of the use of graphs (1736, by Euler)



"Determine whether starting at some land area it is possible to walk across all the bridges exactly once returning to the starting land area."

Koenigsberg Bridge Problem

- The first recorded evidence of the use of graphs (1736, by Euler)

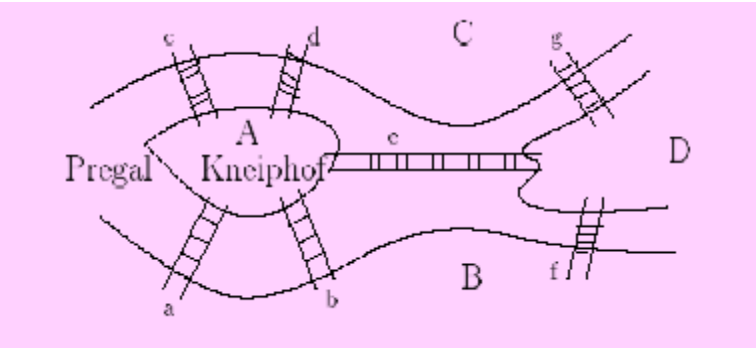


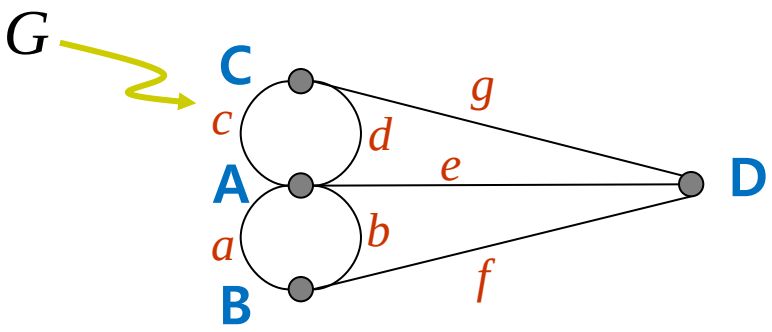
"Determine whether starting at some land area it is possible to walk across all the bridges exactly once returning to the starting land area."

- Euler answered in the negative.

A Walk is Eulerian
iff the degree of each vertex is even.

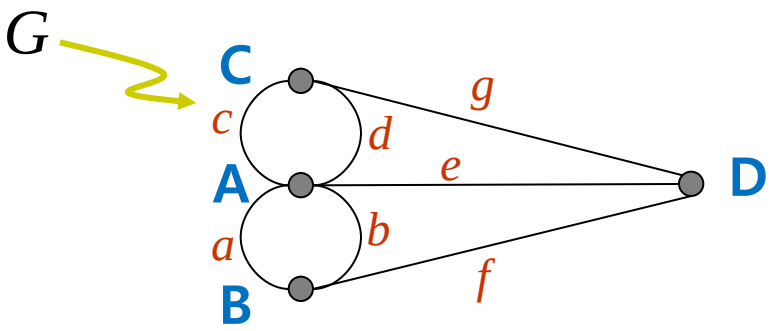
How to represent this geographical info?





$V(G) = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$

$E(G) = \{a, b, c, d, e, f, g\}$



$V(G) = \{A, B, C, D\}$

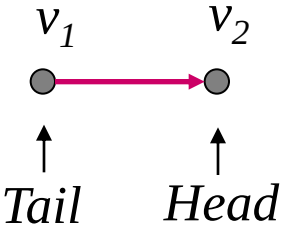
$E(G) = \{a, b, c, d, e, f, g\}$

- A graph $G = (V,E)$ consists of two sets V and E.
  V is a finite non-empty set of vertices.
  E is a set of pairs of vertices (edges).

  $V(G)$: the set of vertices of G
  $E(G)$: the set of edges of G

- Undirected Graph : $(v_1,v_2) = (v_2, v_1)$

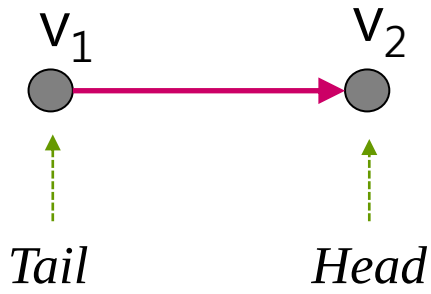
- Directed Graph (Digraph) : $\langle v_1,v_2\rangle \neq \langle v_2, v_1\rangle$



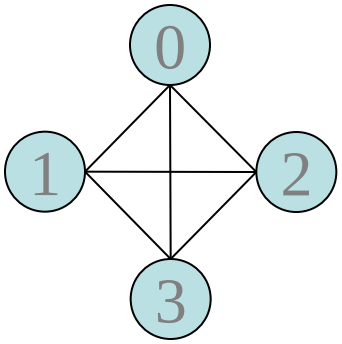
Directed / Undirected Graph

- Undirected Graph : $(v_1,v_2) = (v_2, v_1)$

- Directed Graph (Digraph) : $\langle v_1,v_2 \rangle \neq \langle v_2, v_1 \rangle$



- Undirected Graph

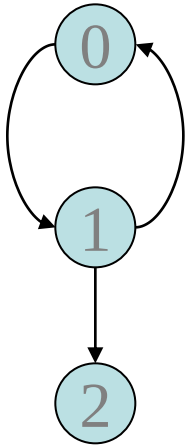


$G_1 = (V_1, E_1)$:

$V_1 = \{ 0,1, 2, 3\}$

$E_1 = \{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)\}$

- Directed Graph (Digraph)



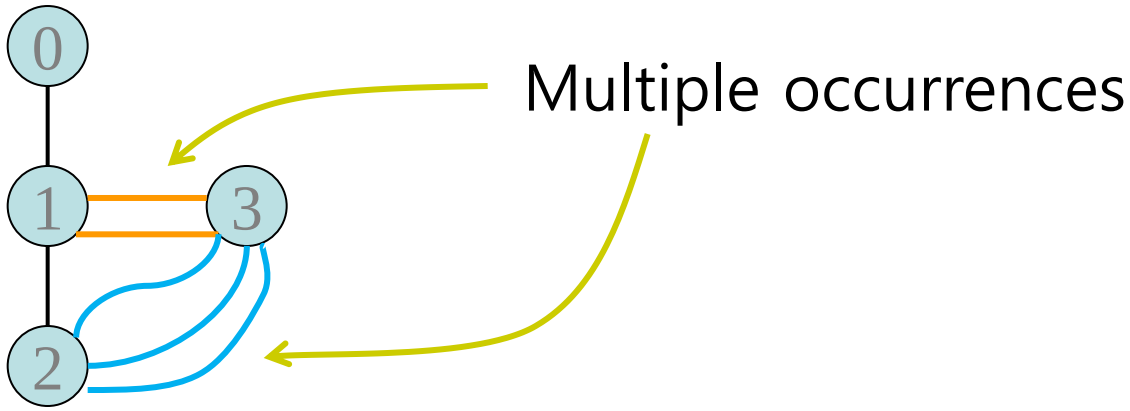
$G_3 = (V_3, E_3)$

$V_3 = \{ 0, 1, 2\}$

$E_3 = \{<0,1>, <1,0>, <1,2>\}$

- Restrictions for graphs in this text.
  1. If (v_1,v_2) or $<v_1, v_2>$ is an edge in $E(G)$, then $v_1 \neq v_2$. That is, self-loops are not allowed.
  2. No multiple occurrences of the same edge. (If not, the graphs are called Multigraph.)

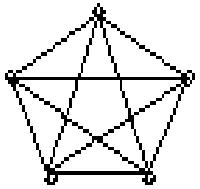
  - (Example) A multigraph



- The number of distinct unordered pairs in a graph with n vertices: $\binom{n}{2} = \frac{n(n-1)}{2}$

- An undirected graph which has n vertices is complete

  if it has exactly $\frac{n(n-1)}{2}$ edges.



$\binom{5}{2} = \frac{5(5-1)}{2} = 10 \quad (edges)$

- For a digraph with n vertices,

  the max number of edges = $\binom{n}{2} \times 2 = n(n-1)$

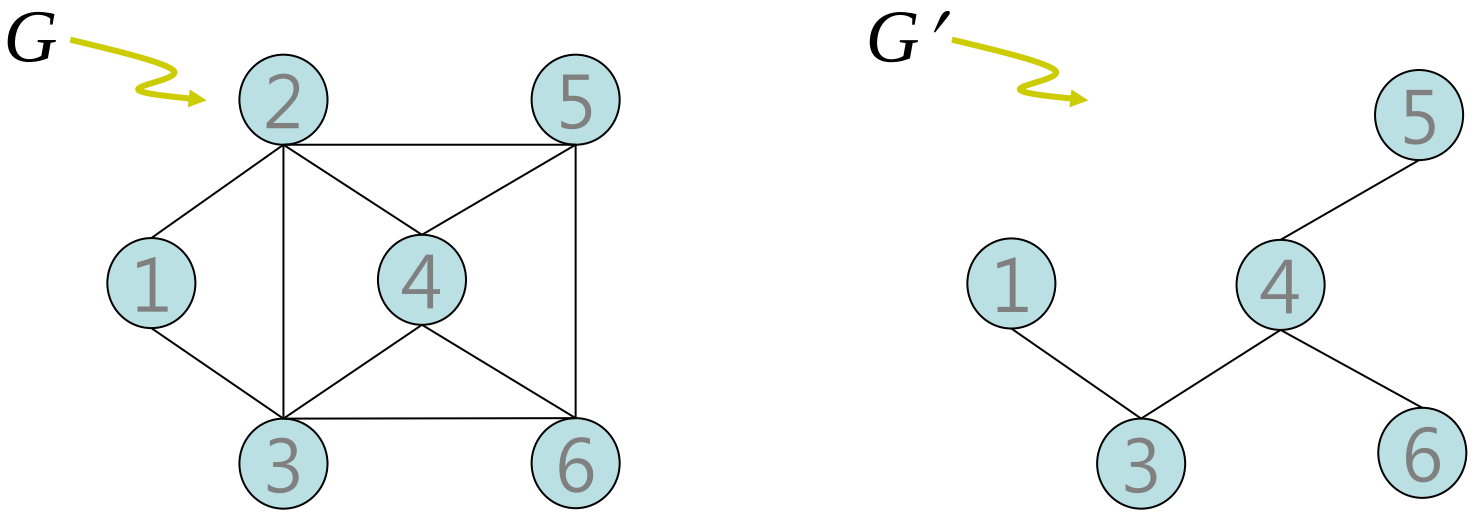
Adjacency

- Let $(v_1,v_2) \in E(G)$ in an undirected graph G.
  - v_1 and v_2 are adjacent
  - (v_1,v_2) is incident on v_1 and v_2

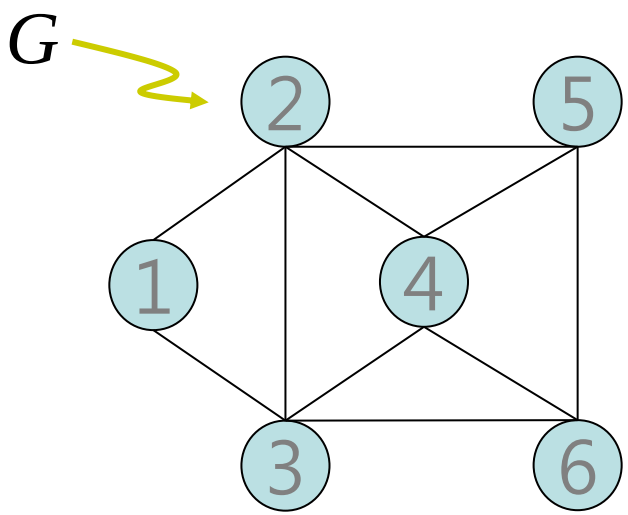
- Let $<v_1,v_2> \in E(G)$ in a digraph G.
  - v_1 is adjacent to v_2
  - v_2 is adjacent from v_1
  - $<v_1,v_2>$ is incident on v_1 and v_2

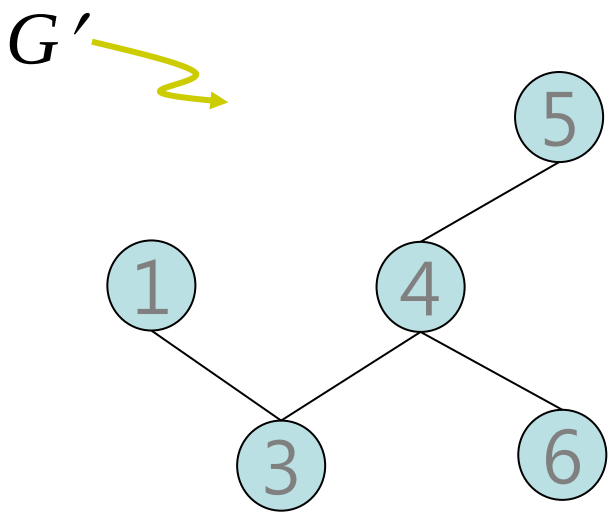
Subgraph

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



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

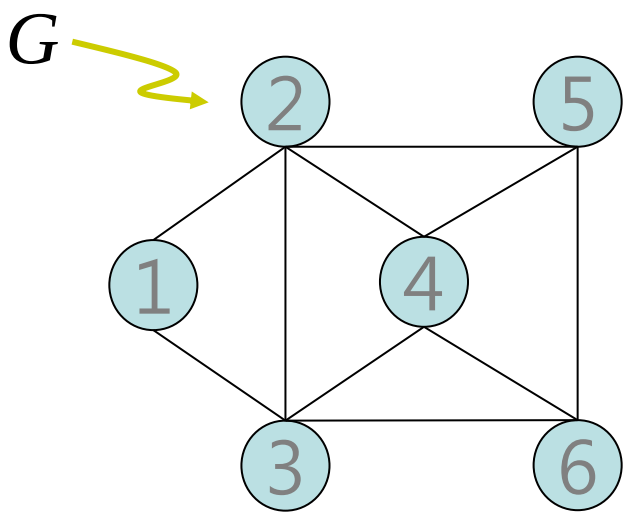


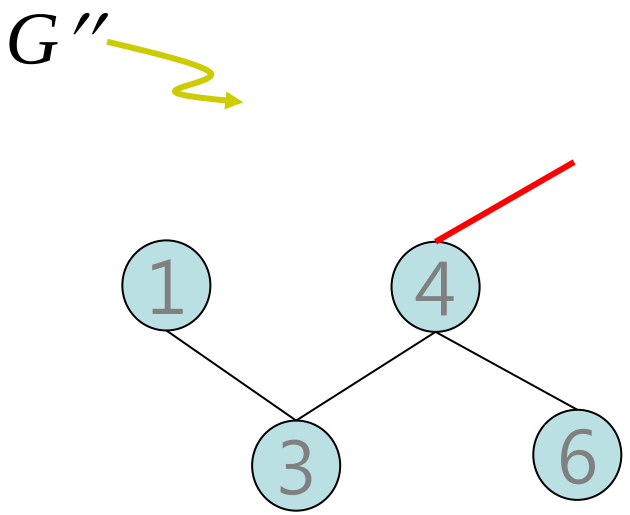


$G = (V, E)$
$V = \{ 1, 2, 3, 4, 5, 6 \}$
$E = \{ (1,2), (1,3), (2,3), (2,4), (2,5), (3,4), (3,6), (4,5), (4,6), (5,6) \}$

$G' = (V', E')$
$V' = \{ 1, 3, 4, 5, 6 \}$
$E' = \{ (1,3), (3,4), (4,5), (4,6) \}$

- $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$
- Is G'' also a subgraph of G''?





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

$G'' = (V'', E'')$
$V'' = \{ 1, 3, 4, 5, 6 \}$
$E'' = \{ (1,3), (3,4), (4,5), (4,6) \}$

Path

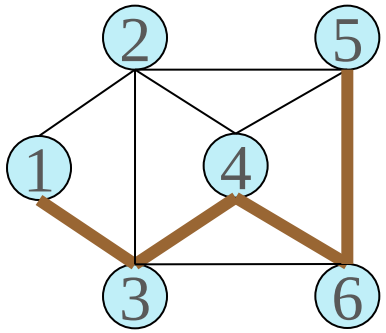
- A path from v_p to v_q in G is a sequence of vertices $v_0, v_1, v_2, \ldots, v_{n-1}, v_n$ such that

$$v_0 = v_p$$
$$v_n = v_q$$

and (v_i, v_{i+1}) [or $<v_i, v_{i+1}>$] $\in E(G)$ *for* $i = 0, 1, \ldots, (n-1)$.

Here, n is called the length of the path from v_p to v_q.

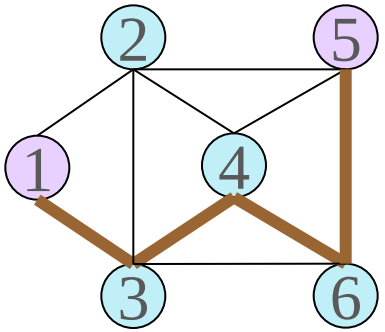
- Example:
  - (1,3) → (3,4) → (4,6)→ (6,5)
  - v_0=1, v_1=3, v_2=4, v_3=6, v_4=5
  - 1,3,4,6,5 is a path from 1(=v_p) to 5(=v_q) of length 4.



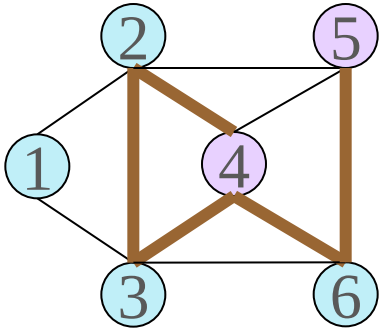
- A path from v_p to v_q in G is cyclic if $v_p = v_q$.
- A path v_0 to v_{n+1} in G is simple if any of the vertices from v_1 to v_n is different from any of the vertices from v_0 to v_{n+1}.
- The length of a path is the number of edges on it.
- A cyclic simple path of length 3 or more is called a cycle.
  - We do not consider the paths of the form v,w,v (path of length 2).

Example

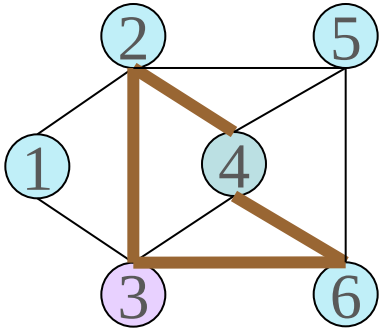
- 1,3,4,6,5 is a path from 1 to 5 of length 4, and simple.



- 4,2,3,4,6,5 is a path from 4 to 5 of length 5, but not simple.



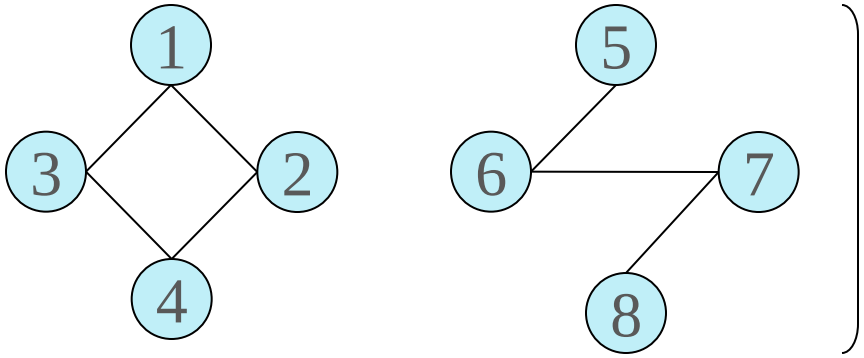
- 3,6,4,2,3 is a simple path and also a cycle.



Connectedness of Undirected Graphs

- The vertices v_i and v_j are connected iff there is a path from v_i to v_j in G.
- An undirected graph G is connected iff for every pair of distinct vertices $v_i, v_j \in V(G)$, there is a path from v_i to v_j in G.
- A maximal connected subgraph of an undirected graph is said to be a (Connected) component.
- An undirected graph is connected iff it has only one component.

- Example: A graph with two components



- G is not connected
- 1 and 4 : connected
- 3 and 7 : disconnected

$G = (V, E)$

$V = \{ 1, 2, 3, 4, 5, 6, 7, 8\}$

$E = \{ (1, 2), (1, 3), (2, 4), (3, 4), (5, 6), (6, 7), (7, 8) \}$

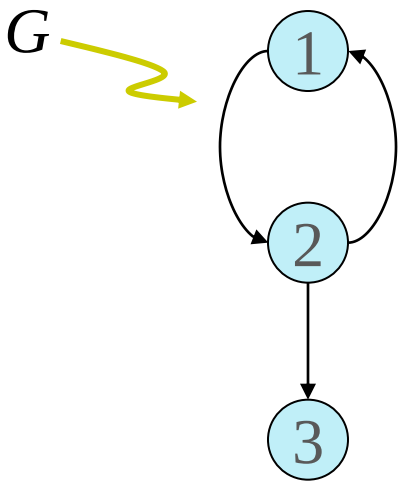
- A graph is acyclic if it has no cycle.
- A (free) tree is a connected acyclic graph.

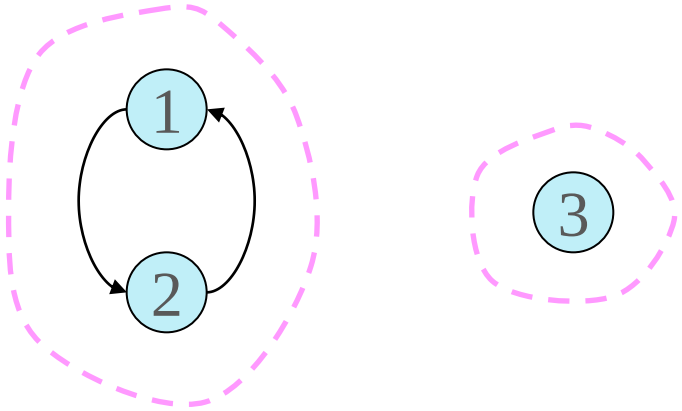
Strong Connectedness of Digraphs

- A digraph G is strongly connected
  iff for every pair of distinct vertices v_i and v_j in $V(G)$,
  there is a directed path from v_i to v_j and also from v_j to v_i
- A maximal subgraph which is strongly connected
  is said to be a strongly connected component
  (a strong component for short).

Strongly Connected Components

- Example:

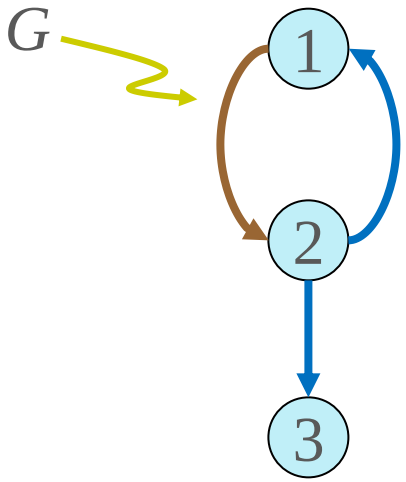




Two Strongly connected Components of graph G

Degree

- The degree of a vertex is the number of edges incident to that vertex.
- Let v be a vertex in a directed graph G.
  - In-degree of a vertex v is the number of edges for which v is the head.
  - Out-degree of a vertex v is the number of edges for which v is the tail.



In-degree of v_2 = 1

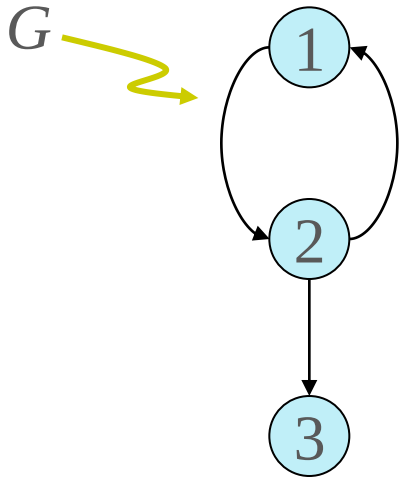
Out-degree of v_2 = 2

Degree

- Let G be any graph with n vertices and e edges. Let d_i = the degree of vertex i, $i = 1, 2, \ldots n$.

$$\text{Then,} \quad e = \frac{1}{2}\sum_{i=1}^{n} d_i$$

- Example
  - $e = 3$
  - $d_1 = 2, d_2 = 3, d_3 = 1$
  - $(d_1+d_2+d_3) / 2 = 3$
  - So, $e = (d_1+d_2+d_3) / 2$



Class "Graph"

Public Functions

- public Graph (int givenNumOfVertices)
  - 그래프 객체 생성.
  - edge는 없고 주어진 수의 vertex만 있는 그래프로 초기화
- public boolean addEdge(Edge e) ;
  - 주어진 edge를 그래프에 삽입한다.
- public boolean doesVertexExist (Vertex v)
  - 그래프에 Vertex v가 존재하는지 여부.
- public boolean doesEdgeExist (Edge e)
  - 그래프에 Edge e 가 존재하는지 여부
- public int numOfVertices ()
  - 그래프의 vertex 개수를 얻는다.
- public int numOfEdges ()
  - 그래프의 edge 개수를 얻는다.
- ..... // 그 밖의 필요한 public functions

Graph Representations (Implementations)

1. Adjacency Matrix
2. Adjacency List

1. Adjacency Matrix

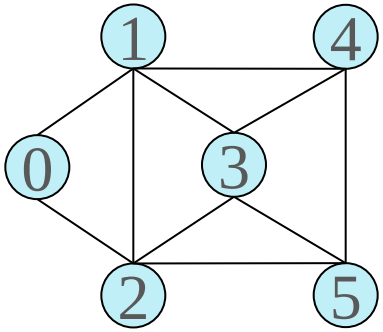
2. Adjacency List

Matrix can represent Adjacency

- Let $G = (V, E)$ with n vertices $(n \geq 1)$.
- The adjacency matrix A for G:

  A[i][j] =1 iff $(v_i, v_j) \in E(G)$

  A[i][j] =0 iff $(v_i, v_j) \notin E(G)$
- Example



Undirected Graph G_1

	0	1	2	3	4	5	sum
0	0	1	1	0	0	0	2
1	1	0	1	1	1	0	4
2	1	1	0	1	0	1	4
3	0	1	1	0	1	1	4
4	0	1	0	1	0	1	3
5	0	0	1	1	1	0	3

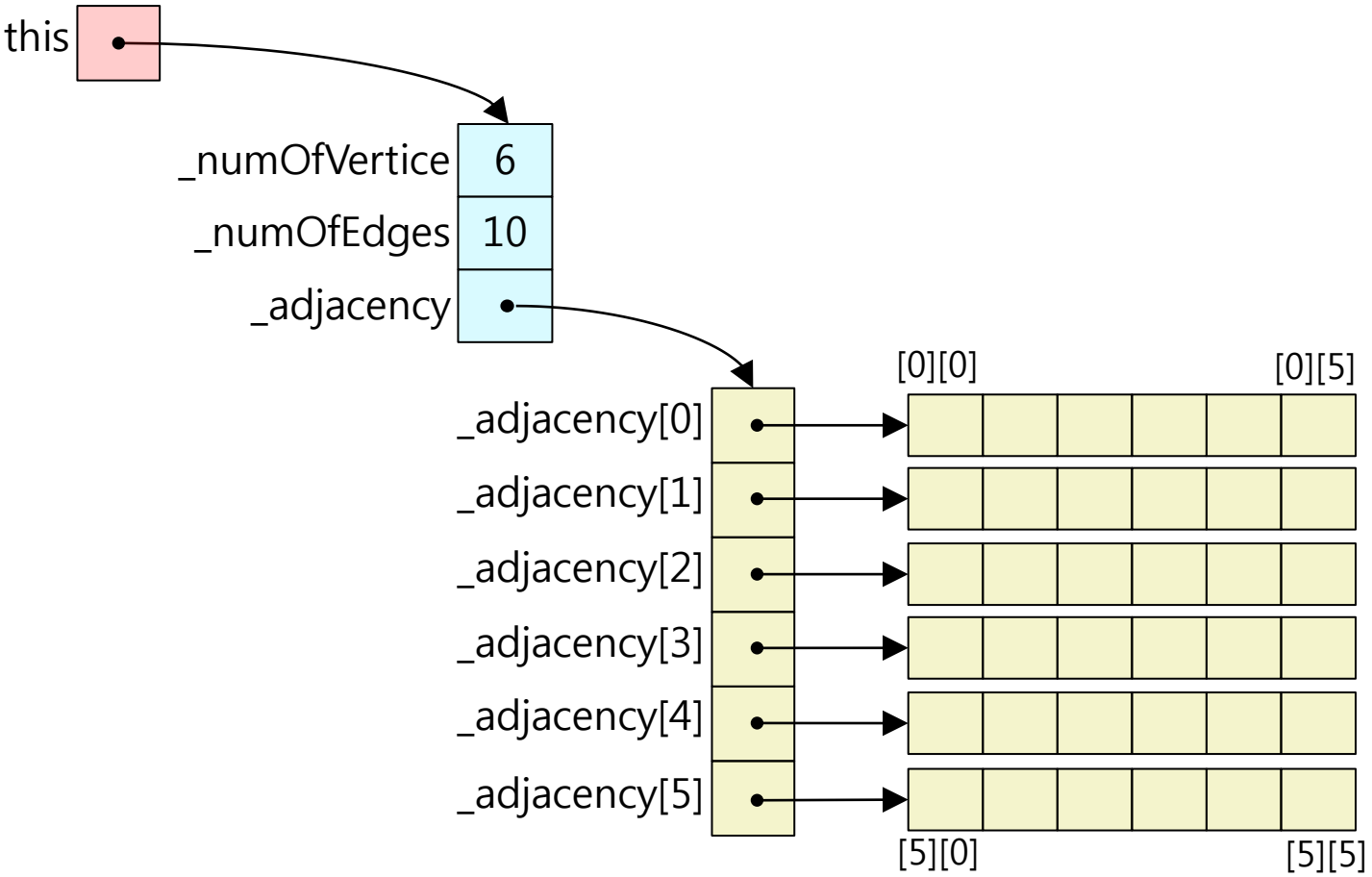
Adjacency matrix of G_1

Adjacency Matrix 의 생성

```
public class Graph {
    private int          _numOfVertices ;
    private int[][]      _adjacency;
    ....


    public Graph(int givenNumOfVertices) {
        this._numOfVertices = givenNumOfVertices ;
        this._adjacency = new int[this._numOfVertices][];
        for (int i = 0; i < this._numOfVertices; i++) {
            this._adjacency[i] = new int[NumOfVertices];
        }
    }

    ......
}
```

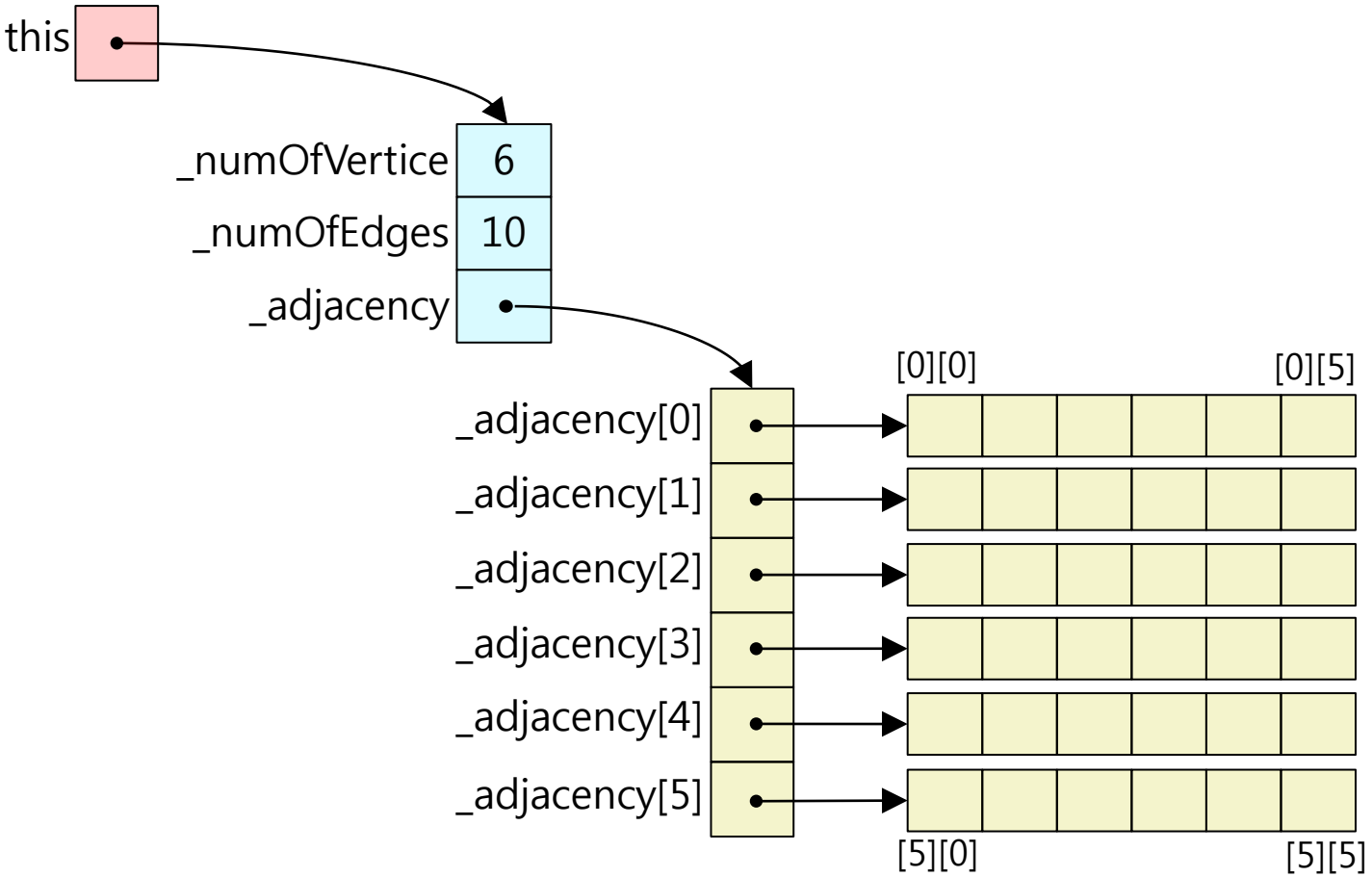
Class Graph의 감추어진 속성

```
private int[][] _adjacency;
private int _numOfVertices ;
private int _numOfEdges ;
```



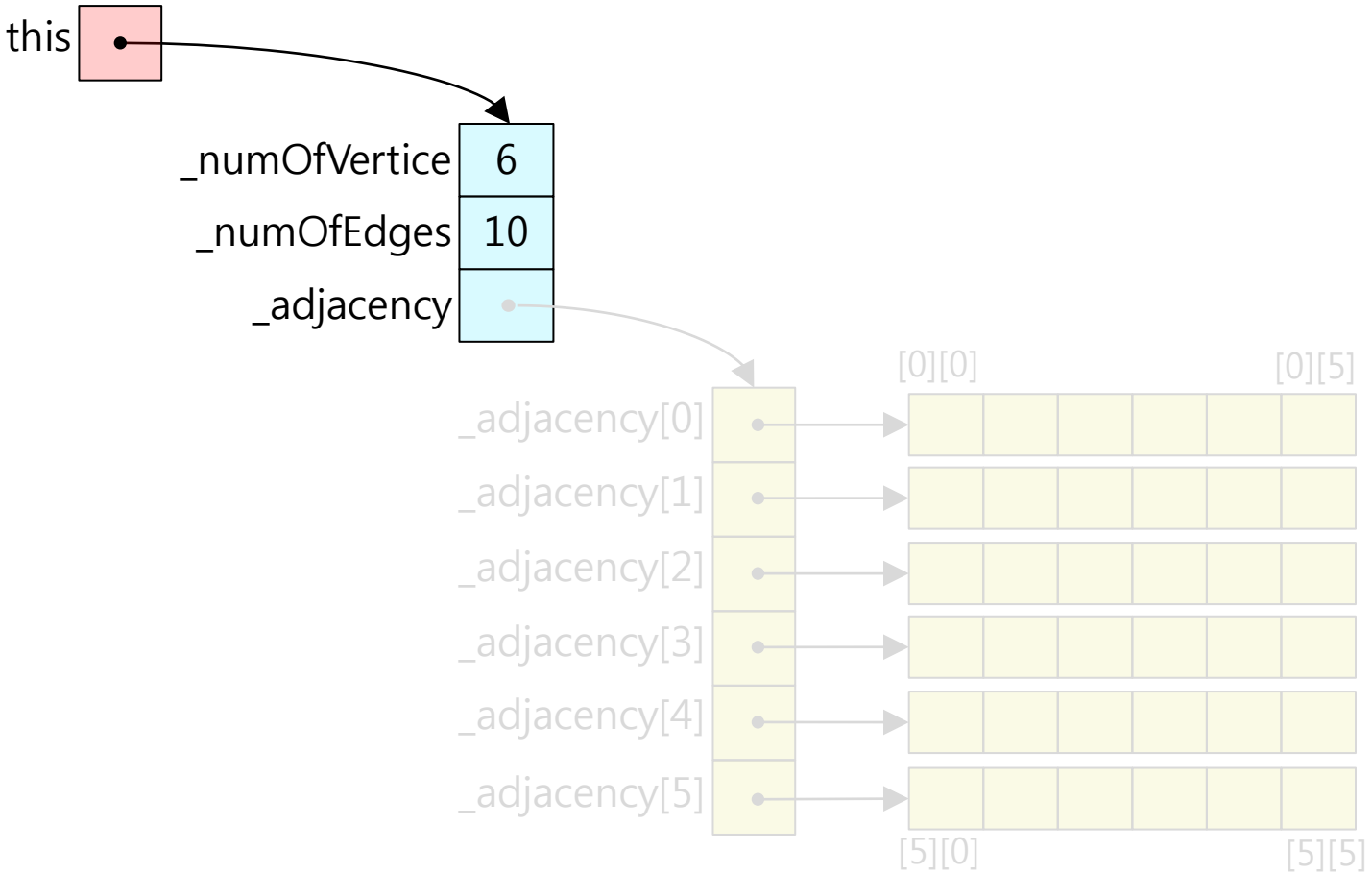
Adjacency Matrix의 동적 할당

```
this._adjacency = new int [ this._numOfVertices ] [ ] ;
for ( int i = 0 ; i < this._numOfVertices ; i++ )
    this._adjacency [ i ] = new int [ this._numOfVertices ] ;
```



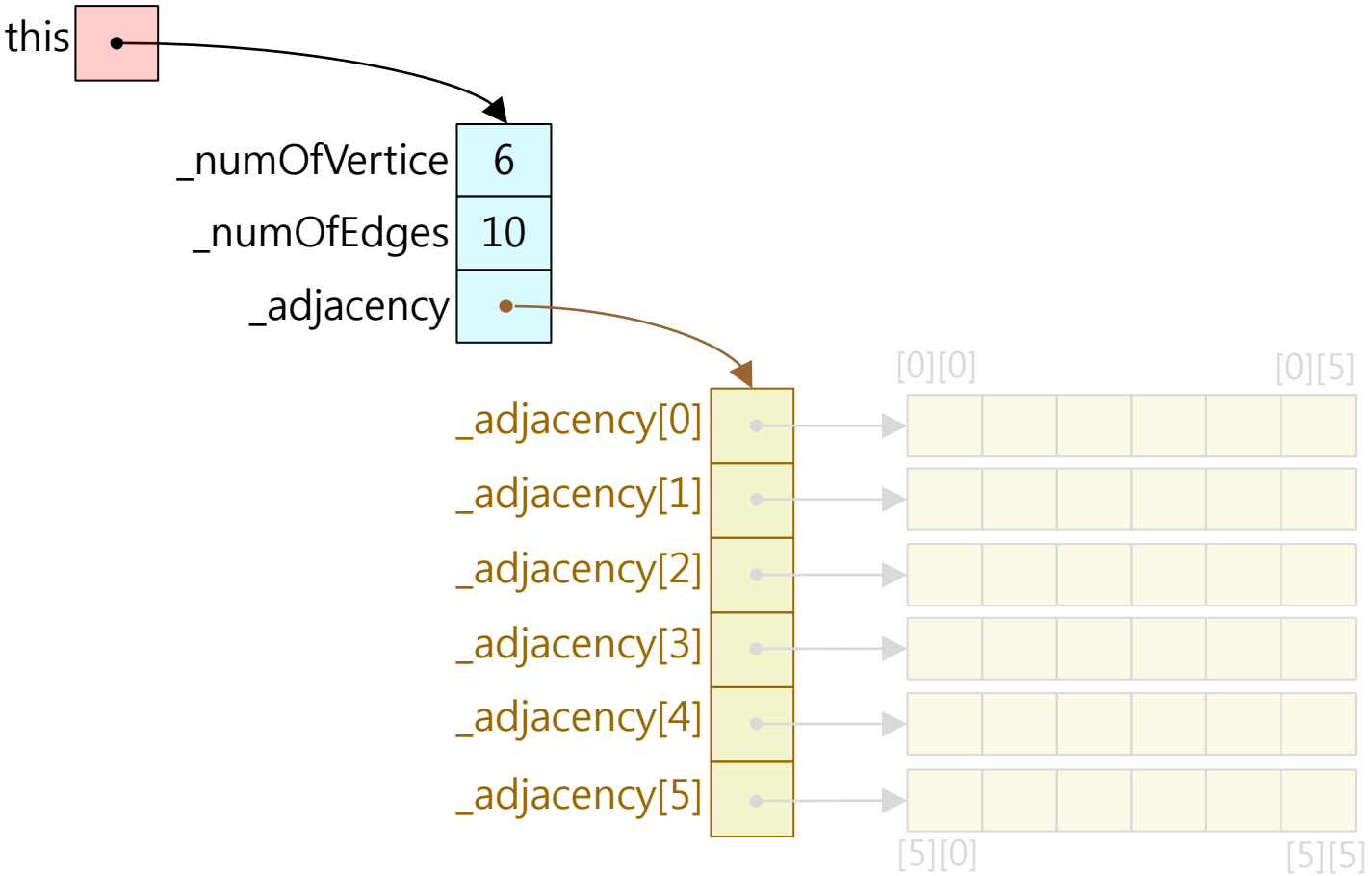
Adjacency Matrix의 동적 할당 과정[1]

```
this._adjacency = new int [ this._numOfVertices ] [ ] ;
for ( int i = 0 ; i < this._numOfVertices ; i++ )
    this._adjacency [ i ] = new int [ this._numOfVertices ] ;
```



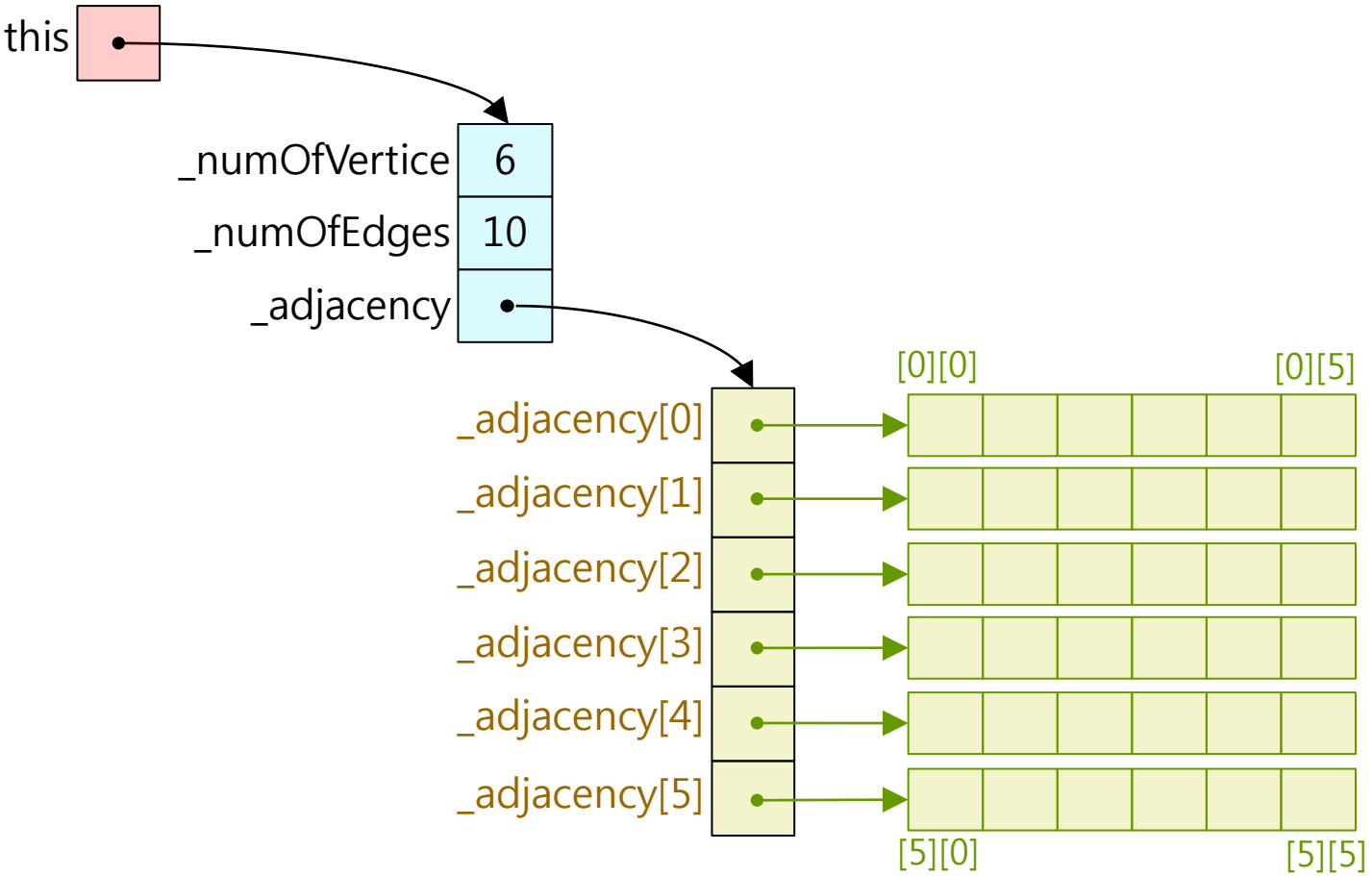
Adjacency Matrix의 동적 할당 과정[2]

```
this._adjacency = new int [ this._numOfVertices ] [ ] ;
for ( int i = 0 ; i < this._numOfVertices ; i++ )
    this._adjacency [ i ] = new int [ this._numOfVertices ] ;
```



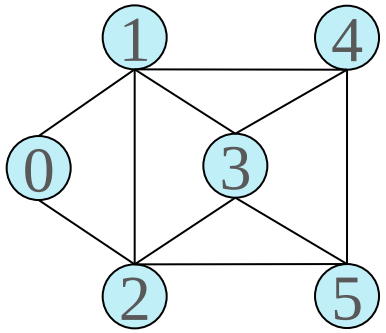
Adjacency Matrix의 동적 할당 과정[3]

```
this._adjacency = new int [ this._numOfVertices ] [ ] ;
for ( int i = 0 ; i < this._numOfVertices ; i++ )
    this._adjacency [ i ] = new int [ this._numOfVertices ] ;
```



Properties of Adjacency Matrix

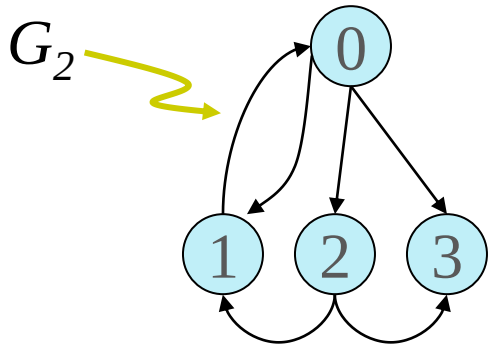
- For an undirected graph,
  - The adjacency matrix is symmetric. That is,
    A[i][j] = 1 iff A[j][i] = 1.
  - The sum of the i-th row $\sum_{j=0}^{n-1} A[i][j]$ is the degree of the vertex i .
- Example:
  - $d_3 = 4$
  - $\sum_{j=0}^{5} A[3][j] = 4$
  - So, $d_3 = \sum_{j=0}^{5} A[3][j]$



	0	1	2	3	4	5	sum
0	0	1	1	0	0	0	2
1	1	0	1	1	1	0	4
2	1	1	0	1	0	1	4
3	0	1	1	0	1	1	4
4	0	1	0	1	0	1	3
5	0	0	1	1	1	0	3

Properties of Adjacency Matrix

- For a directed graph,
  - The row sum $\sum_{j=0}^{n-1} A[i][j]$ is the out-degree of v_i.
  - The column sum $\sum_{i=0}^{n-1} A[i][j]$ is the in-degree of v_j.



	0	1	2	3	sum
0	0	1	1	1	3
1	1	0	0	0	1
2	0	1	0	1	2
3	0	0	0	0	0
sum	1	2	1	2	

Complexities

- Space complexity: $O(n^2)$
- Time complexity
  - Examining all the entries: $O(n^2)$
    - There are total $(n^2 - n)$ entries.

- For sparse matrices,
  the adjacency matrix is inefficient both in time and space since $(e << n^2/2)$.

1. Adjacency Matrix

2. Adjacency List

Data structures for Adjacency Lists

- One linked list for each vertex.
  - The n rows of the adjacency matrix are represented as n linked lists.
- The nodes in the list for the vertex v ≡ the vertices adjacent from v.

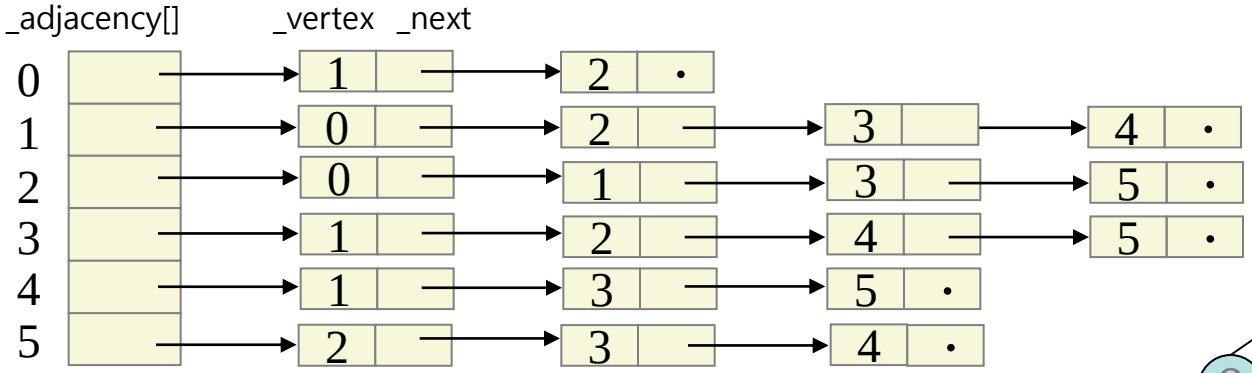
```
public class Graph {
    private int       _numOfVertices ;
    private Node[]  _adjacency ;
    ......
}

public class Node {
    private int       _vertex;
    private Node    _next;
    ....
}
```

Private Attributes for class Graph

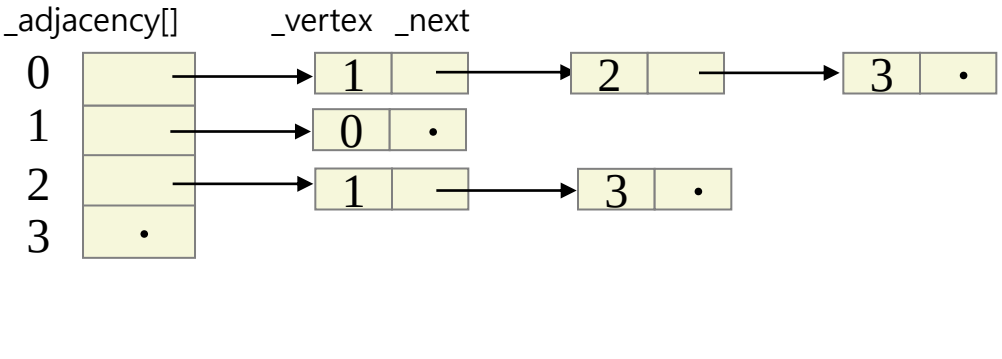
```
public class Graph {
    private Node[]  _adjacency ;
    private int     _numOfVertices ;
    private int     _numOfEdges ;
    ......  // some other attributes if necessary
```

Example

- Adjacency List for the undirected graph G_1.



- Adjacency List for the directed graph G_2.



- Complexities for an undirected graph with n vertices and e edges.
  - Space complexity: $O(n+e)$.
    - n head nodes (the array graph[]).
    - $2e$ list nodes.
  - Time complexity for examining all the edges: $O(n+e)$.
    - Examining all the vertices adjacent to vertex i : $(1+d_i)$
      - Here, d_i is the degree of vertex i .
      - Constant time is needed for examining the element graph[i].
    - Total examining time is:

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

Some Other Representations

- Sequential Representation
- Inverse Adjacency List
- Sparse Matrix Representation

Sequential representation of Adjacency lists

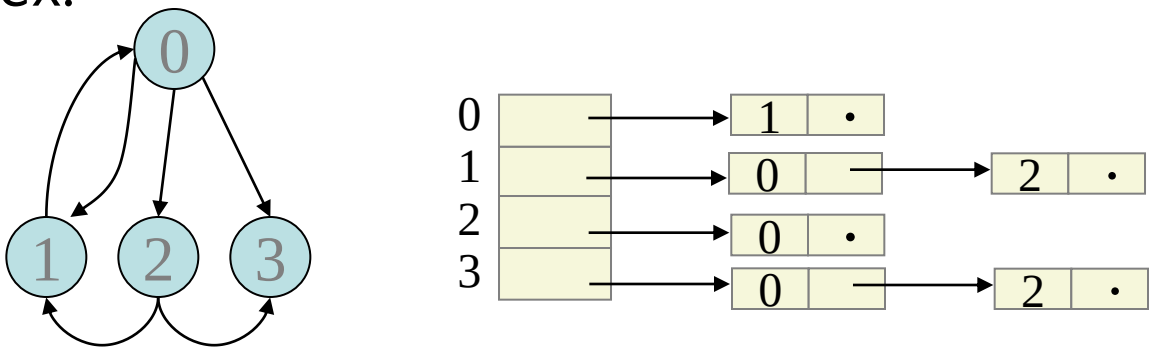
- Array size = n + 2e + 1
  (Here, 6 + 2·10 + 1 = 27)
- The vertices adjacent from vertex i are stored in node[i], · · ·, node[i+1]-1
  - The vertices adjacent from vertex 2 are stored in [13], [14], [15], [16])
    - node[2] = 13
    - node[3] – 1 = 17 – 1 = 16
- No pointer fields.

[0]	7	[7]	1
		[8]	2
[1]	9	[9]	0
		[10]	2
		[11]	3
		[12]	4
[2]	13	[13]	0
		[14]	1
		[15]	3
		[16]	5
[3]	17	[17]	1
		[18]	2
		[19]	4
		[20]	5
[4]	21	[21]	1
		[22]	3
		[23]	5
[5]	24	[24]	2
		[25]	3
		[26]	4
[6]	27		

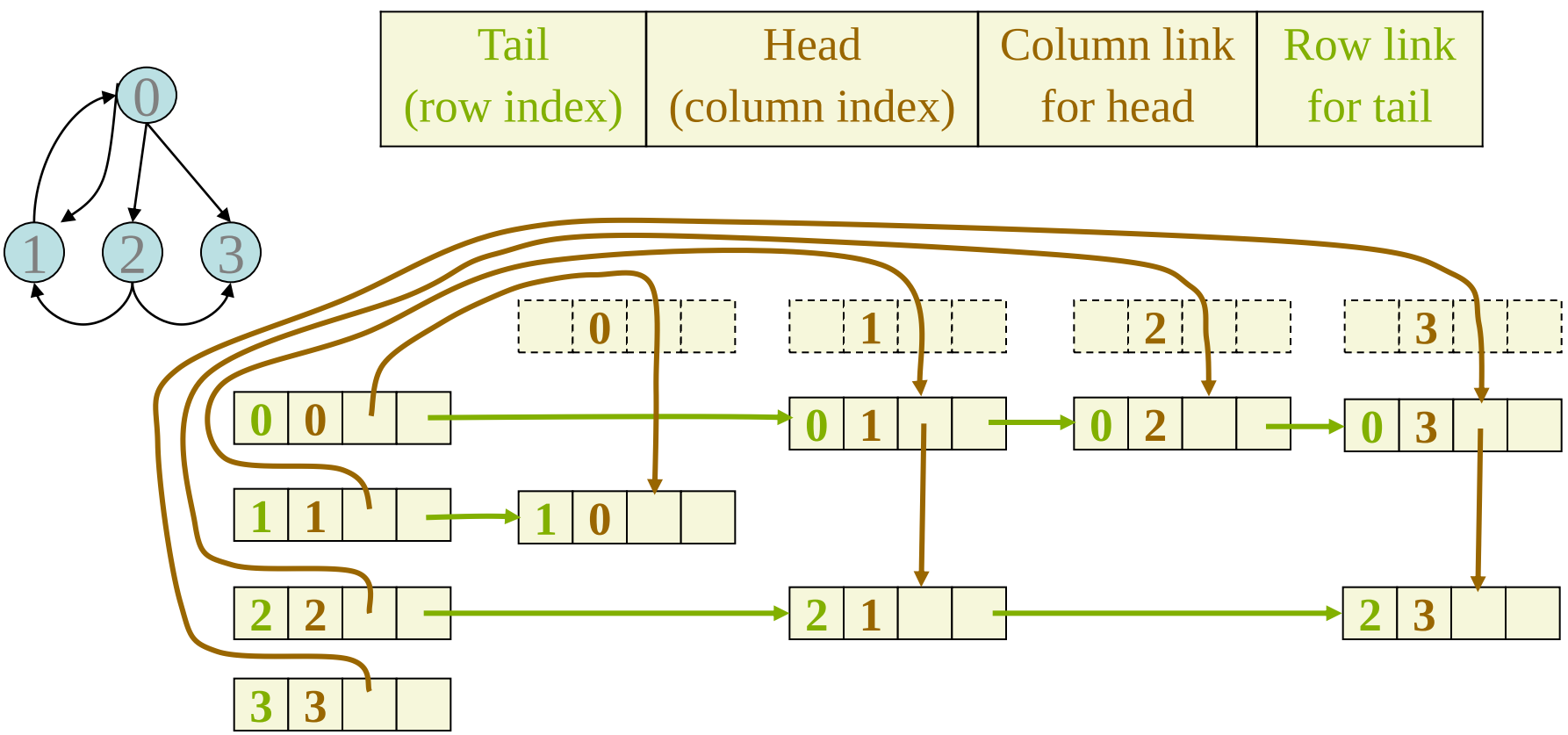
Inverse Adjacency Lists (for digraphs).

- Total # of edges in a digraph (by scanning a graph):
  - It can be determined in time $O(n + e)$. (By counting the out-degree of each vertex.)
- But, how can the in-degree be determined?

  Not so simple!
- Use Inverse adjacency lists.
  - Each list contains a node for each vertex adjacent to the vertex.



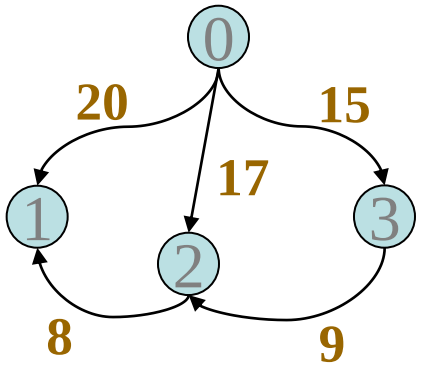
- Sparse Matrix representation of a graph
  - including both a normal adjacency lists and an inverse adjacency lists.
  - Node structure:



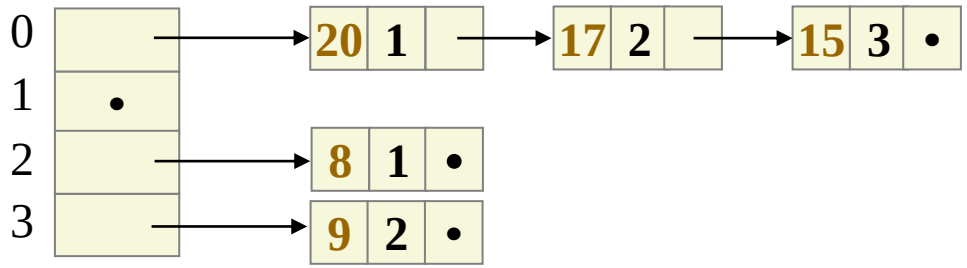
Networks
(Weighted Graphs)

Network

- A graph with weighted edges.
- Representation
  - For an adjacency matrix, A[i][j] stores the weight assigned to the edge (i, j).
  - For an adjacency list, an additional field is included for the weight to each node.



Adjacency lists



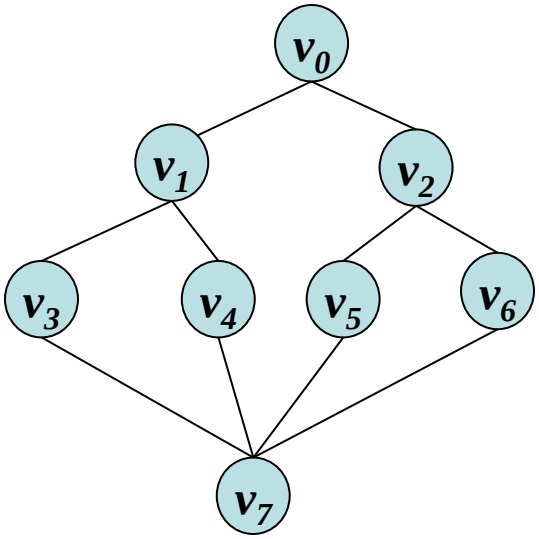
Adjacency Matrix

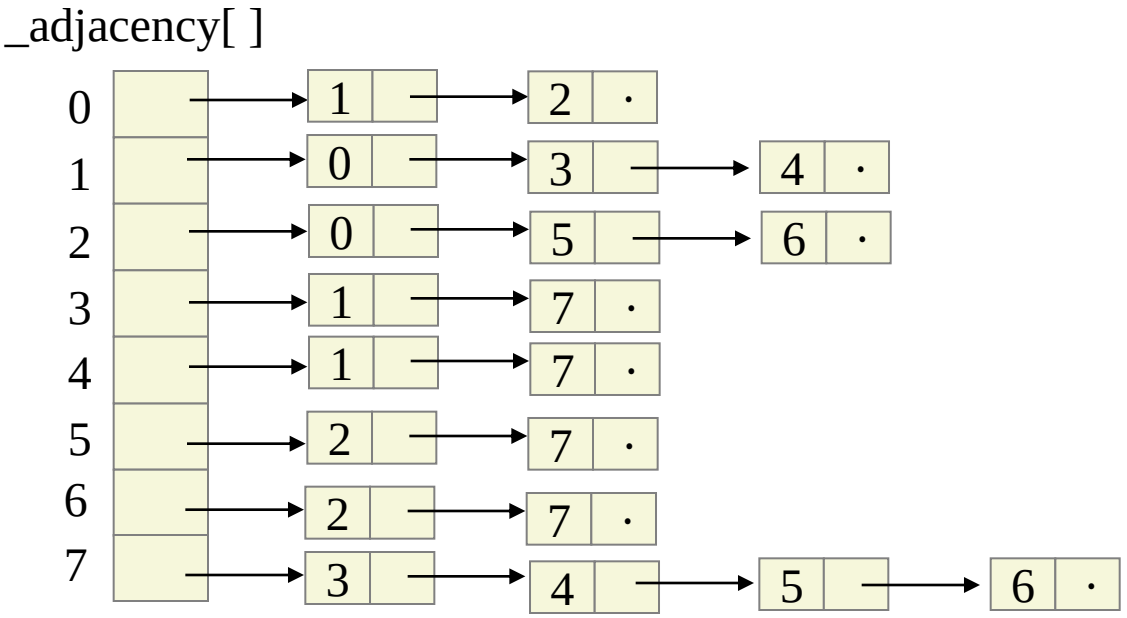
	0	1	2	3
0	0	20	17	15
1	0	0	0	0
2	0	8	0	0
3	0	0	9	0

Traversals on a Graph

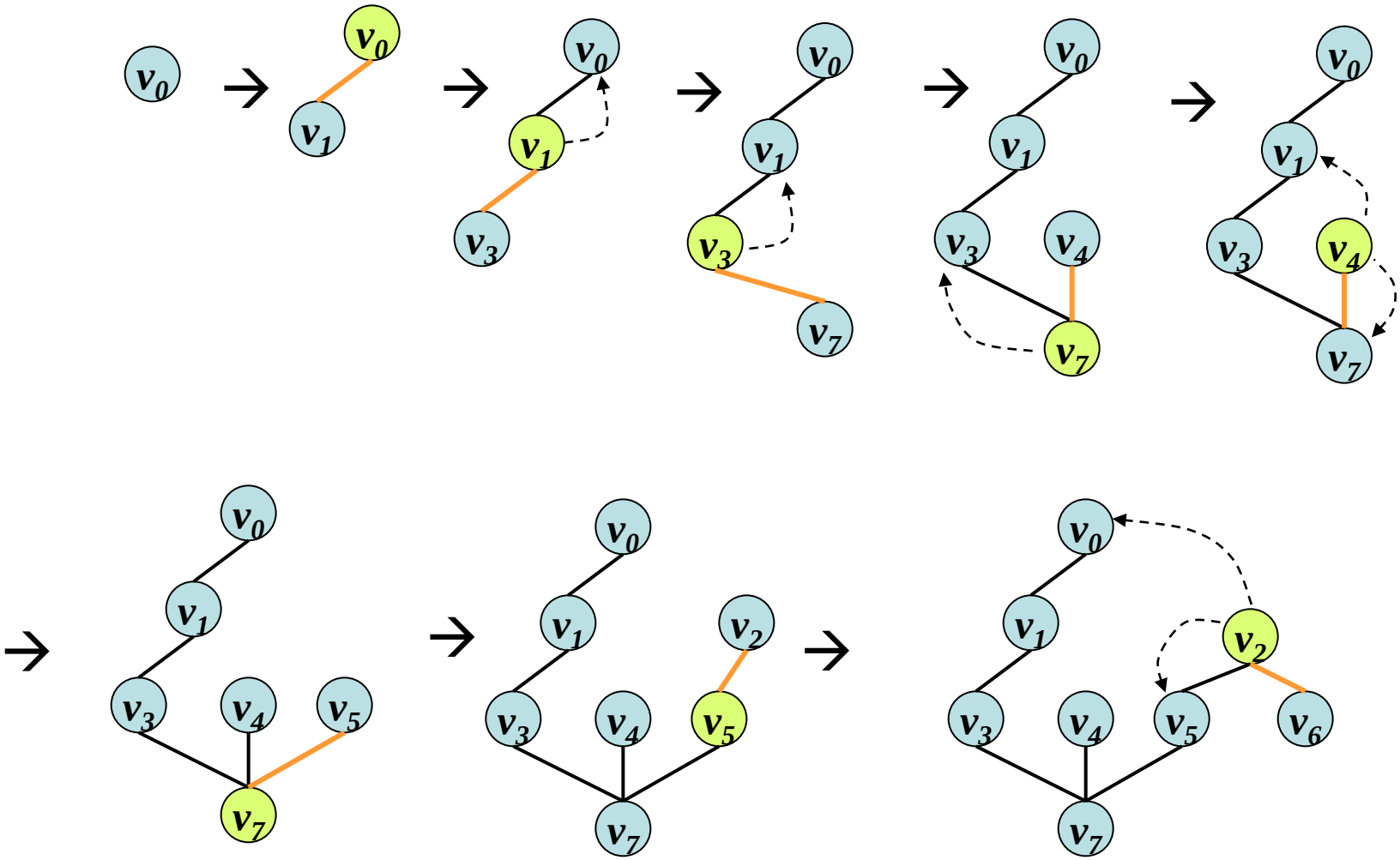
1. DFS (Depth-First Search)
2. BFS (Breadth-First Search)

Example Graph

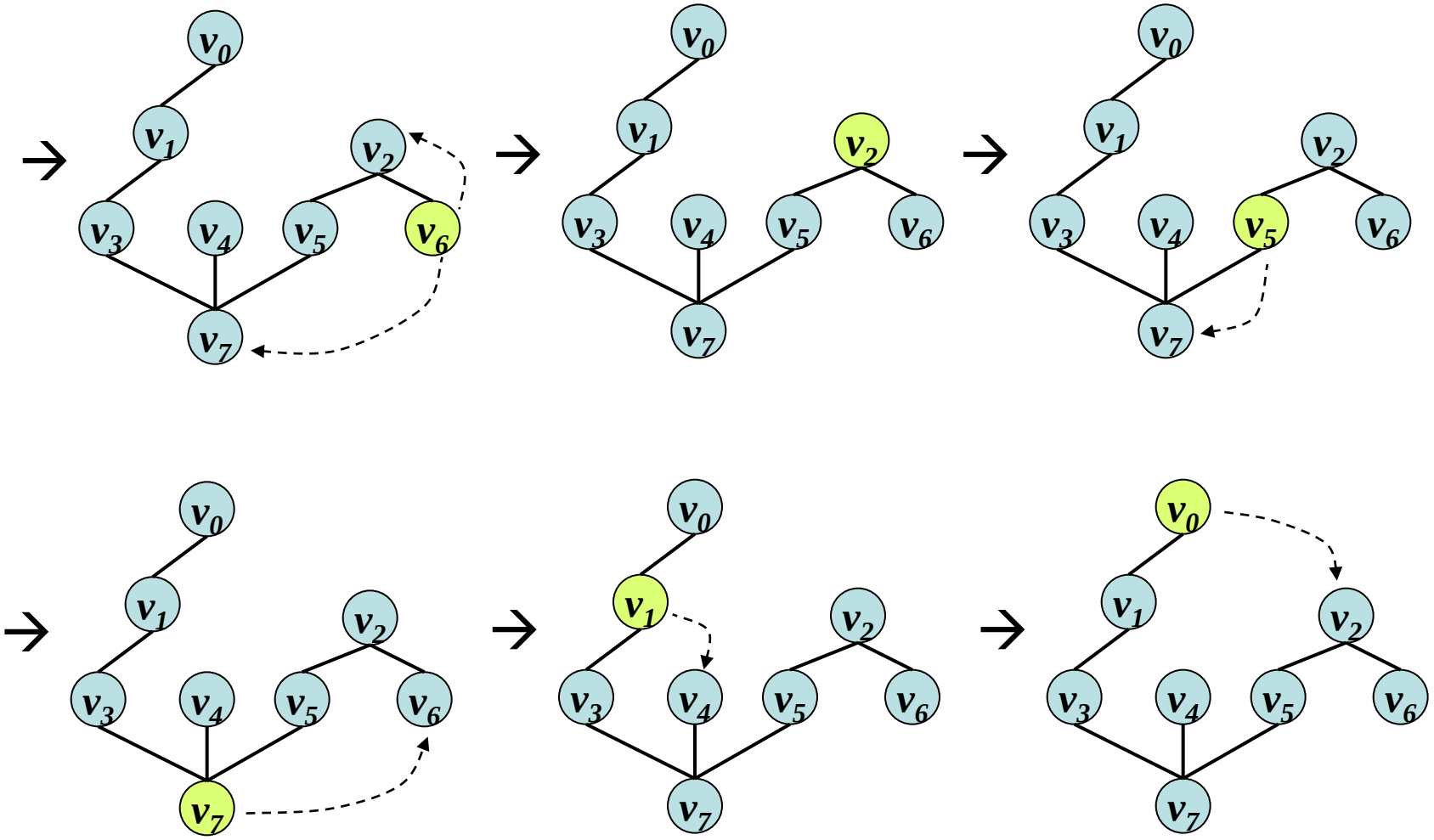




Depth First Search



Depth First Search (Cont'd)



- Algorithm for DFS using Adjacency List

```
/* _visited[]: it may be an attribute of class Graph */

private void  dfs (Vertex fromV)
{
    this._visited[fromV]=true ;
    this.visit(fromV) ;
    Node adjacentNode = this._adjacency[fromV] ;
    while (adjacentNode != null) {
        if ( !this._visited[adjacentNode.vertex()]) {
            dfs((Vertex)adjacentNode.vertex());
        }
        adjacentNode = adjacentNode.next() ;
    }
}
```

  - Time Complexity: O(*e*)

DFS using Adjacency Matrix

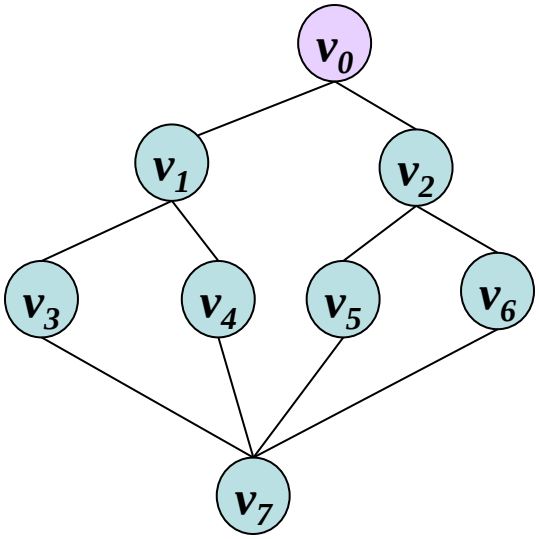
```
/* _visited[]: it may be an attribute of class Graph */

private void  dfs (Vertex fromV)
{
    Vertex toV ;
    this._visited[fromV] = true ;
    this.visit(fromV) ;
    for ( toV = 0 ; toV < this._numOfVertices ; toV++ ) {
        if ( this. _adjacency[fromV][toV].vertex() != 0 && !this._visited[toV] ) {
            dfs(toV) ;
        }
    }
}
```

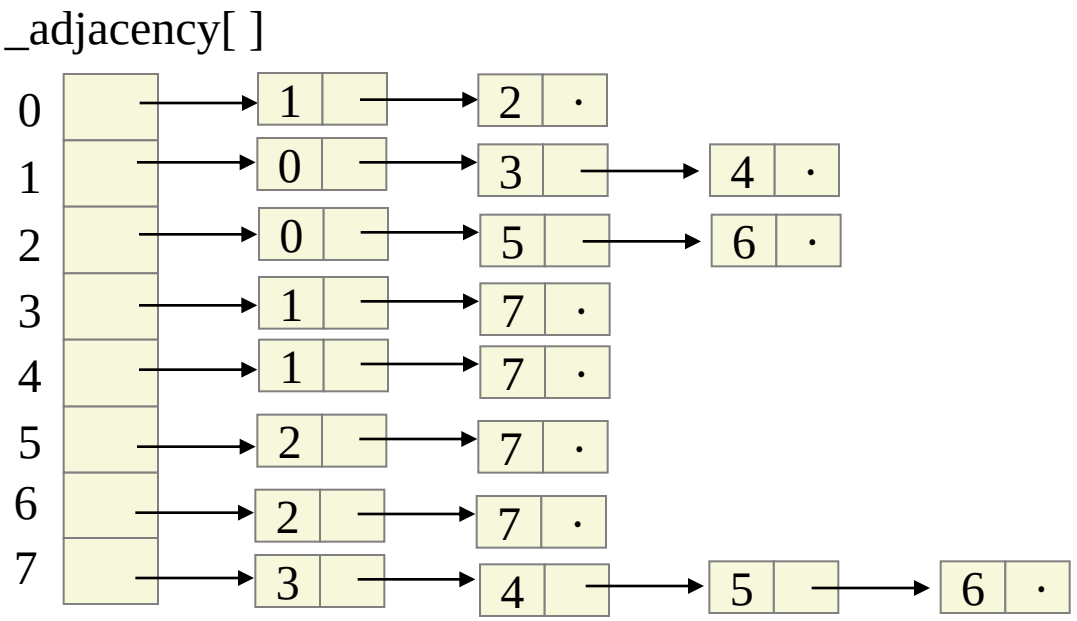
- Time Complexity: $O(n^2)$

Public function dfs()

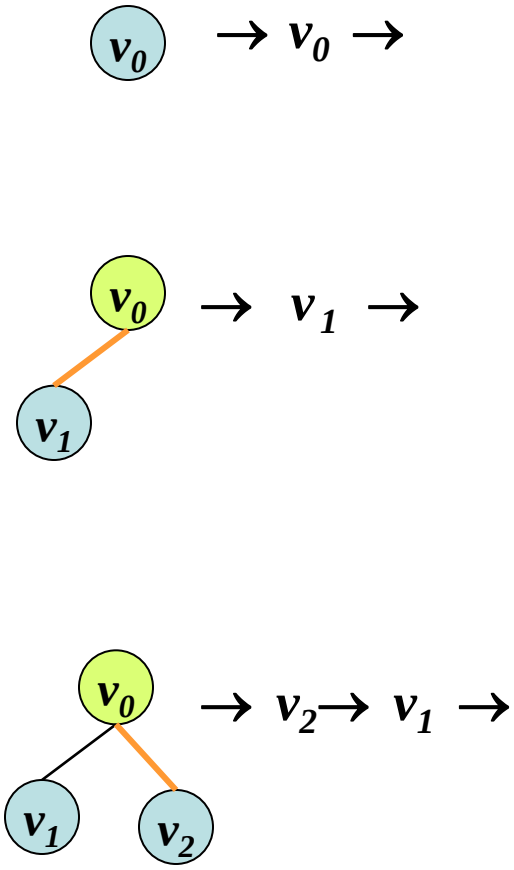
```
public void entireTraverseByDfs ()
{
      int  v;
      for ( v = 0 ; v < n ; v++ )
            this._visited[v] = false ;
      for ( v = 0 ; v < n ; v++ ) {
            if ( ! this._visited[v] )
                  dfs ( v ) ;
      }
}

private void  dfs (Vertex fromV)
{
      this._visited[fromV]=true ;
      this.visit(fromV) ;
      Node adjacentNode = this._adjacency[fromV] ;
      while (adjacentNode != null) {
          if ( !this._visited[adjacentNode.vertex()]) {
                dfs((Vertex)adjacentNode.vertex());
           }
            adjacentNode = adjacentNode.next() ;
      }
}
```

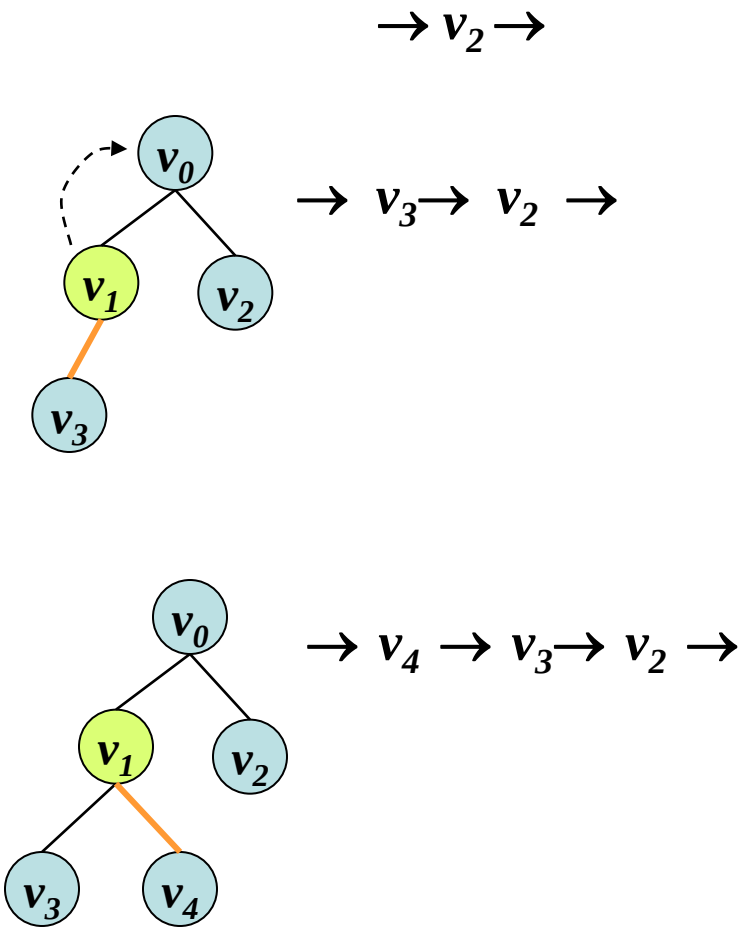
Breadth-First Search



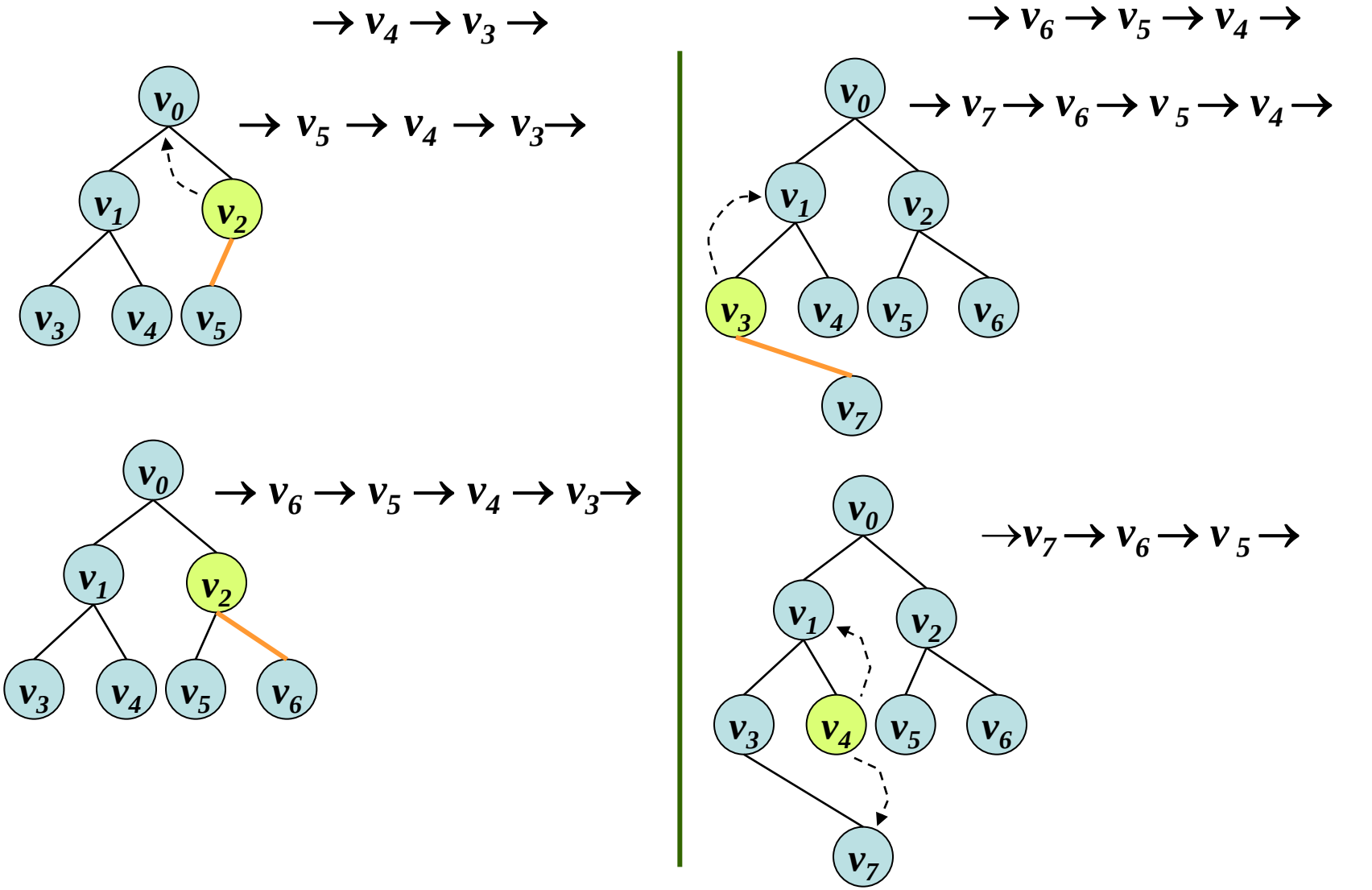


Breadth First Search [1]

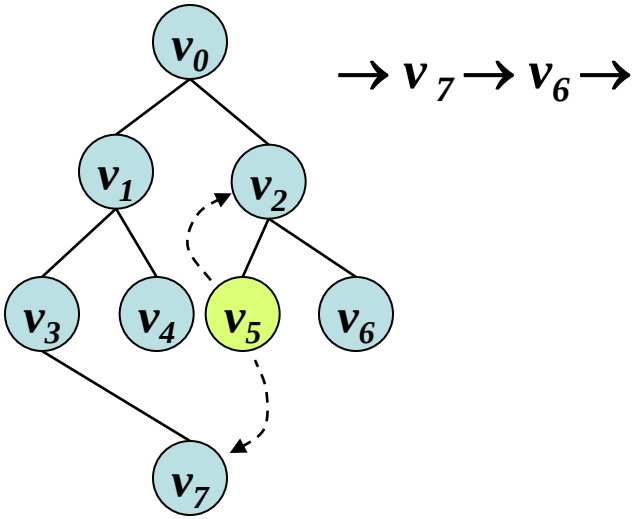


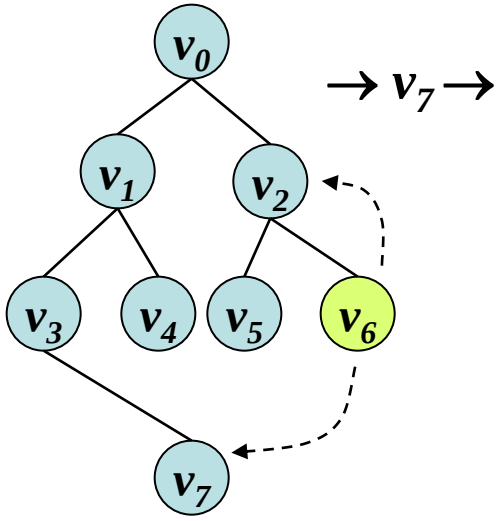


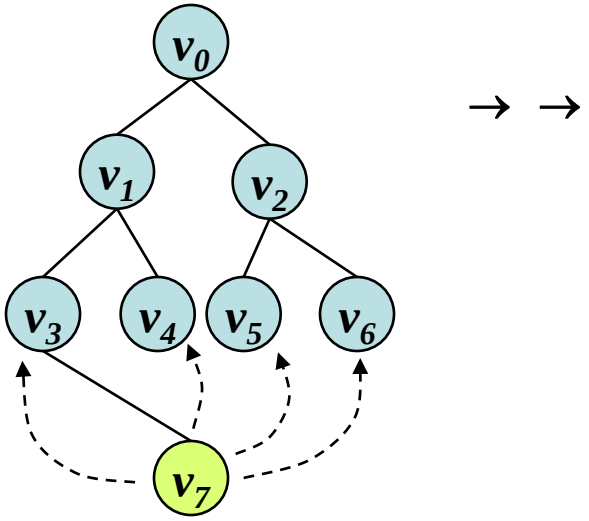
Breadth First Search [2]



Breadth First Search [3]







Here, the queue is empty.
So, BFS terminates.

- Program code for BFS [Cont'd]

```
private void bfs (Vertex startingV)
{
    Node adjacentNode ;
    Vertex fromV, toV ;
    Queue q = new Queue();
    this._visited[startingV]=true ;
    this.visit (startingV) ;
    q.add(startingV) ;
    while ( ! q.isEmpty() ) {
        fromV = q.remove();
        adjacentNode = this._adjacency[fromV] ;
        while (adjacentNode != null) {
            toV = adjacentNode.vertex() ;
            if ( !this._visited[toV] ) {
                this._visited[toV] = true ;
                visit(toV) ;
                q.add(toV) ;
            }
            adjacentNode = adjacentNode.next() ;
        }
    }
}
```

Remind !

- Public functions for class Queue<T>

```
public Queue() ;
public boolean add(T anElement) ;
public T max() ;
public T removeMax() ;
public int size() ;
public boolean isEmpty() ;
public boolean isFull() ;
```

- List all the Connected Components.
  - One component for each line.

```
public void  showConnectedComponents()
{
    Vertex        v ;
    for (v = 0 ; i < n ; i++)
        this._visited[v] = false ;
    for (v = 0 ; v < n ; v++) {
        if ( !this._visited[v] ) {
            dfs (v) ;
            System.out.println();
        }
    }
}
```

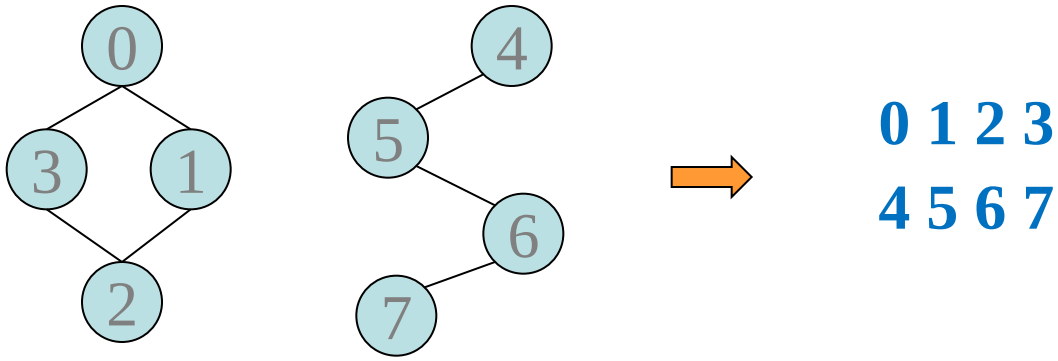
  - entireTraverseByDfs()와의 차이점은?

List all the Connected Components [Cont'd].

- Adjusting the DFS code.

  visit(v) ≡ {System.out.println(v.vertex()) ; }

- Example:



- Analysis of Time Complexity:
  - Adjacency list : O($n + e$)
  - Adjacency Matrix : O(n^2)

End of Graph Basics