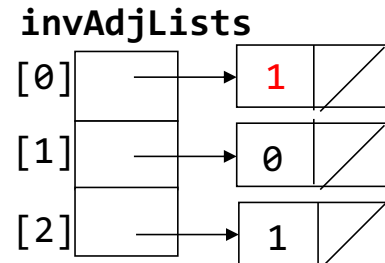
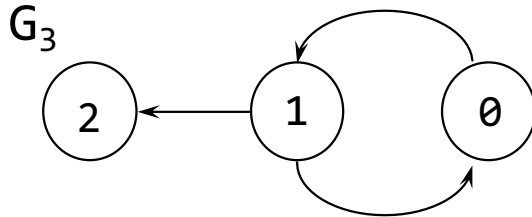

Lecture #21: Graph

School of Computer Science and Engineering
Kyungpook National University (KNU)

Woo-Jeoung Nam

Inverse Adjacency List (역 인접 리스트)

- 리스트가 표현하는 정점에 인접한 각 정점에 대해 하나의 노드
- In undirected graph, the degree of any vertex can be determined by counting number of nodes in its adjacency list
- Similarly, the out-degree of any vertex in directed graph
- To find **in-degree** of a vertex, we use additional list called Inverse Adjacency Lists
- It contains one list for each vertex i




Weighted Edges(가중치 간선)

- In many applications, the edges of a graph have weights assigned to them.
- These weights may represent
 - Distance from one vertex to another, or
 - Cost of going from one vertex to an adjacent vertex
- The representation of the graph needs to be modified to signify the weight of the edge
 - for adjacency matrix: weight instead of 1
 - for adjacency list: add additional weight field
- 그래프의 간선에 가중치 부여
- 인접행렬 : 행렬 엔트리에 $a[i][j]$ 의 가중치 정보 저장
- 인접리스트 : 노드 구조에 **weight** 필드를 추가
- 네트워크 : 가중치 간선을 가진 그래프

-	0	1	2	3
0	0.1	0	0.8	0
1	0	0	0.3	0
2	0	0.5	0	0
3	0	0	0	1

Basic Graph Operations

- Breadth First Search (BFS)
 - Depth First Search (DFS)
 - Connected Components
 - Spanning Trees
 - Biconnected Components
- 

The Graph Traversal Problem

- Given an undirected graph $G(V, E)$ and a vertex v in $V(G)$, we want to visit all vertices in G that are reachable from v
- Two choices to walk (or traverse) the graph
 - Ex) 지구상에 존재하는 모든 친구 관계를 그래프로 표현한 후 Ash와 Vanessa 사이에 존재하는 경로를 찾는 경우
 - BFS (Breath First Search)
 - Ash와 가까운 관계부터 탐색
 - DFS (Depth First Search)
 - 모든 친구 관계를 다 살펴봐야 할수도 있다

Breadth-First Search (BFS, 너비우선탐색)

- Given a graph $G = (V, E)$, and a source vertex s , BFS searches the edges of G to “discover” every vertex that is reachable from s
 - 루트 노드(혹은 다른 임의의 노드)에서 시작해서 인접한 노드를 먼저 탐색하는 방법
 - 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법
- It computes the distance from s to each reachable vertex
 - 깊게(deep) 탐색하기 전에 넓게(wide) 탐색하는 것
- During the search, a “breadth-first tree” with root s is produced
 - Thus, a new tree is built for each different source
- For any vertex v reachable from s , the simple path in the tree from s to v is a “shortest path”
- BFS works both on undirected and directed graphs



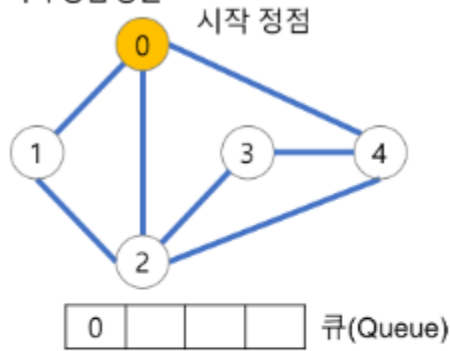
Breadth-First Search (BFS, 너비우선탐색)

- 직관적이지 않은 면이 있다.
- BFS는 시작 노드에서 시작해서 거리에 따라 단계별로 탐색한다고 볼 수 있다.
- BFS는 재귀적으로 동작하지 않는다.
- 그래프 탐색의 경우 어떤 노드를 방문했었는지 여부를 반드시 검사 해야 한다.
- 이를 검사하지 않을 경우 무한루프에 빠질 위험이 있다.
- BFS는 방문한 노드들을 차례로 저장한 후 꺼낼 수 있는 자료 구조인 큐(Queue)를 사용한다.
- 즉, 선입선출(FIFO) 원칙으로 탐색일반적으로 큐를 이용해서 반복적 형태로 구현하는 것이 가장 잘 동작한다.
- 'Prim', 'Dijkstra' 알고리즘과 유사하다.

Breadth-First Search (BFS, 너비우선탐색)

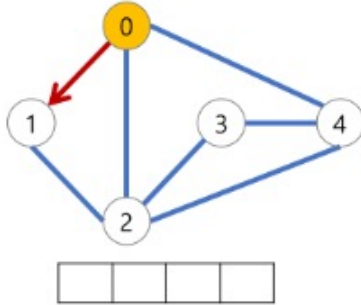
- a 노드(시작 노드)를 방문한다. (방문한 노드 체크)
 - 큐에 방문된 노드를 삽입(enqueue)한다.
 - 초기 상태의 큐에는 시작 노드만이 저장
 - 즉, a 노드의 이웃 노드를 모두 방문한 다음에 이웃의 이웃들을 방문한다.
- 큐에서 꺼낸 노드와 인접한 노드들을 모두 차례로 방문한다.
 - 큐에서 꺼낸 노드를 방문한다.
 - 큐에서 꺼낸 노드와 인접한 노드들을 모두 방문한다.
 - 인접한 노드가 없다면 큐의 앞에서 노드를 꺼낸다(dequeue).
- 큐에 방문된 노드를 삽입(enqueue)한다.
 - 큐가 소진될 때까지 계속한다

(1) 시작점 방문

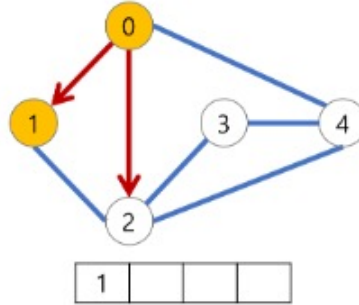


Breadth-First Search (BFS, 너비우선탐색)

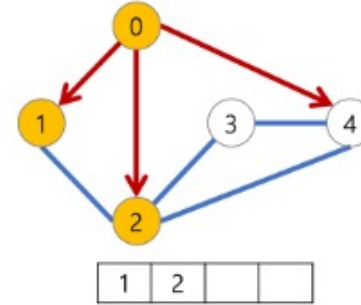
(2)



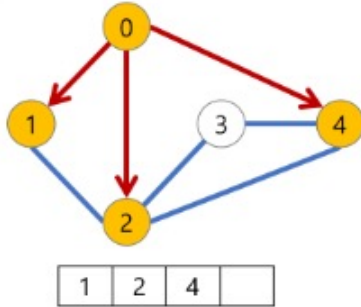
(3)



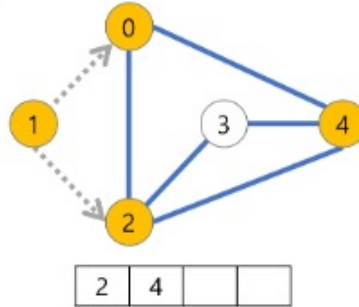
(4)



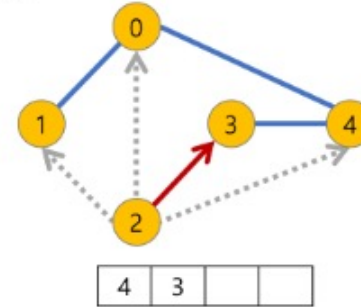
(5)



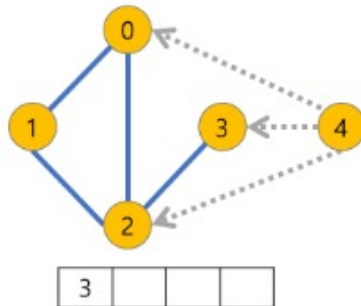
(6)



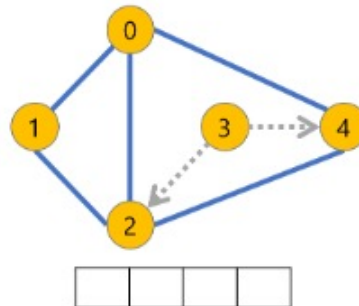
(7)



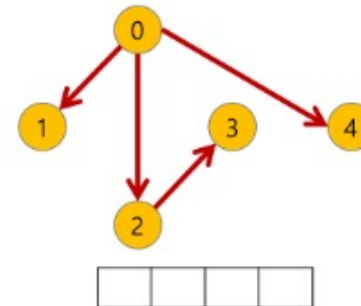
(8)



(9)

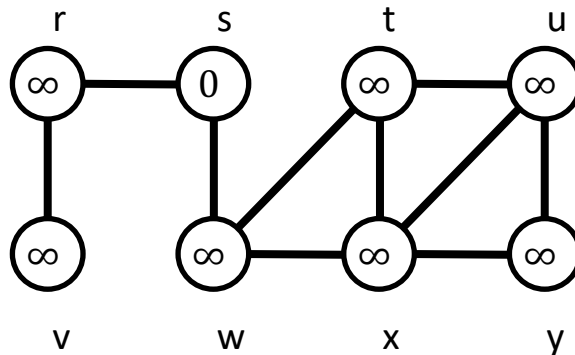


(10) 탐색 결과 (방문 순서: 0,1,2,4,3)



Breadth-First Search (BFS) – Example (교재)

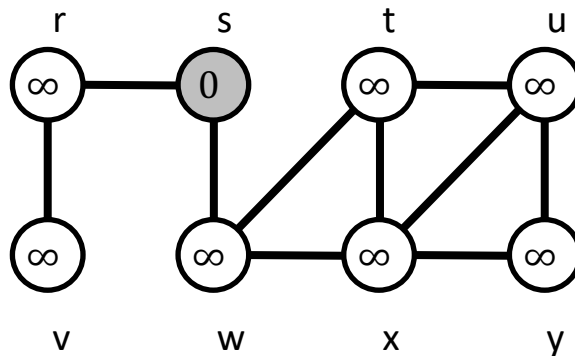
Source = s



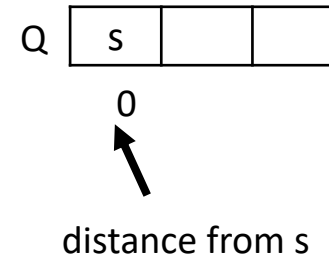
- Objective: Visit every vertex reachable from s and compute the distance
- For the search, we need to remember what nodes we visited, and what nodes we need to search for
- A queue will be used for this purpose

Breadth-First Search (BFS) – Example

Source = s

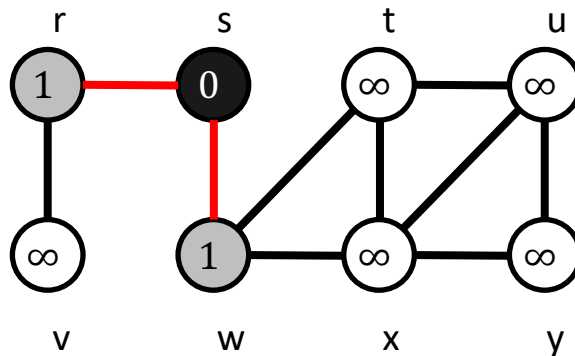


- We start BFS from vertex s
- Because we visited s , color it gray and insert s into queue Q

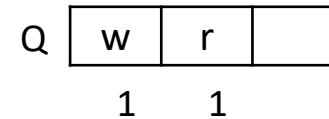


Breadth-First Search (BFS) – Example

Source = s

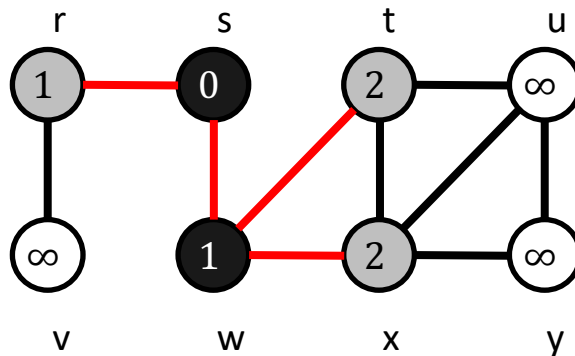


- Search the adjacent vertices to s, which are r and w
- Since we visited r and w mark them grey and add them to queue
- But before, the s vertex is done, dequeue it from Q



Breadth-First Search (BFS) – Example

Source = s



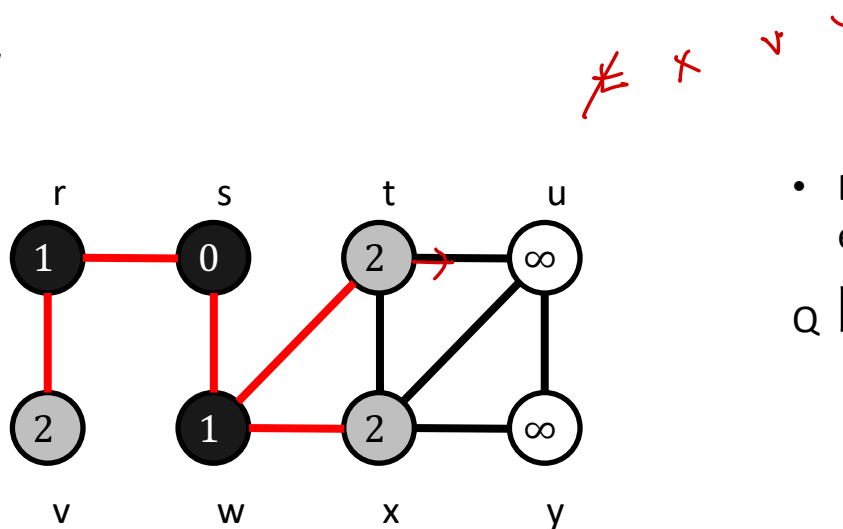
- Dequeue w and search for its adjacent vertices, which are t and x (and queue)

Q

r	t	x
1	2	2

Breadth-First Search (BFS) – Example

Source = s



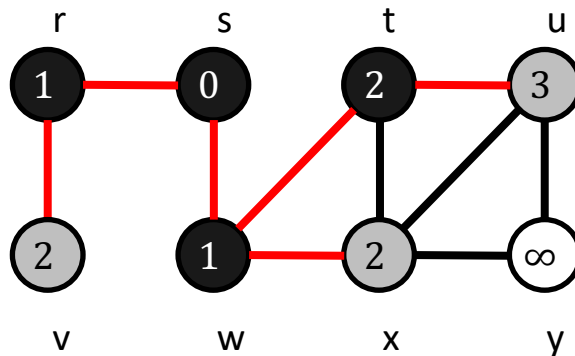
- Dequeue r and search for its adjacent vertices, which is v

Q

t	x	v
2	2	2

Breadth-First Search (BFS) – Example

Source = s



- Dequeue t and search for its adjacent vertices, which is x and u
- Since x is already in queue, only add u

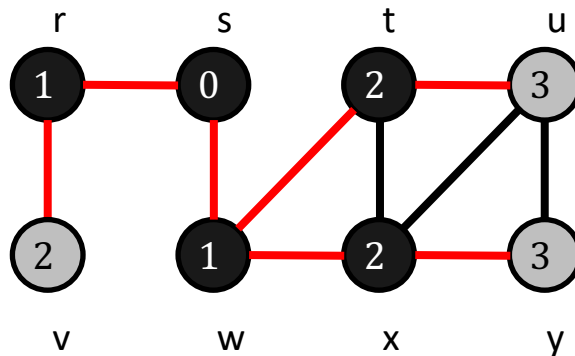
Q

x	v	u
2	2	3

~~x~~ x v

Breadth-First Search (BFS) – Example

Source = s



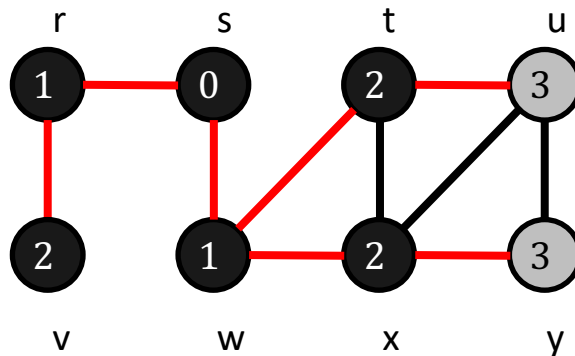
- Dequeue x and search for its adjacent vertices, which is u and y
- Since u is already in queue, only add y

Q

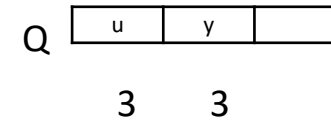
v	u	y
2	3	3

Breadth-First Search (BFS) – Example

Source = s

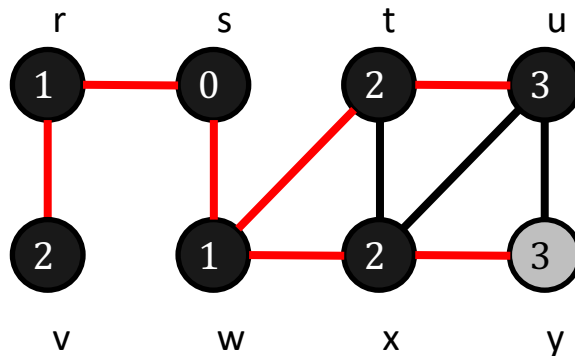


- Dequeue v and search for its adjacent vertices, which is none



Breadth-First Search (BFS) – Example

Source = s



- Dequeue u and search for its adjacent vertices, which is y (ignore)

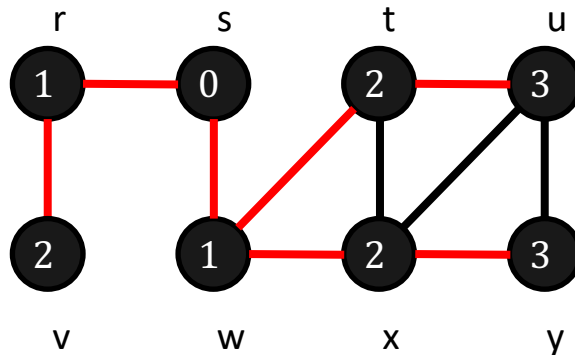
Q

y		
---	--	--

3

Breadth-First Search (BFS) – Example

Source = s



- Dequeue y and search for its adjacent vertices, which none

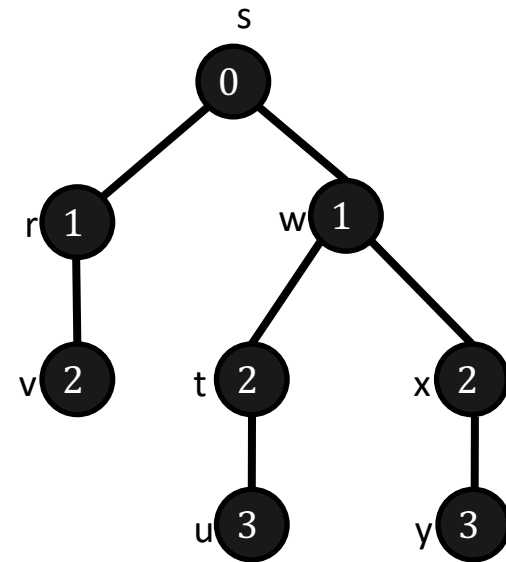
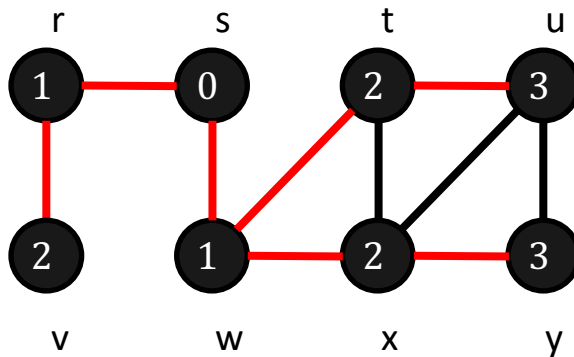
Q

--	--	--

Queue is empty, and so BFS terminates

Breadth-First Search (BFS) – Example

Source = s



* From the tree, the simple path from s to some vertex is the shortest path.

Breadth-First Search (BFS) – Pseudocode

처음 s는 Gray, 나머지는 white

다 방문했으면 black

start

```

BFS( $G, s$ )
1  for each vertex  $u \in G.V - \{s\}$ 
2     $u.color = WHITE$ 
3     $u.d = \infty$ 
4     $u.\pi = NIL$ 
5   $s.color = GRAY$ 
6   $s.d = 0$ 
7   $s.\pi = NIL$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11    $u = DEQUEUE(Q)$ 
12   for each  $v \in G.Adj[u]$ 
13     if  $v.color == WHITE$ 
14        $v.color = GRAY$ 
15        $v.d = u.d + 1$ 
16        $v.\pi = u$ 
17       ENQUEUE( $Q, v$ )
18    $u.color = BLACK$ 
    
```

시작 지점 표시

Initialize graph

Initialize source s and queue Q .
Distance from s to s is 0

If the adjacent vertexes of u are to be visited, dequeue u (color as black)

For each vertex v adjacent to u

- Color them grey
- Store their distance (+1)
- Set parent as u
- And insert v into queue

skipping black vertices
(already visited)

black

차이점 X

Breadth-First Search (BFS) – Analysis

- The input data is a graph $G = (V, E)$
- By the pseudocode, we can see that **each vertex** is queued only once, and thus dequeued once – $O(1)$ time
- So, the operation cost here is enqueue and dequeue, which is $O(V)$
- However, we need to search for adjacent vertices for each v , which takes $O(E)$
- The total time complexity is $O(V+E)$, so it is running in linear time! (in respect to the size of the adjacency list)

$O(V+E)$

선형 시간

$$\frac{O(V) + O(E) + O(V)}{= O(V+E)}$$

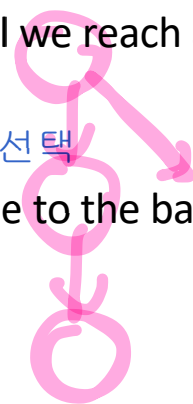
```

BFS( $G, s$ )
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
    
```

$O(E+V)$

Depth-First Search (DFS)

- Given a graph $G = (V, E)$, and a source vertex s , DFS searches the edges of G to “discover” every vertex that is reachable from s (exactly the same as BFS)
 - 하나의 정점으로부터 시작하여 차례대로 모든 정점들을 한 번씩 방문하는 것
- The search is a little more complex than BFS but still straightforward, which involves back tracking
- As its name, it searches deeper instead of broader
 - 넓게(wide) 탐색하기 전에 깊게(deep) 탐색하는 것
- For each vertex v adjacent to s , we keep on searching for adjacent vertices that are 1 distance away from v , which is repeated until we reach to a dead end (no more child vertices)
 - 미로를 탐색할 때 한 방향으로 갈 수 있을 때까지 계속 가다가 더 이상 갈 수 없게 되면 다시 가장 가까운 갈림길로 돌아와서 이곳으로부터 다른 방향으로 다시 탐색을 진행하는 방법과 유사하다.
- In case we reach a dead end, we back track until we reach a vertex v of which some adjacent vertices are not visited
 - 모든 노드를 방문하고자 하는 경우에 이 방법 선택
- DFS also produces a tree but different to BFS due to the back track procedure



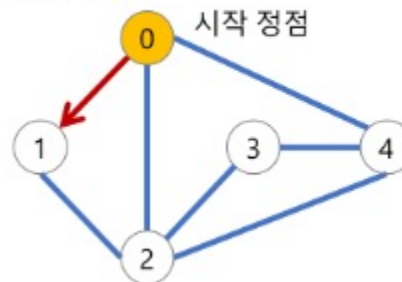
Depth-First Search (DFS)

- 자기 자신을 호출하는 순환 알고리즘의 형태 를 가지고 있다.
- 전위 순회(Pre-Order Traversals)를 포함한 다른 형태의 트리 순회는 모두 DFS의 한 종류이다.
- 그래프 탐색의 경우 어떤 노드를 방문했었는지 여부를 반드시 검사 해야 한다는 것이다.
- 이를 검사하지 않을 경우 무한루프에 빠질 위험이 있다

Depth-First Search (DFS)

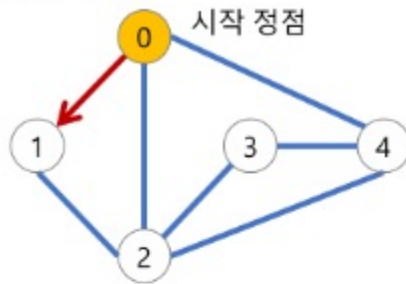
- a 노드(시작 노드)를 방문한다.방문한 노드는 방문했다고 표시한다.a와 인접한 노드들을 차례로 순회한다.
 - a와 인접한 노드가 없다면 종료한다.
- a와 이웃한 노드 b를 방문했다면, a와 인접한 또 다른 노드를 방문하기 전에 b의 이웃 노드들을 전부 방문해야 한다.
 - b를 시작 정점으로 DFS를 다시 시작하여 b의 이웃 노드들을 방문한다.
- b의 분기를 전부 완벽하게 탐색했다면 다시 a에 인접한 정점들 중에서 아직 방문이 안 된 정점을 찾는다.
 - 즉, b의 분기를 전부 완벽하게 탐색한 뒤에야 a의 다른 이웃 노드를 방문할 수 있다는 뜻이다.
 - 아직 방문이 안 된 정점이 없으면 종료한다.
 - 있으면 다시 그 정점을 시작 정점으로 DFS를 시작한다

(1) 정점 1 방문

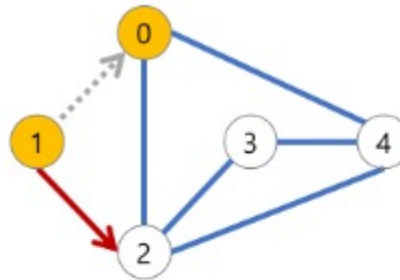


Depth-First Search (DFS)

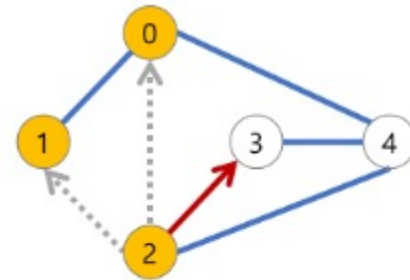
(1) 정점 1 방문



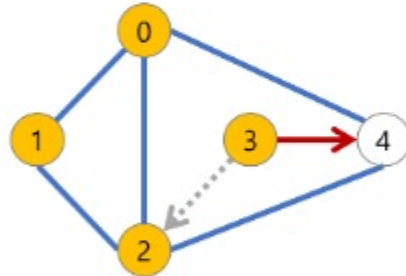
(2) 정점 2 방문



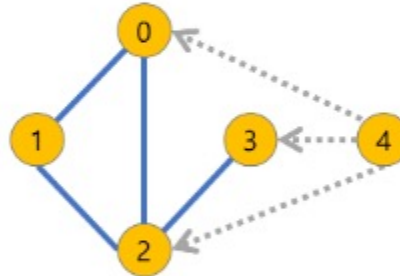
(3) 정점 3 방문



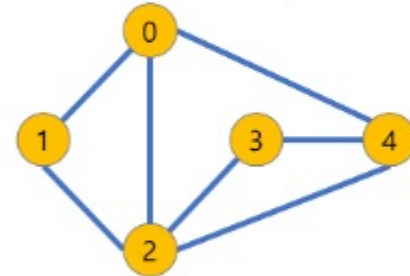
(4) 정점 4 방문



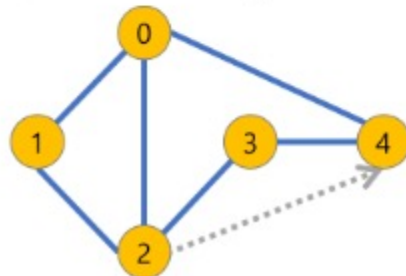
(5) 정점 3으로 backtracking
(다시 돌아와서 탐색하지 않은 정점이 있는지 확인)



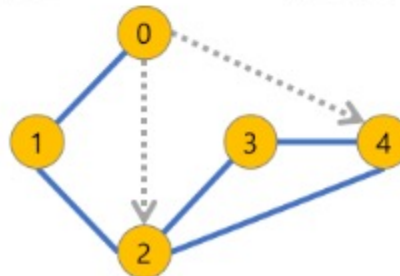
(6) 정점 2로 backtracking



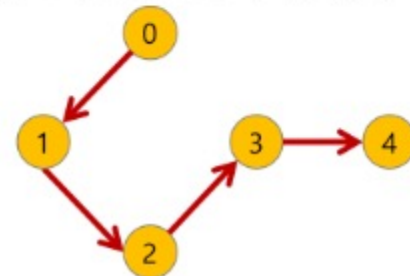
(7) 정점 1로 backtracking



(8) 정점 0으로 backtracking (탐색 종료)



(9) 탐색 결과 (방문 순서: 0, 1, 2, 3, 4)



Depth-First Search (DFS) – Example (교재)

- Given a graph $G = (V, E)$, start DFS at node u
 - (발견시간/종료시간)

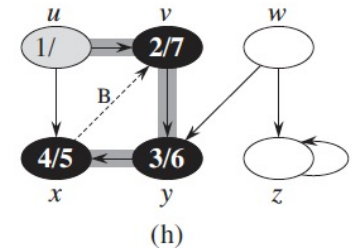
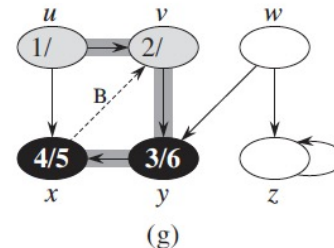
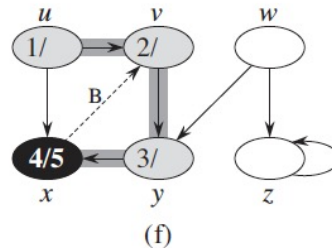
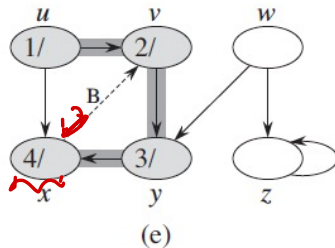
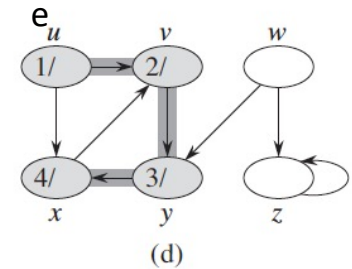
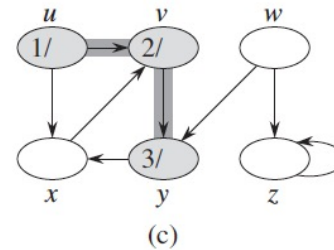
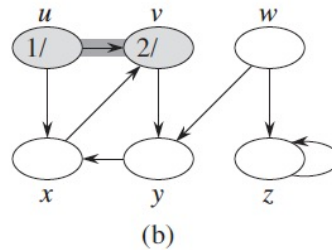
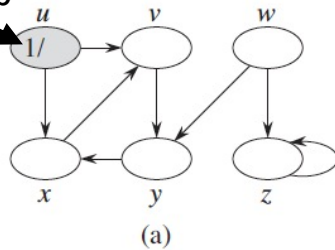
Color codes

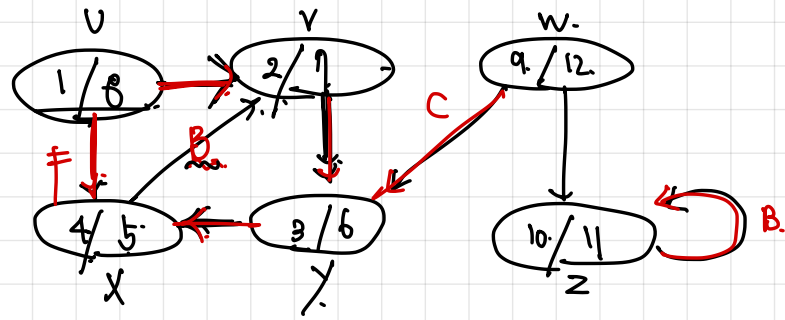
- Grey: visited
- Black: finished

Line types

- Grey: visited
- Dotted: not included in tree

timestamp





Depth-First Search (DFS) – Example (pg 605)

- Given a graph $G = (V, E)$, start DFS at node u
 - B: Back(역행)
 - C: cross(교차)
 - F: forward(순행)

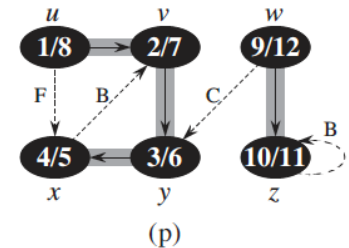
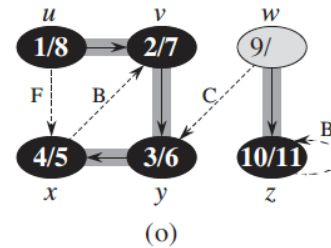
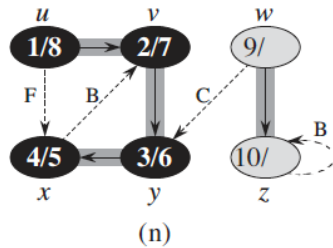
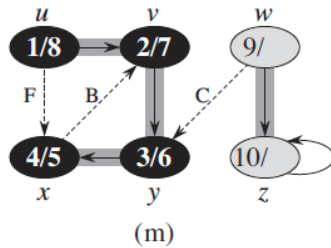
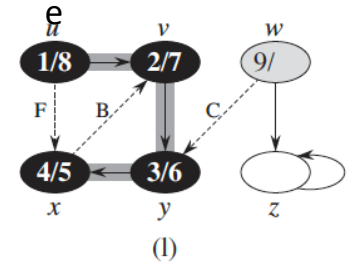
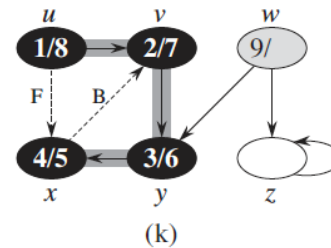
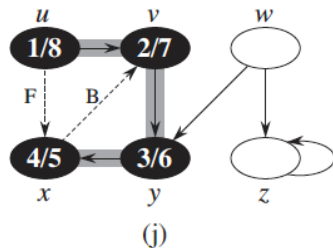
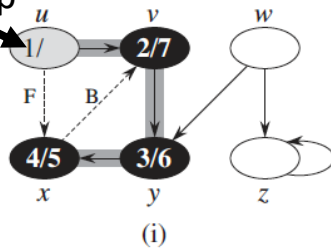
Color codes

- Grey: visited
- Black: finished

Line types

- Grey: visited
- Dotted: not included in tre

timestamp



Depth-First Search (DFS) – Pseudocode

DFS(G)

```

1  for each vertex  $u \in G.V$ 
2     $u.color = \text{WHITE}$ 
3     $u.\pi = \text{NIL}$ 
4   $time = 0$ 
5  for each vertex  $u \in G.V$ 
6    if  $u.color == \text{WHITE}$ 
7      DFS-VISIT( $G, u$ )
    
```

Initialize graph

Perform DFS on each vertex u (Important! Different to BFS)

DFS-VISIT(G, u)

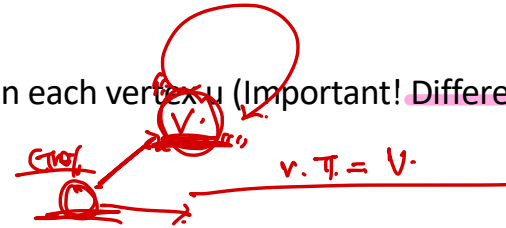
```

1   $time = time + 1$ 
2   $u.d = time$ 
3   $u.color = \text{GRAY}$ 
4  for each  $v \in G.Adj[u]$ 
5    if  $v.color == \text{WHITE}$ 
6       $v.\pi = u$ 
7      DFS-VISIT( $G, v$ )
8   $u.color = \text{BLACK}$ 
9   $time = time + 1$ 
10  $u.f = time$ 
    
```

As a event (visit) has happened increment time, and color the visited vertex as grey

For every vertex v adjacent to u , perform DFS. Skip any black and grey colored vertices

For exploring is finished for vertex u , color u as black and increment time for terminating the exploration for u



$v.color = \text{black}$
 $time + 1$
 $u.f = time$
 black을 할
 끝난 시간 저장

Depth-First Search (DFS) – Analysis

DFS(G)

```

1  for each vertex  $u \in G.V$ 
2     $u.color = WHITE$ 
3     $u.\pi = NIL$ 
4   $time = 0$ 
5  for each vertex  $u \in G.V$ 
6    if  $u.color == WHITE$ 
7      DFS-VISIT( $G, u$ )
    
```

DFS-VISIT(G, u)

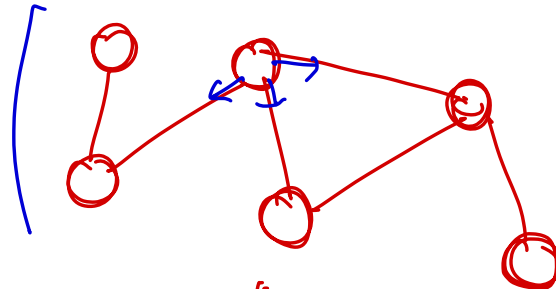
```

1   $time = time + 1$ 
2   $u.d = time$ 
3   $u.color \neq GRAY$ 
4  for each  $v \in G.Adj[u]$ 
5    if  $v.color == WHITE$ 
6       $v.\pi = u$ 
7      DFS-VISIT( $G, v$ )
8   $u.color = BLACK$ 
9   $time = time + 1$ 
10  $u.f = time$ 
    
```

$\theta(V)$

$\theta(V)$

$\theta(|Adj[v]|)$



이런식 방문할 때

$\theta(V+E)$

이것은 Vertex 가 차례 방문

$\theta(E)$

$$\sum_{v \in V} |Adj[v]| = \theta(E)$$

Total time complexity of DFS is $\theta(E + V)$
(Same as BFS)

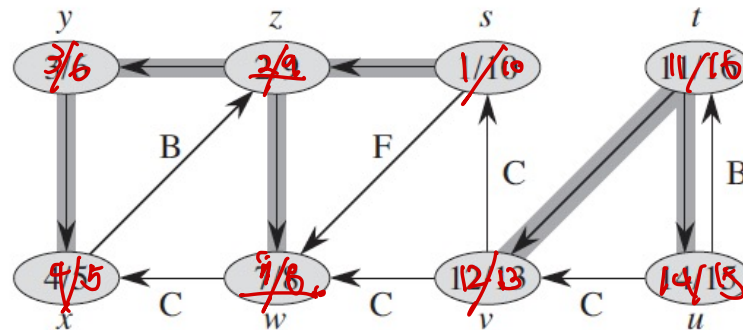
이제 방문 시간

Properties of Depth-First Search (DFS)

- The DFS can tell us valuable information about the **structure** of the graph
- First, DFS outputs a set of trees to form a **forest** (BFS outputs a single tree)
- For $V = \{s, t, \dots\}$

DFS는 forest를 생성한다

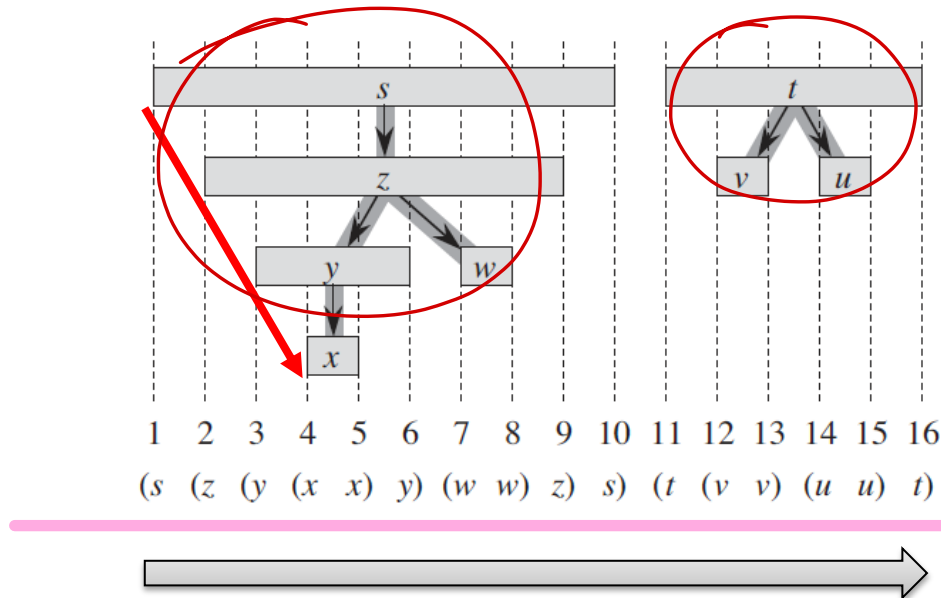
BFS는 하나의 tree를



DFS는 forest를 생성한다
BFS는 single tree

Properties of Depth-First Search (DFS)

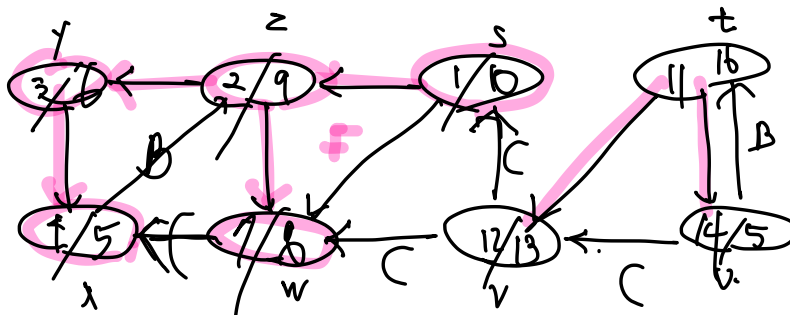
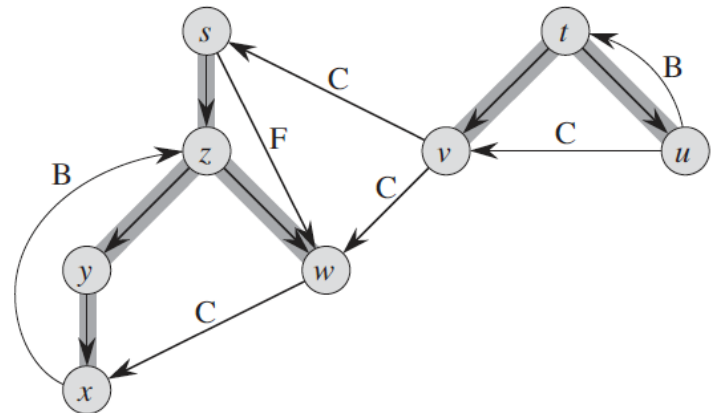
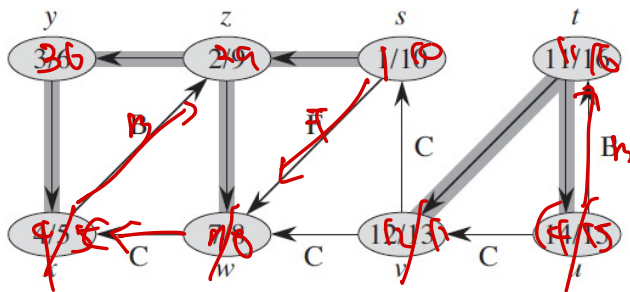
- The DFS can tell us valuable information about the **structure** of the graph
- The timestamp for visiting each vertex can be presented in a string of parenthesis structure
- 각 정점의 발견시간과 종료시간에 대한 구간을 보여준 괄호
- 각 사각형은 해당 정점의 발견 시간과 종료시간에 의해 주어지는 구간에 걸쳐있음



Properties of Depth-First Search (DFS)

- The DFS can tell us valuable information about the **structure** of the graph
- Second, we can classify the type of a vertex
- For example, we can determine whether a graph is an acyclic graph during the DFS
- 모든 트리와 깊이 우선 트리에서 아래로 향하는 순행간선과 자손에서 조상으로 향하는 모든 역행간선과 함께 다시 그린 그래프

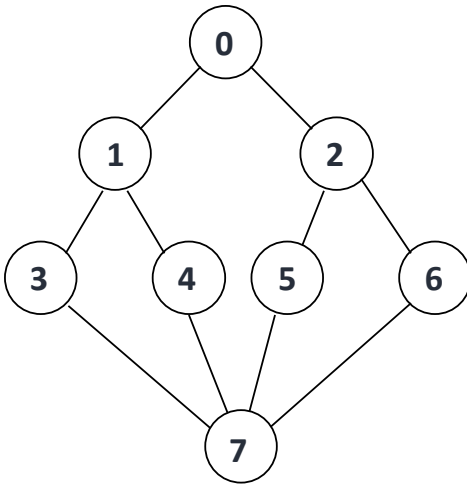
① connected components
② cycle detection



- by calling DFS(0) or BFS(0) and then determining the maximum number of connected components of a graph. This can be done by calling DFS(0) or BFS(0) where v is a vertex in the graph.

Connected Components - Example

- Let's use ~~DFS~~ to search for connected components

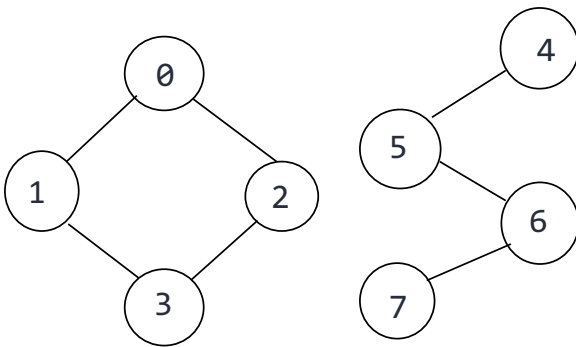


Solve for DFS(G), where $V = \{0,1,2,3,4,5,6,7\}$

Sequence = [0, 1, 3, 7, 4, 5, 2, 6]

Connected Components - Example

- Let's use DFS to search for connected components



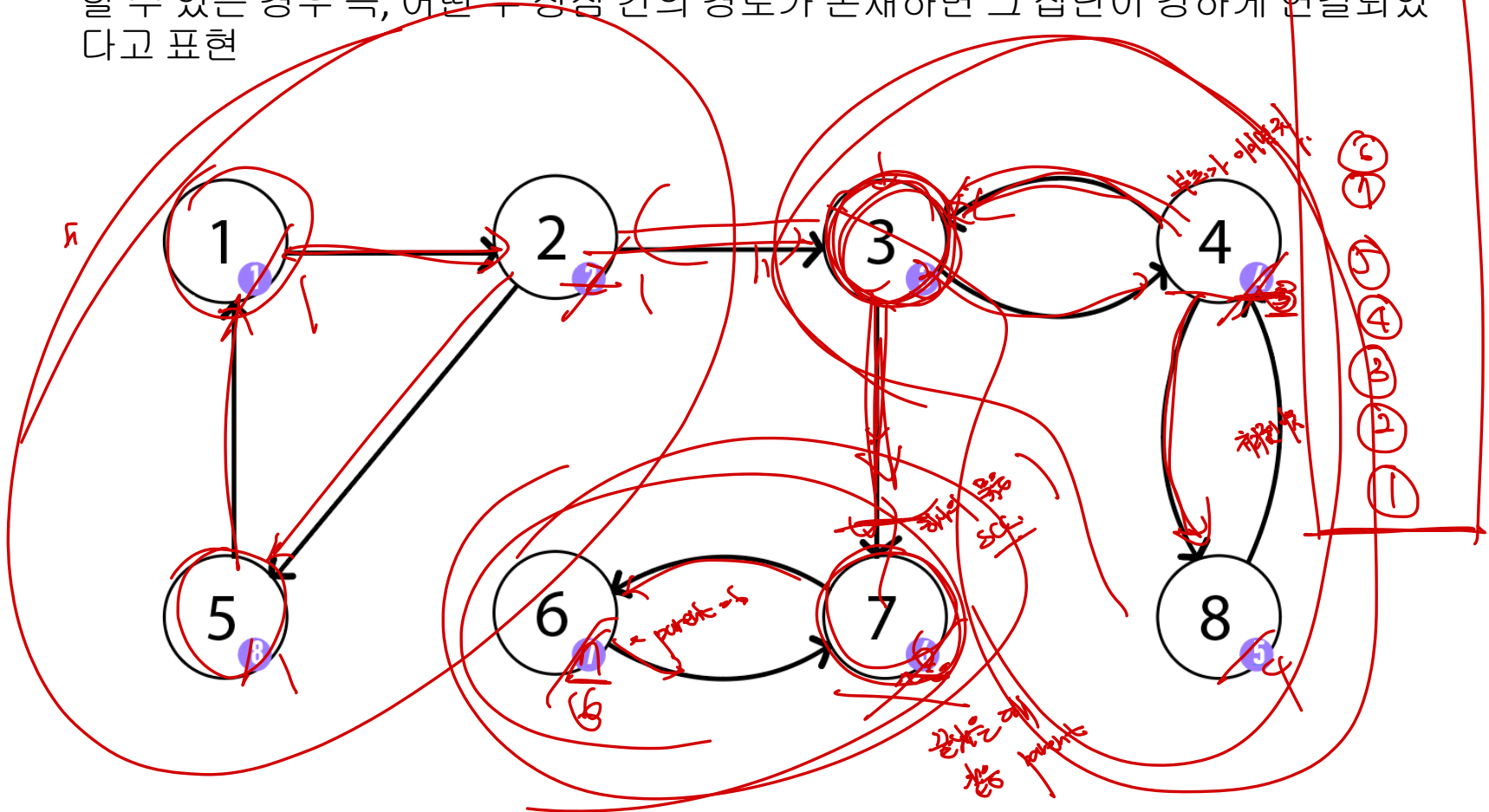
Solve for DFS(G), where $V = \{0,1,2,3,4,5,6,7\}$

Sequence 1 = [0, 1, 3, 2]

Sequence 2 = [4, 5, 6, 7]

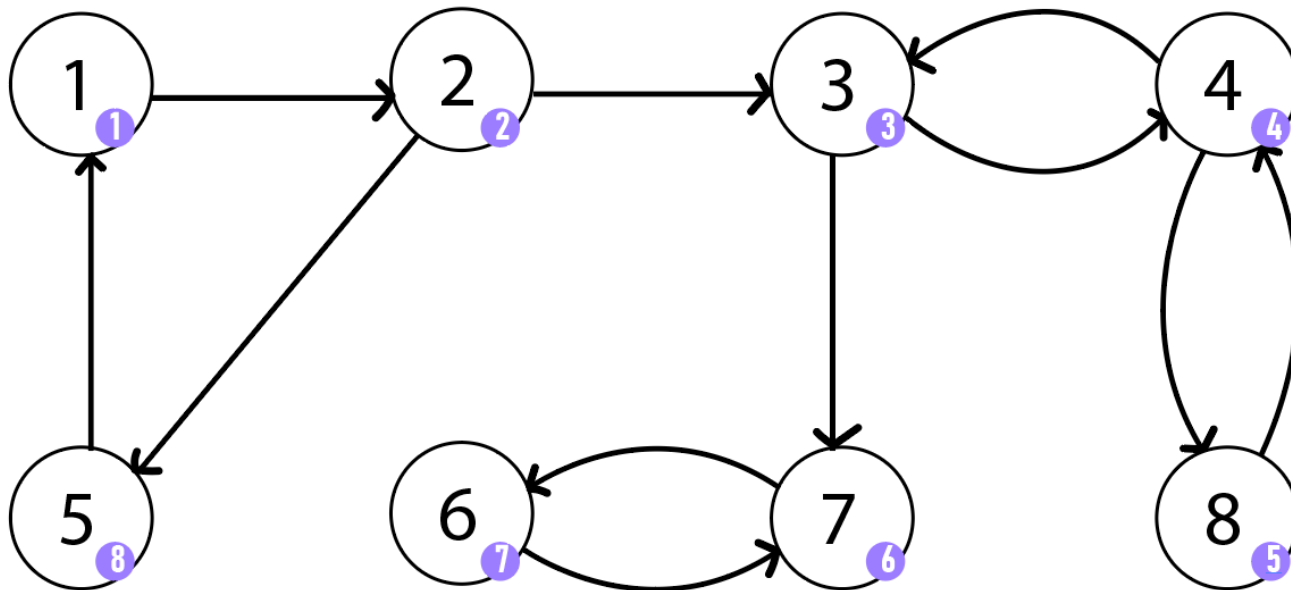
Connected Components – directed graph

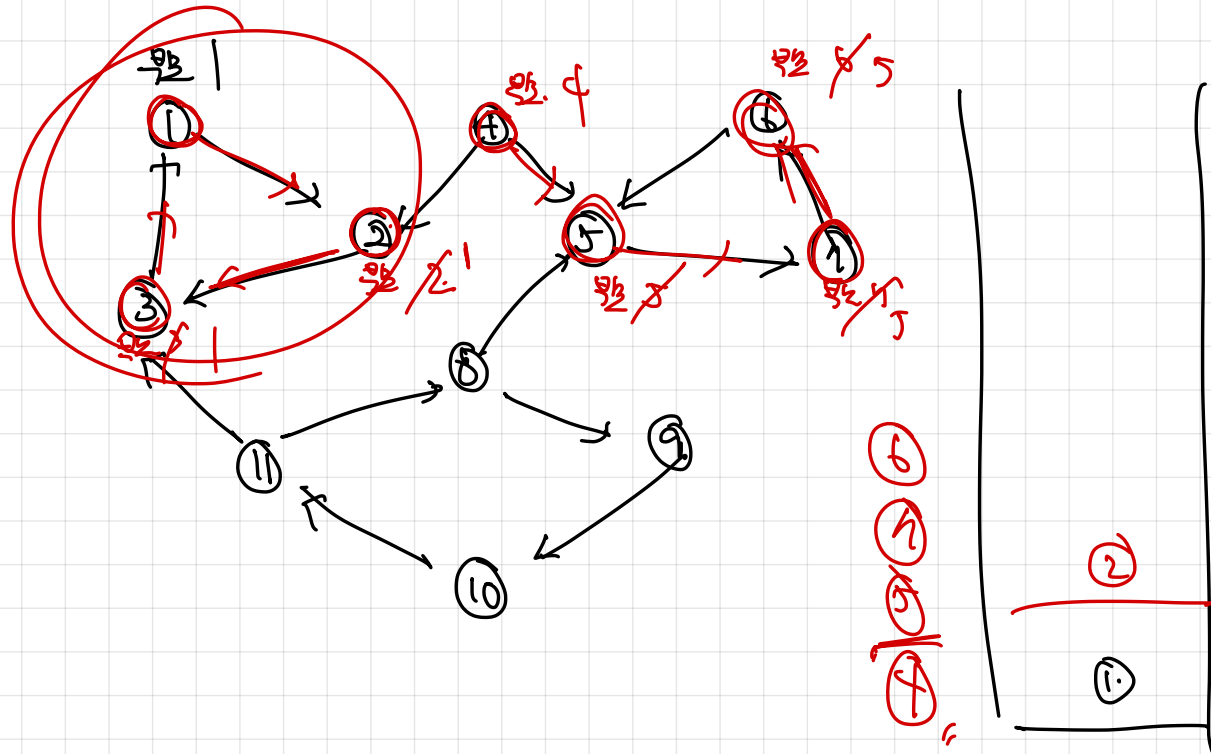
- 방향성이 존재하는 유형 그래프에서 모든 정점이 다른 모든 정점들에 대하여 방문할 수 있는 경우 즉, 어떤 두 정점 간의 경로가 존재하면 그 집단이 강하게 연결되었다고 표현



Connected Components – directed graph

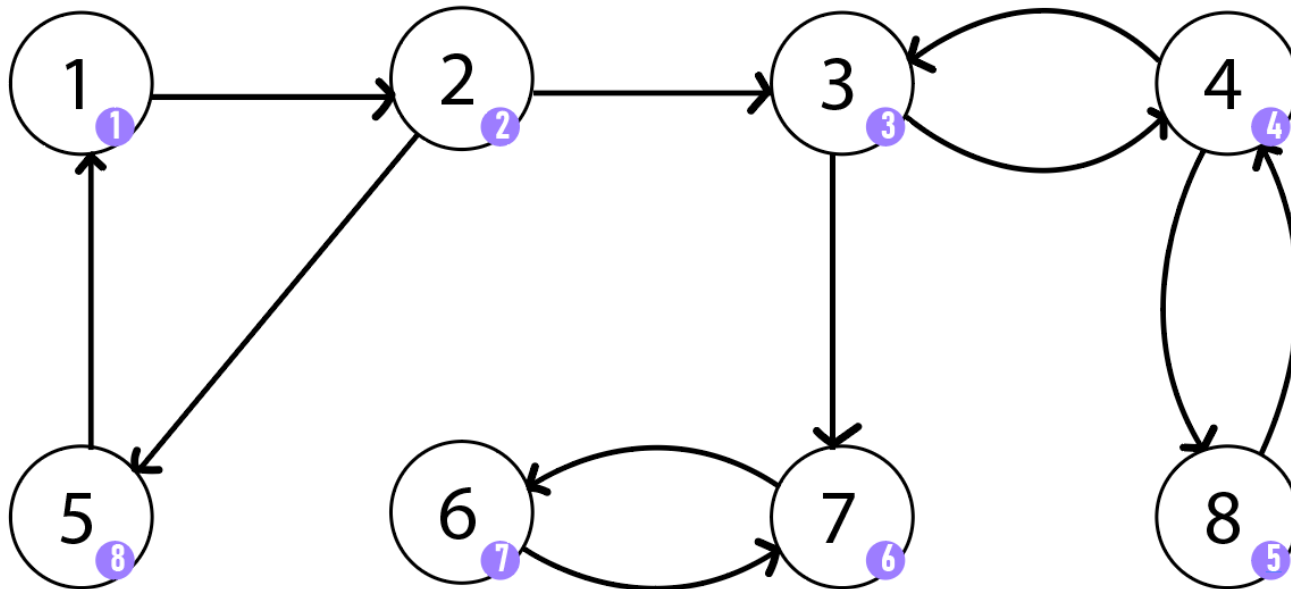
- DFS 수행 순서, 즉 아직 방문하지 않은 정점에 대하여 방문하는 순서에 따라 각 정점에 **고유한 id값**을 매긴다
- d값은 처음에 0으로 되어있고 방문할 때마다 1씩 증가시켜 할당





Connected Components – directed graph

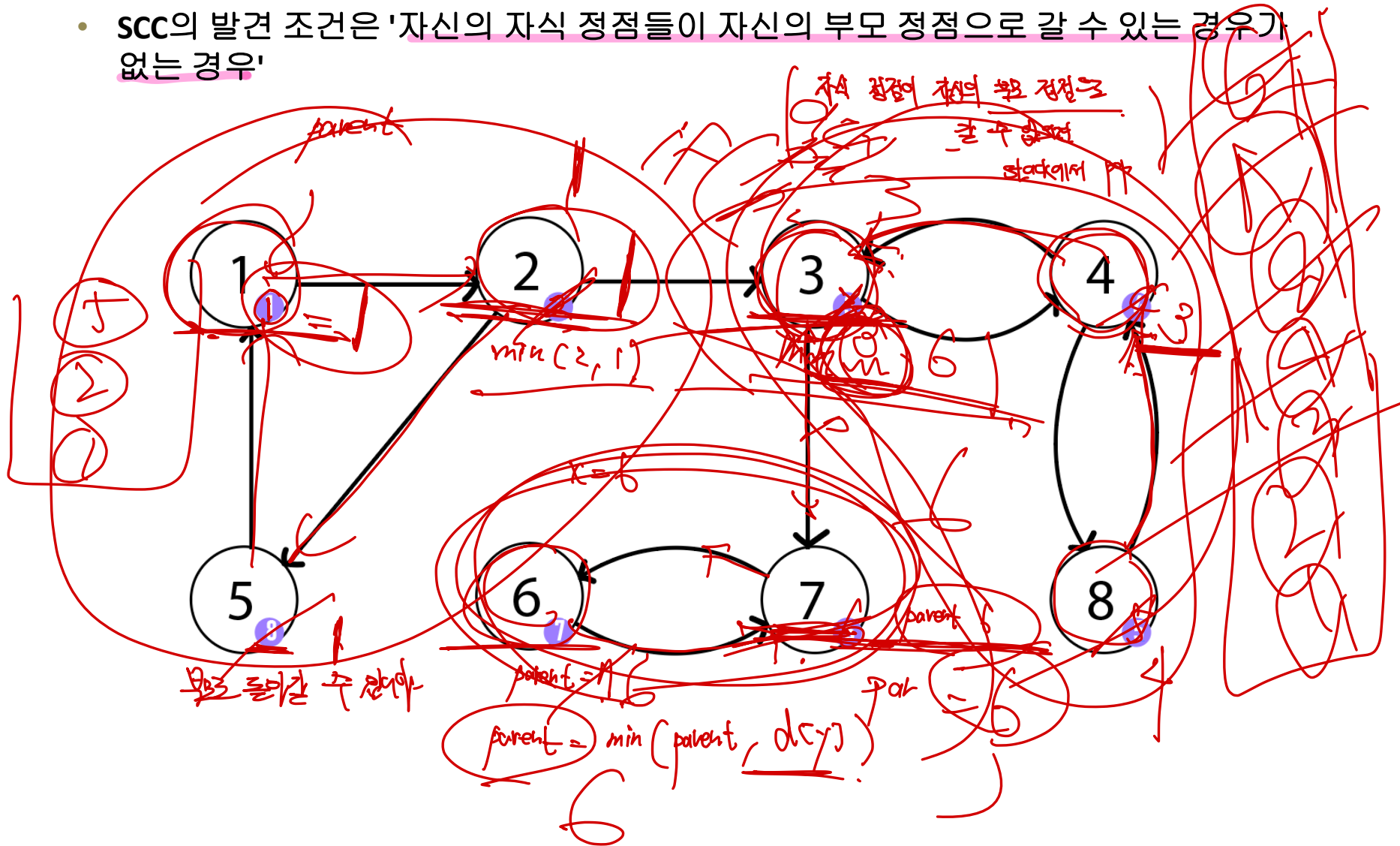
- 방문 순서는 **1 -> 2 -> 3 -> 4 -> 8 -> 7 -> 6 -> 5**
- 방문할 때마다 해당 정점을 스택에 삽입
- 여러개의 정점을 방문 가능하다면 사전 순으로 우선하는 정점을 먼저 방문
- 자신의 부모로 돌아가는 간선은 **(4, 3), (8, 4), (5, 1), (6, 7)**



Connected Components – directed graph

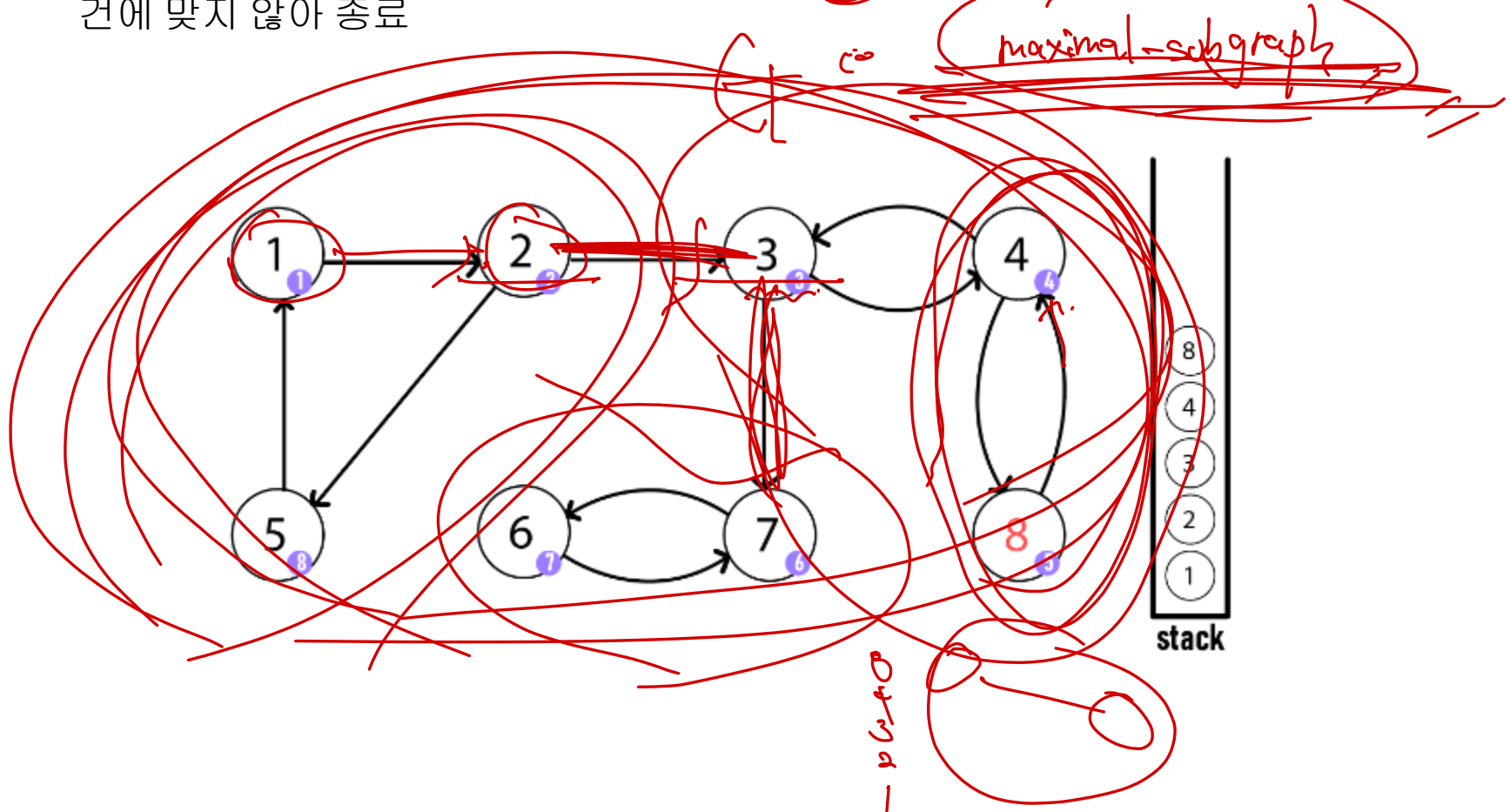
$scc = 2 [6, 4]$

- scc의 발견 조건은 '자신의 자식 정점들이 자신의 부모 정점으로 갈 수 있는 경우가 없는 경우'



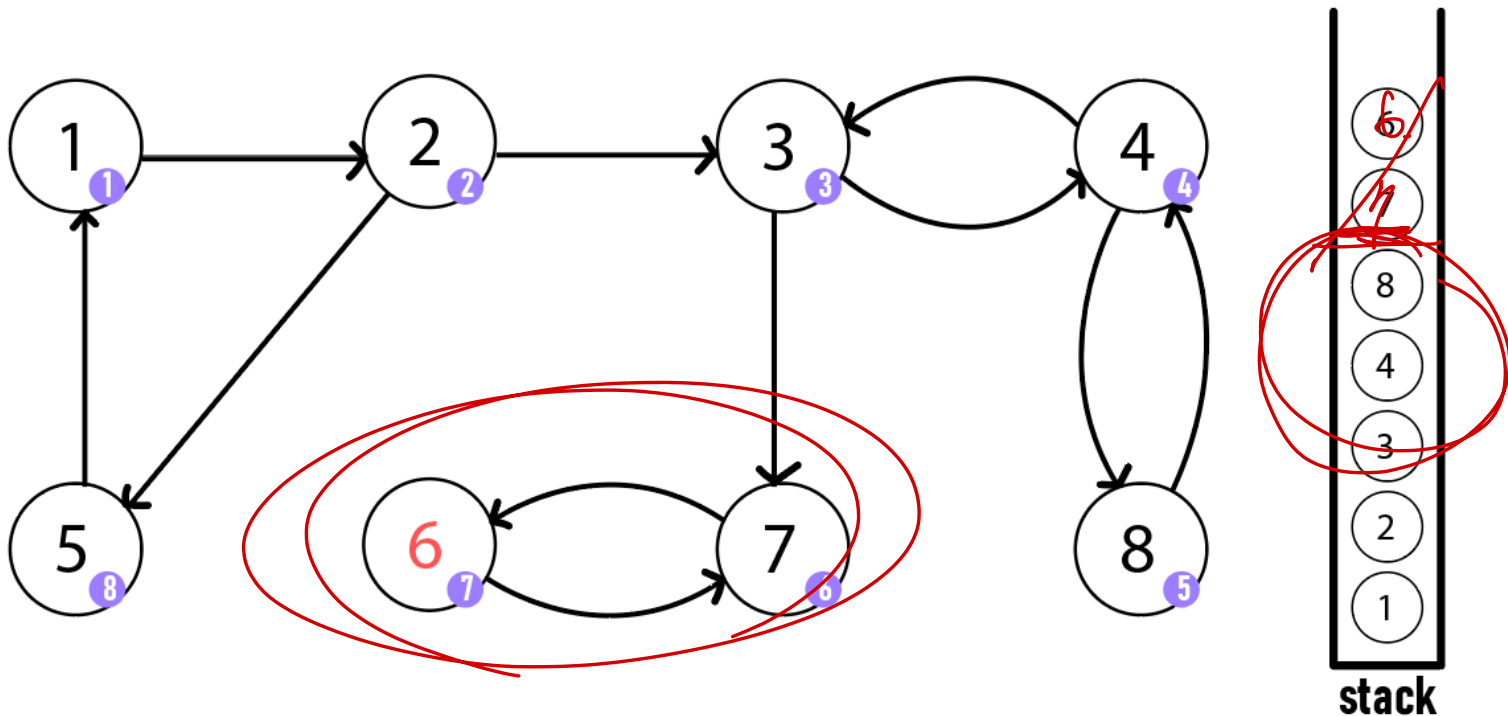
Connected Components – directed graph

- 방향 간선대로 1, 2, 3, 4, 8을 방문하고 스택에 삽입
- 정점 8은 방문할 수 있는 정점이 정점 4인데 자신의 부모 정점이므로 SCC의 발견 조건에 맞지 않아 종료



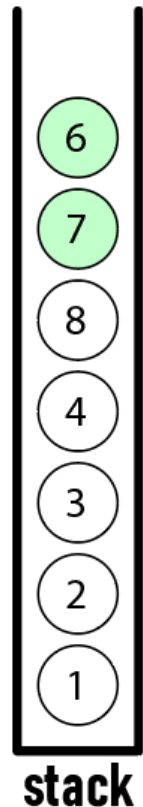
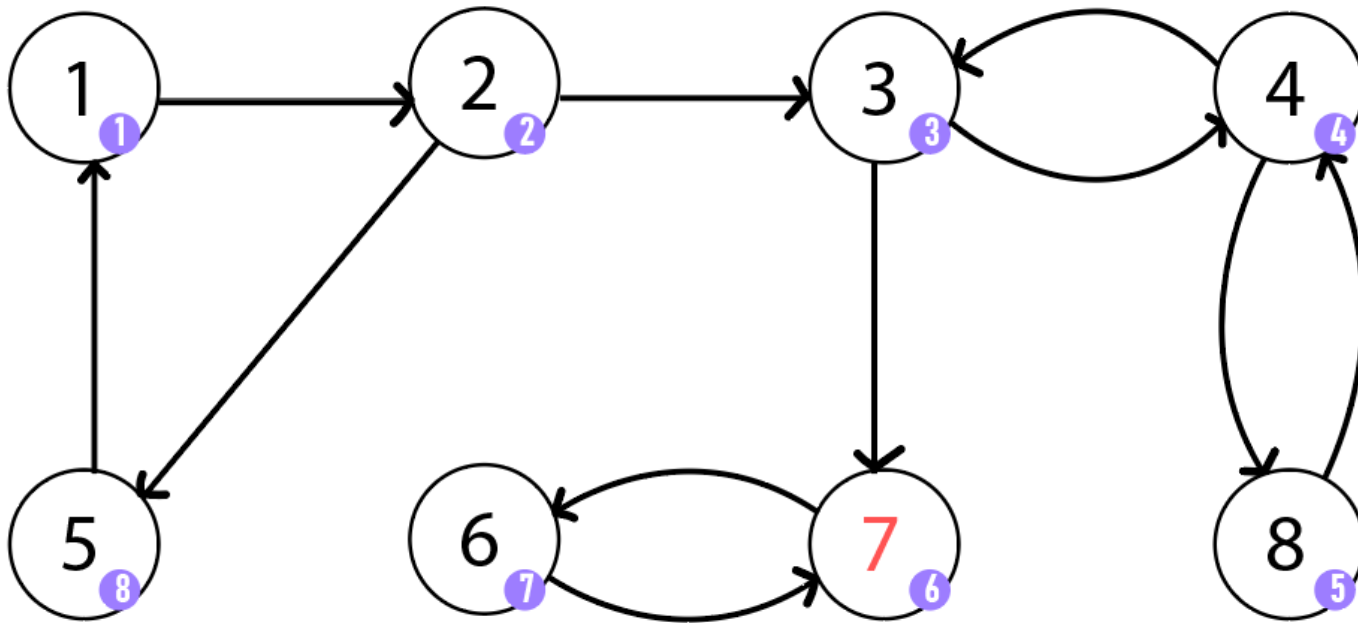
Connected Components – directed graph

- 정점 4도 마찬가지로 자신의 부모 정점인 정점 3으로 갈 수 있으므로 조건에 맞지 않아 종료



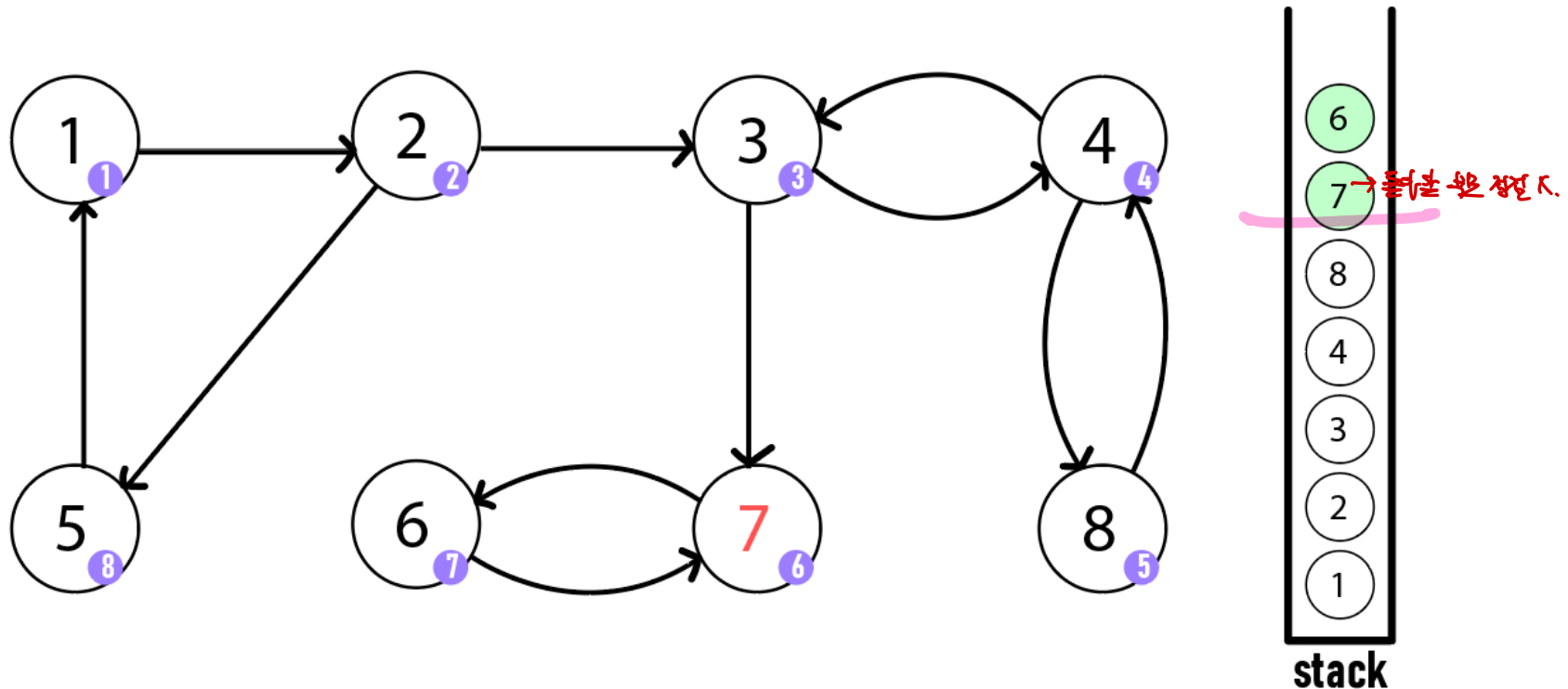
Connected Components – directed graph

- 그 다음 정점 3은 정점 7로 방문할 수 있다
- 정점 7을 방문하고 스택에 삽입
- 정점 7은 정점 6을 방문하여 스택에 삽입
- 정점 6은 자신의 부모 정점 7로 갈 수 있으므로 조건에 맞지 않아 종료



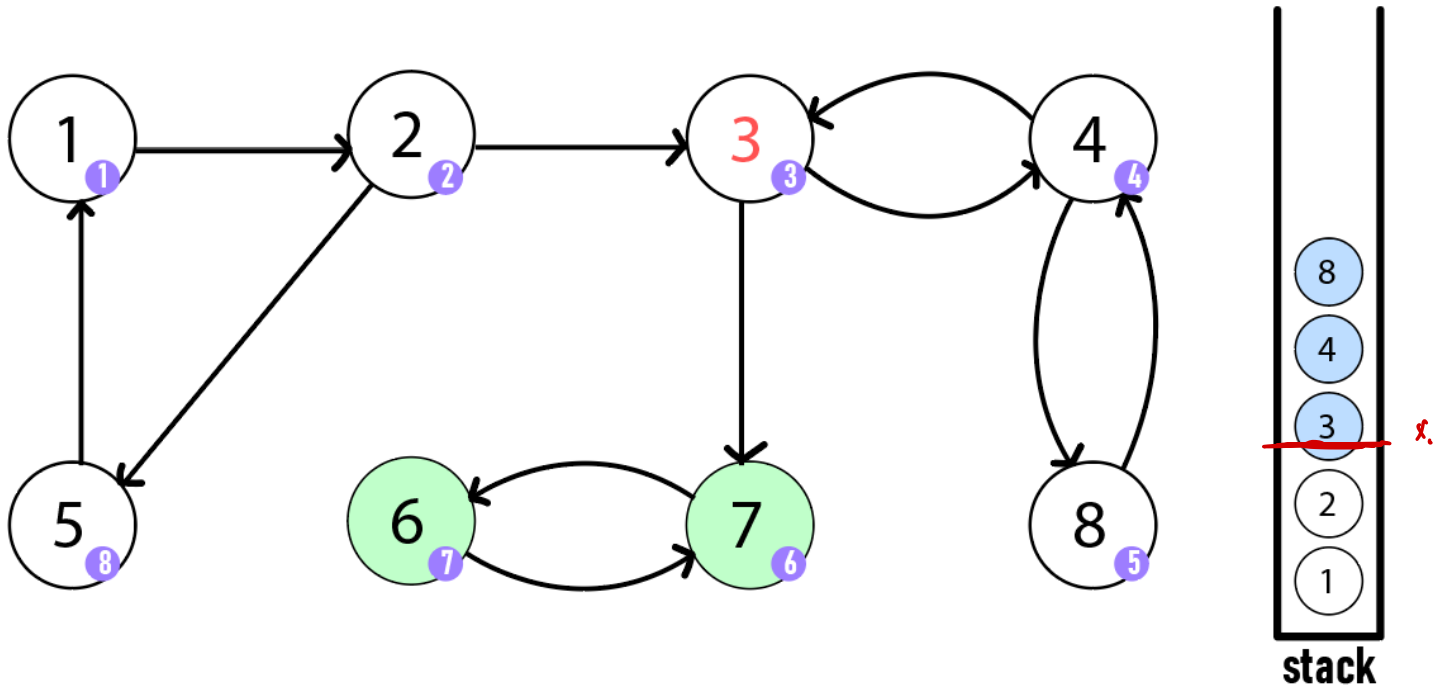
Connected Components – directed graph

- 정점 7로 돌아갔고 정점 7은 돌아갈 수 있는 부모 정점이 없으므로 SCC 발견 조건을 만족
- 정점 7과 자신의 자식 정점인 정점 6도 마찬가지로 정점 7의 부모 정점으로 돌아갈 수 없습니다
- 택에서 정점 7과 그 위에 쌓여있는 정점 6을 묶어서 하나의 SCC를 성립



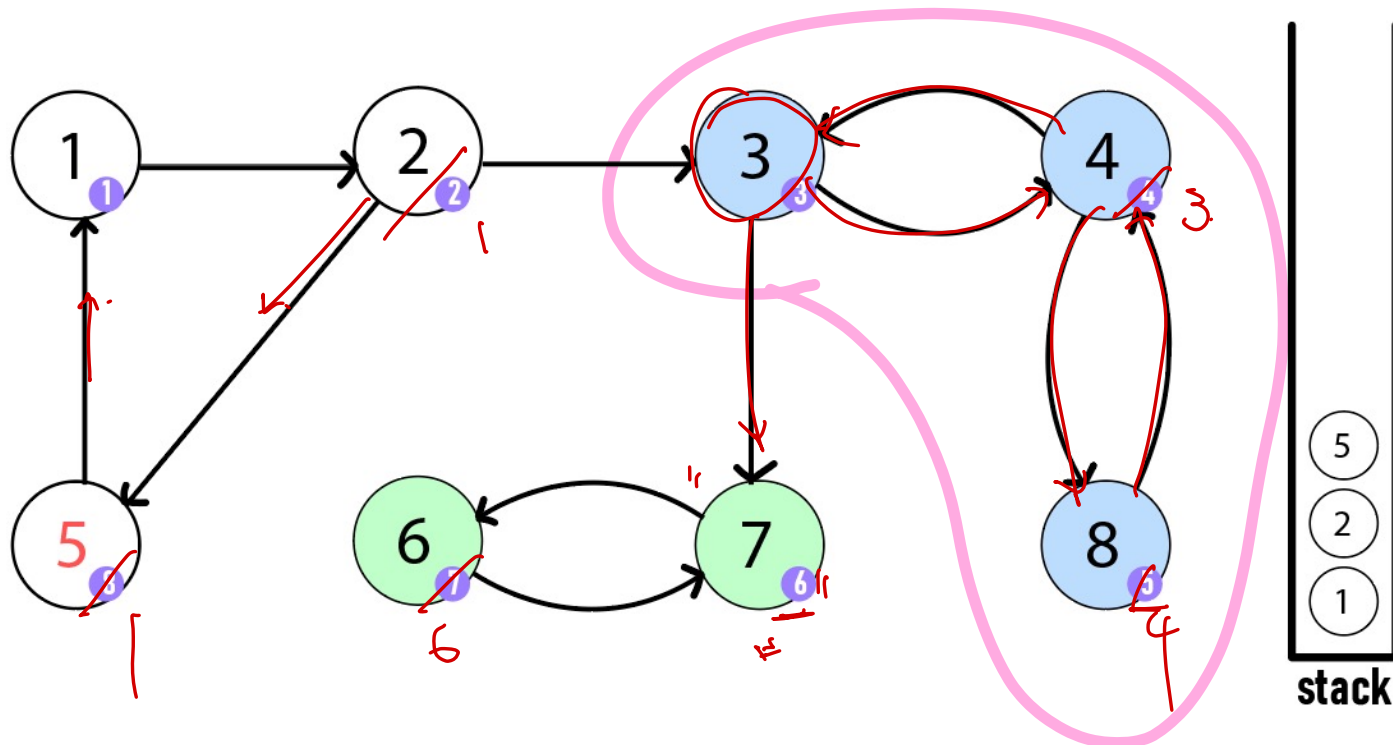
Connected Components – directed graph

- 정점 3으로 돌아갔을 때 정점 3은 더이상 방문할 수 있는 정점이 없으므로 SCC 발견 조건을 만족
- 그러므로 스택에서 정점 3과 그 위에 쌓여있는 정점 4와 8을 묶어서 하나의 SCC를 성립



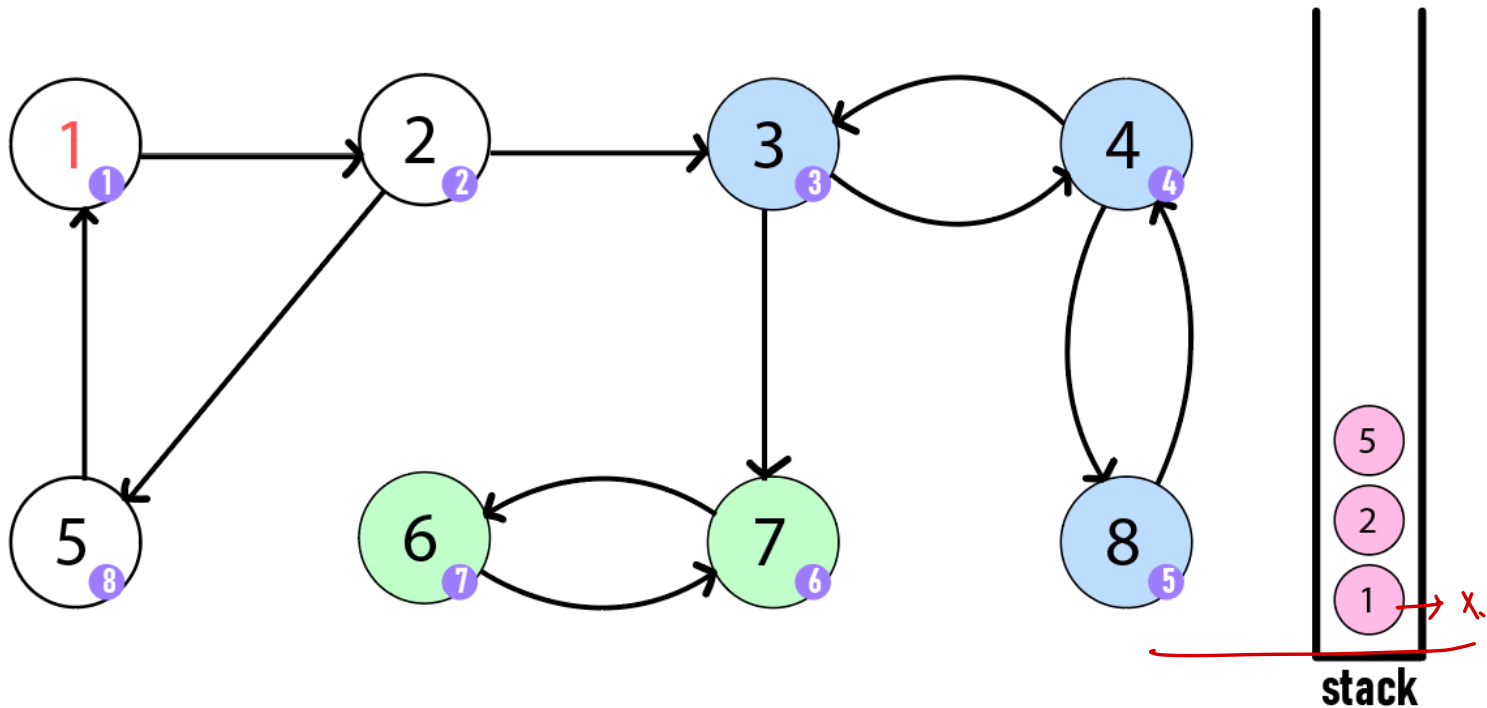
Connected Components – directed graph

- 정점 2로 돌아가서 정점 2는 아직 방문하지 않은 정점 5를 방문
- 정점 5는 부모 정점 1로 갈 수 있으므로 종료



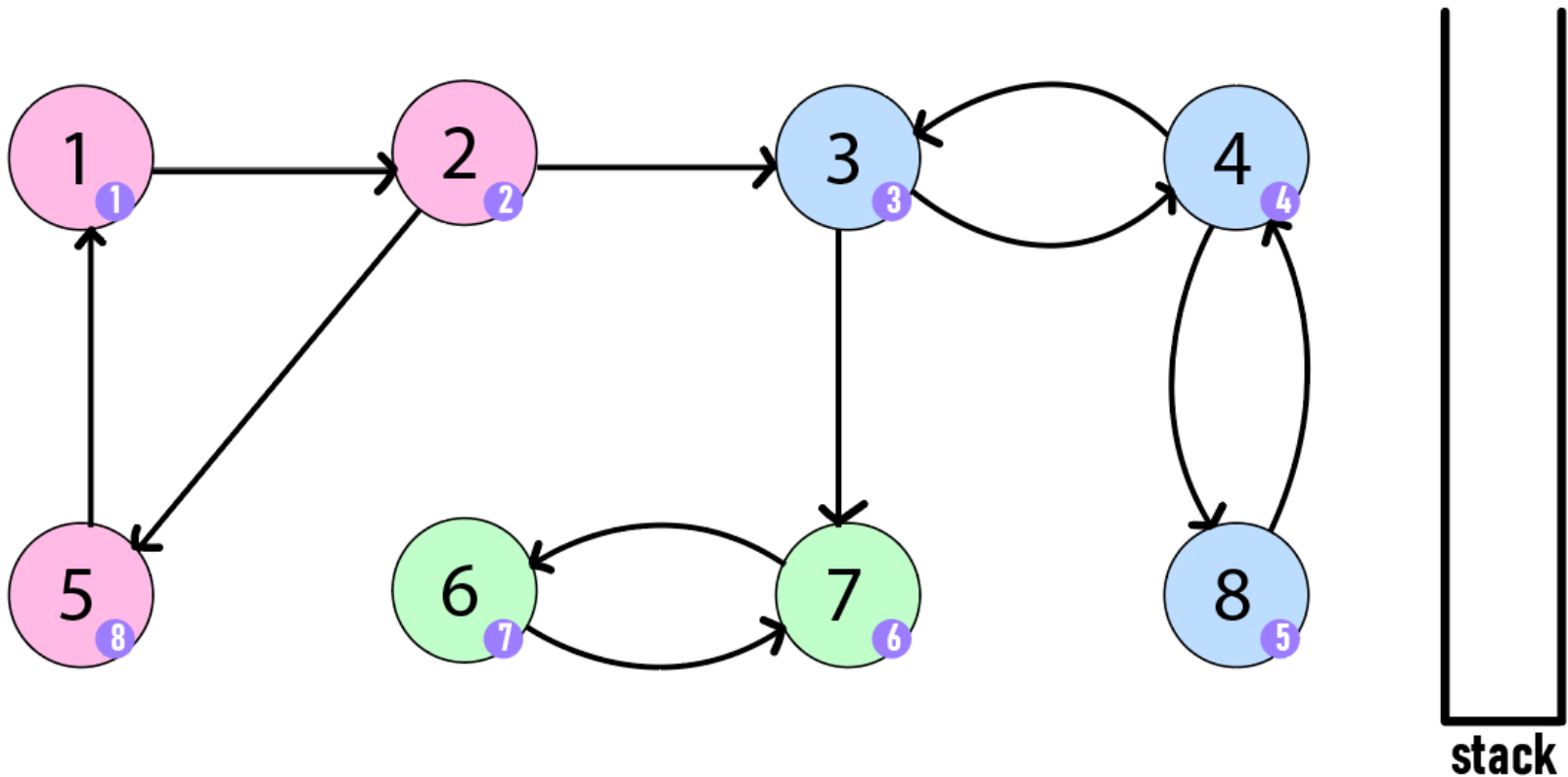
Connected Components – directed graph

- 정점 1로 갔을 때 정점 1은 자신이 루트 정점이었으므로 돌아갈 부모 정점이 없어 SCC 발견 조건을 만족



Connected Components – directed graph

- 모든 scc를 발견하여 스택이 비었고 결과적으로 [정점 6, 7], [정점 3, 4, 8], [정점 1, 2, 5]가 강한 연결 요소



Connected Components – directed graph

- 모든 scc를 발견하여 스택이 비었고 결과적으로 [정점 6, 7], [정점 3, 4, 8], [정점 1, 2, 5]가 강한 연결 요소

