DFS, BFS

Graph, DFS, BFS

그래프(Graph)

- 정의
  - 아이템(사물 또는 추상적 개념)들과 이들 사이의 연결 관계를 표현하는 자료 구조.
  - 정점(Vertex)들의 집합과 이들을 연결하는 간선(Edge)들의 집합으로 구성된 자료 구조
    - |V| : 정점의 개수, |E| : 그래프에 포함된 간선의 개수
    - |V| 개의 정점을 가지는 그래프에서
      최소 (|V| – 1) 간선 , 최대 |V| (|V| – 1)/ 2 간선이 가능

    예> 5개 정점이 있는 그래프의 최대 간선 수는 10(= 5*4/2) 개이다..
  - 선형 자료구조나 트리 자료구조로 표현하기 어려운 N : N 관계를 가지는 원소들을 표현하기에 용이하다

그래프(Graph)

- 용어
  - 인접(Adjacency)
    - 두 개의 정점에 간선이 존재(연결됨)하면 서로 인접해 있다고 한다.
  - 경로란 간선들을 순서대로 나열한 것
    - 간선들: (0, 2), (2, 4), (4, 6)
    - 정점들: 0 – 2 – 4 – 6
  - 경로 중 한 정점을 최대한 한번만 지나는 경로를 단순경로라 한다.(예: 0 – 2 – 4 – 6, 0 – 1 – 6)
  - 시작한 정점에서 끝나는 경로를 사이클(Cycle)이라고 한다. (예: 1 – 3 – 5 – 1)

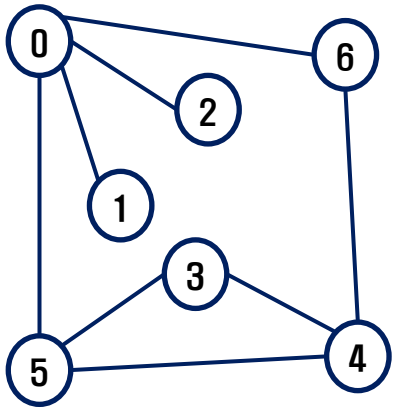
그래프(Graph)의 표현

- 인접 행렬 (Adjacent matrix)
  - |V| x |V| 크기의 2차원 배열을 이용해서 간선 정보를 저장
  - 배열의 배열
- 인접 리스트 (Adjacent List)
  - 각 정점마다 해당 정점으로 나가는 간선의 정보를 저장
- 간선 배열
  - 간선(시작 정점, 끝 정점)을 배열에 연속적으로 저장

그래프 -인접 행렬

- 두 정점을 연결하는 간선의 유무를 행렬로 표현
  - |V| x |V| 정방 행렬
  - 행 번호와 열 번호는 그래프의 정점에 대응
  - 두 정점이 인접되어 있으면 1, 그렇지 않으면 0으로 표현
  - 무향 그래프인 경우
    - *i* 번째 행의 합 = *i* 번째 열의 합 = V_i 의 차수
  - 유향 그래프인 경우
    - 행 *i* 의 합 = V_i 의 진출 차수
    - 열 *i* 의 합 = V_i 의 진입 차수

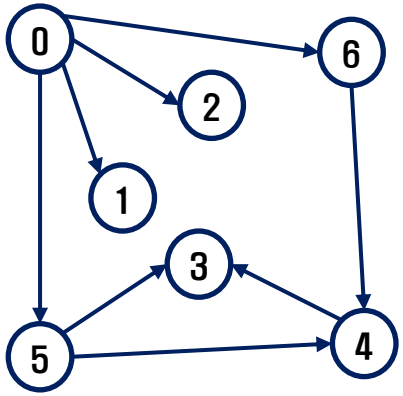
그래프 -인접 행렬



	0	1	2	3	4	5	6
0	0	1	1	0	0	1	1
1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	0
3	0	0	0	0	1	1	0
4	0	0	0	1	0	1	1
5	1	0	0	1	1	0	0
6	1	0	0	0	1	0	0

1, 무향그래프

- 인접한 두 정점의 행 번호, 열 번호에 해당하는 곳에 1 로 표시
- 1의 개수는 간선 개수의 두 배



	0	1	2	3	4	5	6
0	0	1	1	0	0	1	1
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0
5	0	0	0	1	1	0	0
6	0	0	0	0	1	0	0

→ 진출차수

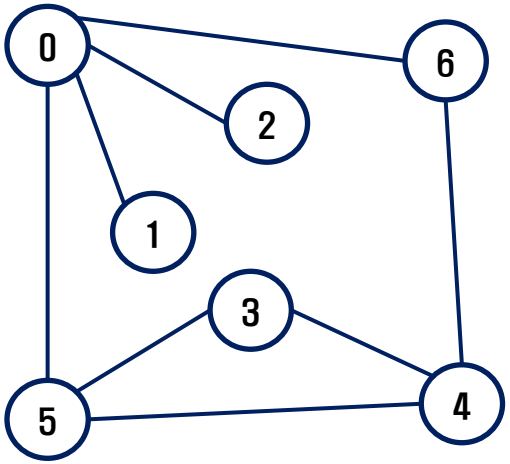
↓ 진입차수

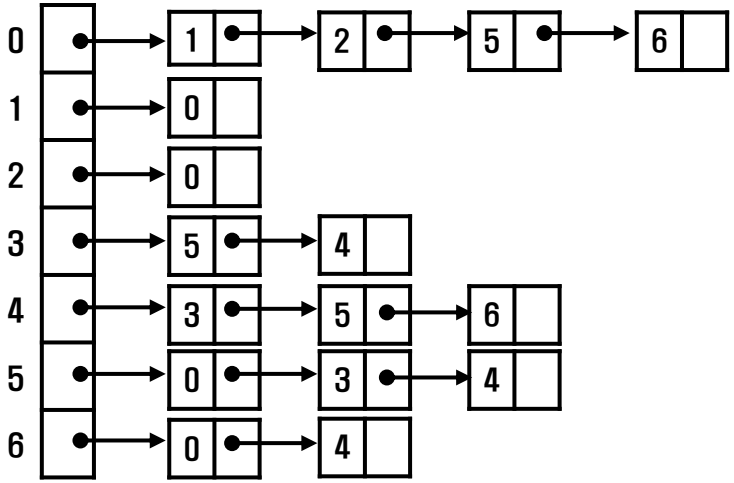
2. 유향그래프

- 간선의 수 만큼 1로 표현
- 정점의 진출 차수
  – 행에 표시된 1의 개수
- 정점의 진입 차수
  - 열에 표시된 1의 개수

그래프 -인접 리스트

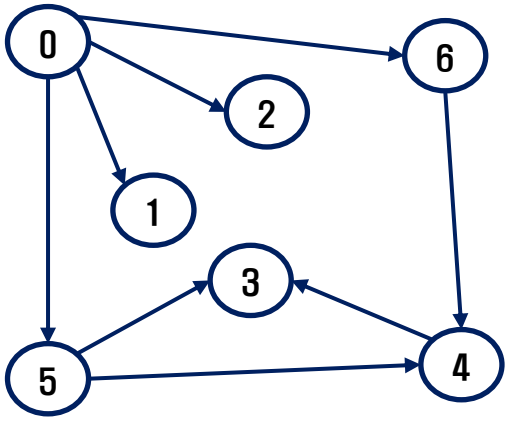
- 인접리스트
  - 인접 행렬의 메모리 낭비를 줄이기 위한 구현 방법
- 무방향 그래프
  - 노드 수 = 간선의 수 * 2
  - 각 정점의 노드 수 = 정점의 차수

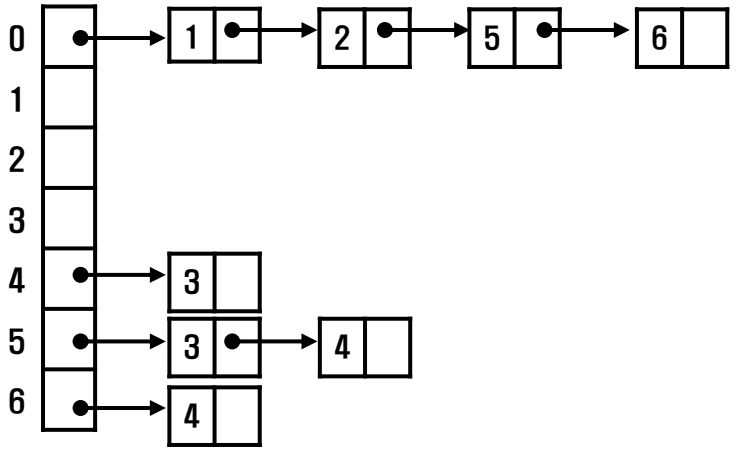




그래프 -인접 리스트

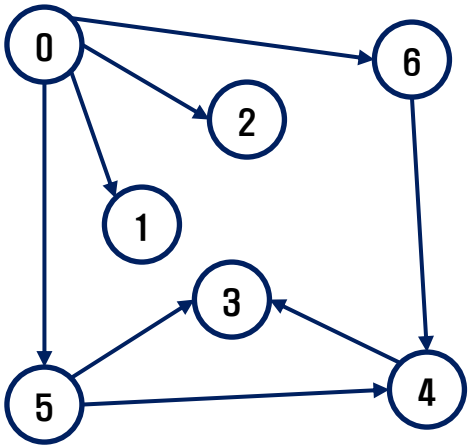
- 방향 그래프
  - 노드 수 = 간선의 수
  - 각 정점의 노드 수 = 정점의 진출 차수
- 인접 리스트는 링크드 리스트의 형태로 구현되나,
  2차원 배열에 인접 정점의 정보를 저장하여 구현할 수도 있다.





그래프 – 간선 배열

- 간선 배열
  - 간선의 수가 많지 않을 경우 메모리의사용을 줄이기 위해 간선의 정보를 나열해서 저장
  - 간선의 시작 정점과 끝 정점의 정보 저장
  - 간선 리스트에서 어떤 정점의 인접 정점은 일치하는 시작 정점을 찾으면 됨.



	시작 정점	끝 정점
0	0	1
1	0	2
2	0	5
3	0	6
4	4	3
5	5	3
6	5	4
7	6	4

그래프 표현에 따른 비교

- 인접 행렬
  - 정점의 개수 n 이 커지면 인접 행렬에 필요한 메모리의 크기는 n^2 에 비례 : 메모리 낭비
  - 어떤 정점의 인접 정점을 찾을 때마다 n 번의 탐색
- 인접 리스트
  - 인접 행렬에 비해 적은 메모리를 사용하고, 인접 정점을 탐색 횟수를 줄일 수 있다.
- 간선 배열
  - 간선의 수가 적을 경우 메모리 사용을 최소할 수 있다.
  - 인접 정점을 찾을 때마다 간선 배열을 순차 검색해야 한다.

그래프 탐색

- 비선형구조인 그래프로 표현된 모든 자료(정점)를 빠짐없이 탐색하는 것을 의미한다.

- 두 가지 방법
  - 깊이 우선 탐색(Depth First Search, DFS)
  - 너비 우선 탐색(Breadth First Search, BFS)

BFS vs. DFS

- BFS(Breath-First Search)
  - 시간 복잡도 : O(m+n) m: 간선의 수 n: 정점의 수
  - FIFO : 큐 사용하여 구현
  - 같은 layer에서 탐색 기법
  - 최단 경로, 무향 그래프에서의 연결된 정점 처리에 유용

- DFS(Depth-First Search)
  - 시간 복잡도 : O(m+n) m: 간선의 수 n: 정점의 수
  - LIFO : 스택 또는 재귀로 구현
  - 필요할 때만 탐색하고, 그렇지 않을 경우 되돌아가는 경우, 미로 찾기 구현에 적합
  - 유향 그래프에서의 연결된 정점 처리에 유용

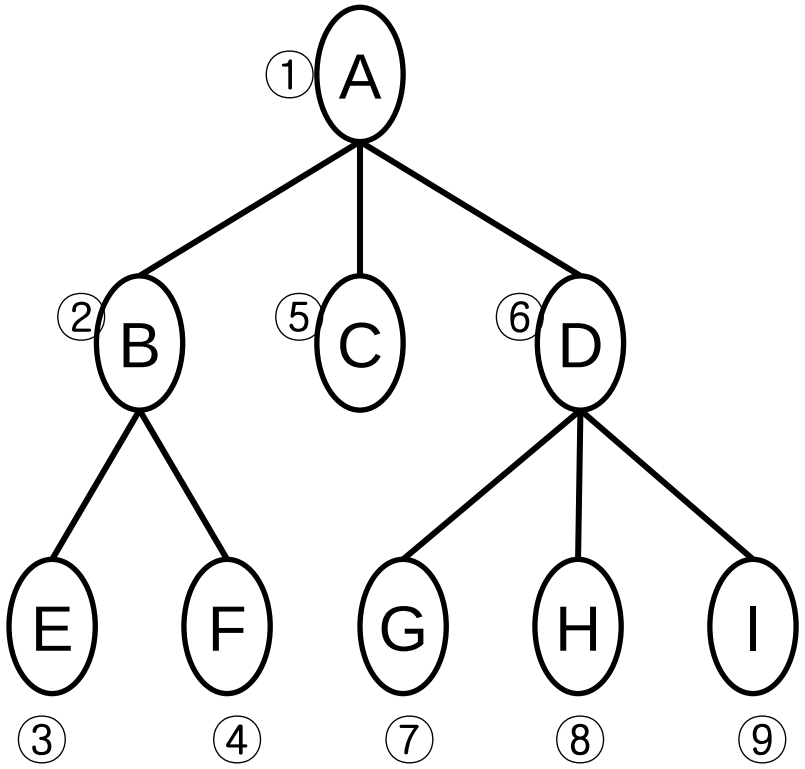
DFS(깊이우선탐색)

- 시작 정점의 한 방향으로 갈 수 있는 경로가 있는 곳까지 깊이 탐색해 가다가 더 이상 갈 곳이 없게 되면, 가장 마지막에 만났던 갈림길 간선이 있는 정점으로 되돌아와서 다른 방향의 정점으로 탐색을 계속 반복하여 결국 모든 정점을 방문하는 순회방법

- 가장 마지막에 만났던 갈림길의 정점으로 되돌아가서 다시 깊이 우선 탐색을 반복해야 하므로 후입선출 구조의 스택 사용

DFS(Depth First Search)

- DFS는 예제 그래프를 붙여진 번호 순서로 탐색



1. A 탐색 시작
2. A 의 인접 정점 중 선택(B)
3. B 방문
4. B 의 인접 정점 중 선택(E)
5. E 방문,
6. E의 인접 정점 중 방문하지 않은 정점 없으므로 B 로 되돌아 감.
7. B 의 또 다른 인접 정점 F 선택
8. F 방문, F 의 인접 정점 중 방문하지 않은 정점 없으므로 B로 되돌아 감.
9. B 의 인접 정점은 모두 방문했으므로, A 로 되돌아 감.
10. A 의 인접 정점 중 C , D 에 대해서도 위와 같은 방법으로 반복

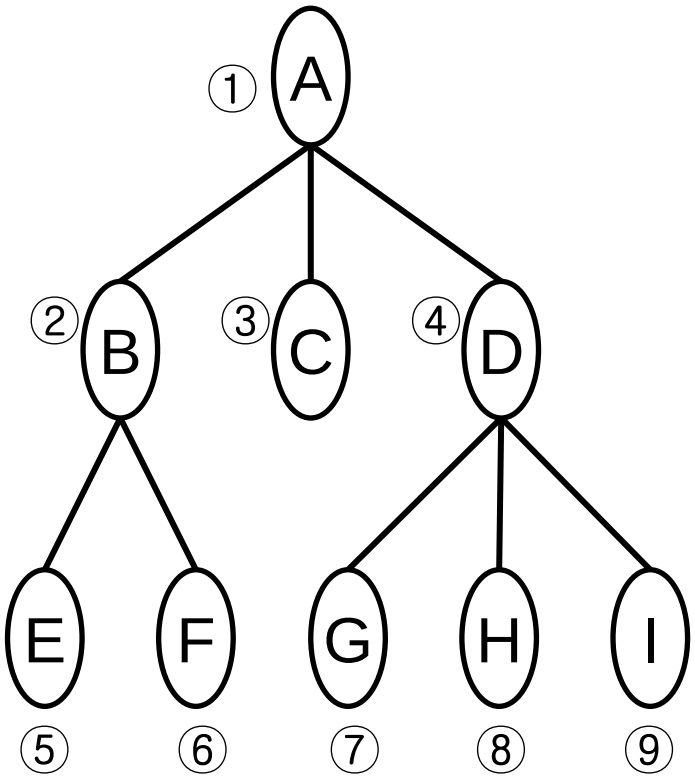
BFS(Breadth First Search)

- 너비우선탐색은 탐색 시작점의 인접한 정점들을 먼저 모두 차례로 방문한 후에, 방문했던 정점을 시작점으로 하여 다시 인접한 정점들을 차례로 방문하는 방식

- 인접한 정점들에 대해 탐색을 한 후, 차례로 다시 너비우선탐색을 진행해야 하므로, 선입선출 형태의 자료구조인 큐를 활용함

BFS(Breadth First Search)

- BFS는 예제 그래프를 붙여진 번호 순서로 탐색함



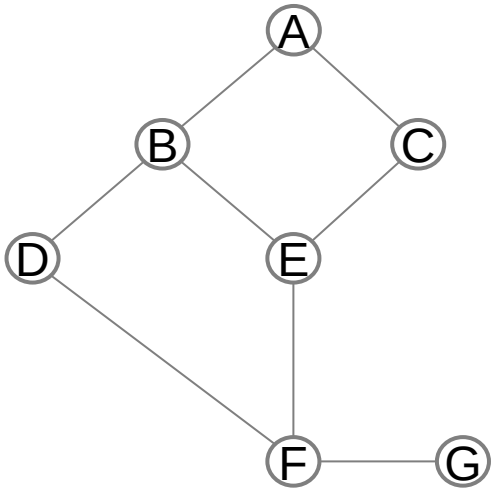
1. A 탐색 시작
2. A 의 인접 정점들 B, C, D 를 방문
3. B 의 인접 정점 E, F 를 방문
4. C 는 방문하지 않은 인접 정점이 없으므로 큐에서 새로운 정점(D) 선택
5. D의 인접 정점 중 방문하지 않은 정점 G, H, I 를 방문

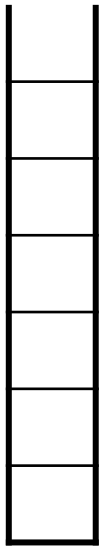
DFS 알고리즘 1

```
visited[] : 방문 여부 체크 배열, stack[]: 방문했던 정점들 저장
DFS(v):
     v 방문                          # 출력, 계산 등의 처리
     visited[v] = true               # v 의 방문을 표시

     while(v)  {
          v의 인접 정점 중 방문 안 한 정점 w 를 찾는다.
          if (w)                     # W 가 존재한다면
               push(v)               # 스택에 현재 정점을 저장한다.

          while( w )                 # 인접 정점이 존재하는 동안 반복 실행
               w 방문                # 인접 정점 W 의 출력, 계산 등의 처리
               visited[w] = true     # w 의 방문을 표시
               push(w)               # 스택에 현재 정점엔 W 저장
               v = w                 # v 를 현재 정점 W 로 바꿔준다.
               v의 인접 정점 중 방문 안 한 정점 w 를 찾는다.

          v = stack.pop()            # 최근 방문한 정점을 가져오게 된다.
```

DFS 알고리즘 동작 예

`visited[], stack[]` 초기화





그래프의 표현 방법

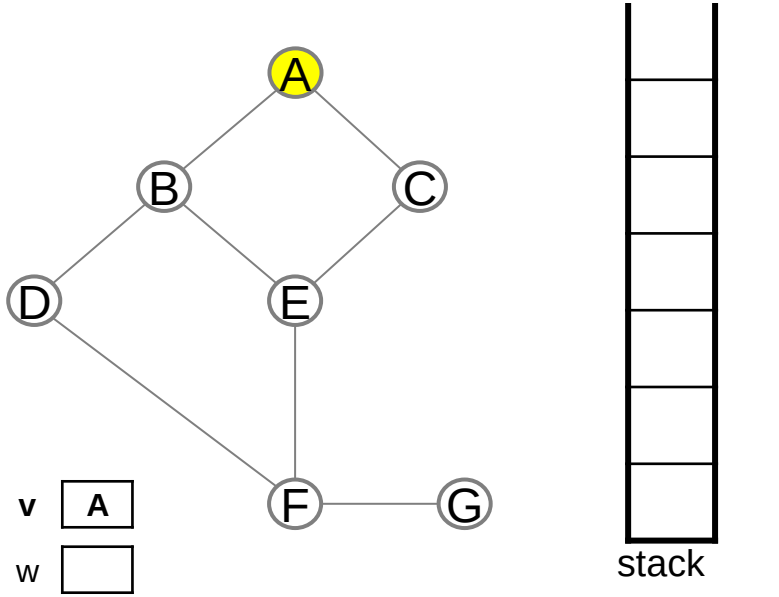
	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	1	0	0	1	1	0	0
C	1	0	0	0	1	0	0
D	0	1	0	0	0	1	0
E	0	1	1	0	0	1	0
F	0	0	0	1	1	0	1
G	0	0	0	0	0	1	0

A	**B**	**C**	
B	A	D	E
C	A	E	
D	B	F	
E	B	C	F
F	D	E	G
G	F		

정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	F	F	F	F	F	F	F

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
       w 방문
       visited[w] = true
       push(w)
       v = w
       w = v의 미방문 인접정점

    v = stack.pop()
```

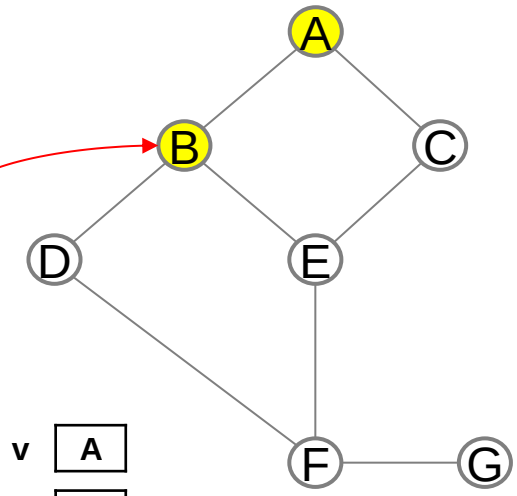


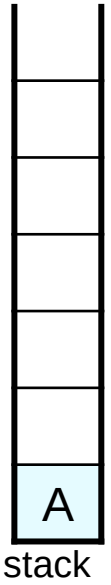
정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	F	F	F	F	F	F

출력 : A

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```



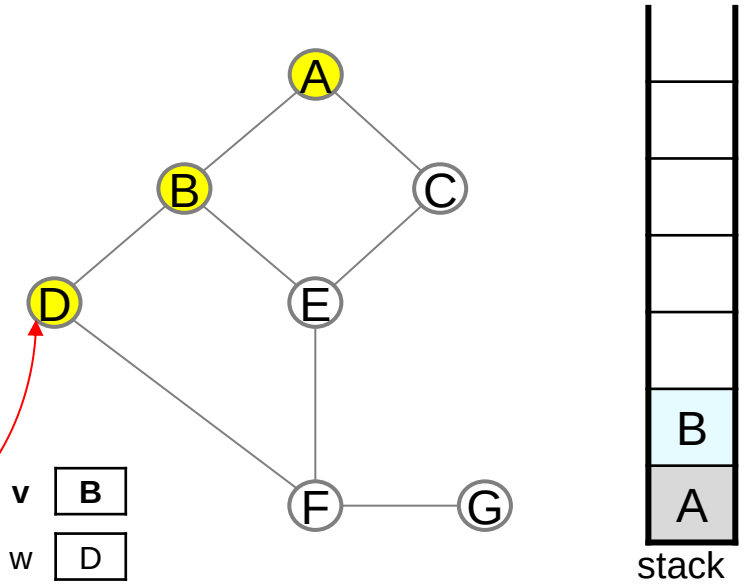


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	F	F	F	F	F

출력 : A - B

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```

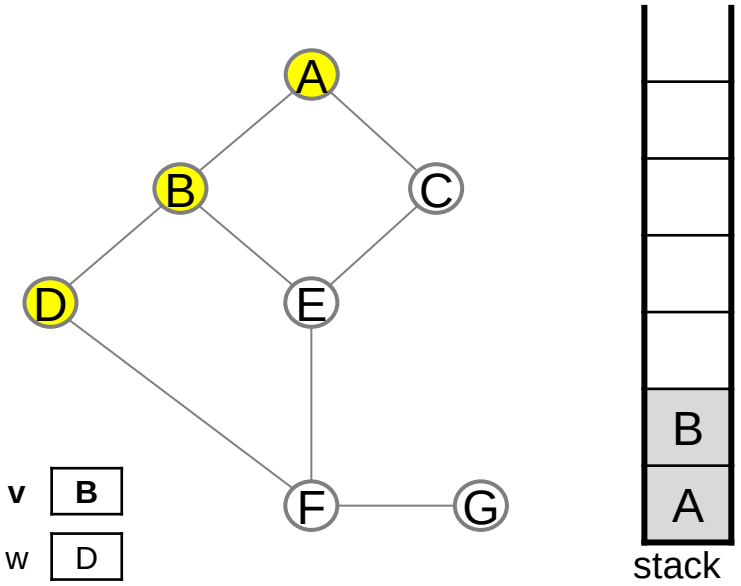


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	F	F	F	F	F

출력 : A - B

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```

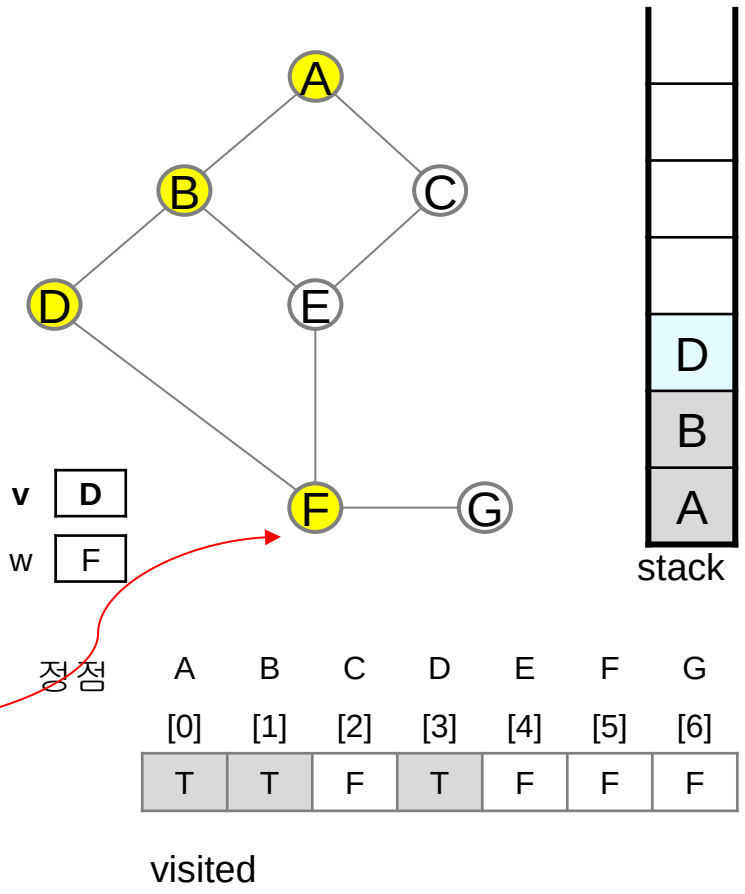


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	F	T	F	F	F

출력 : A – B – D

DFS 알고리즘 동작 예

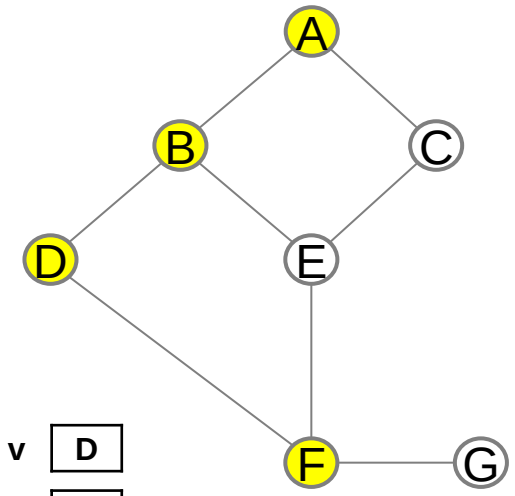
```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```

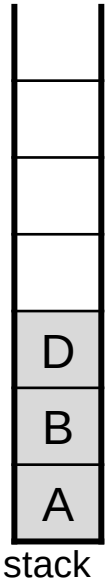


출력 : A – B – D

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```



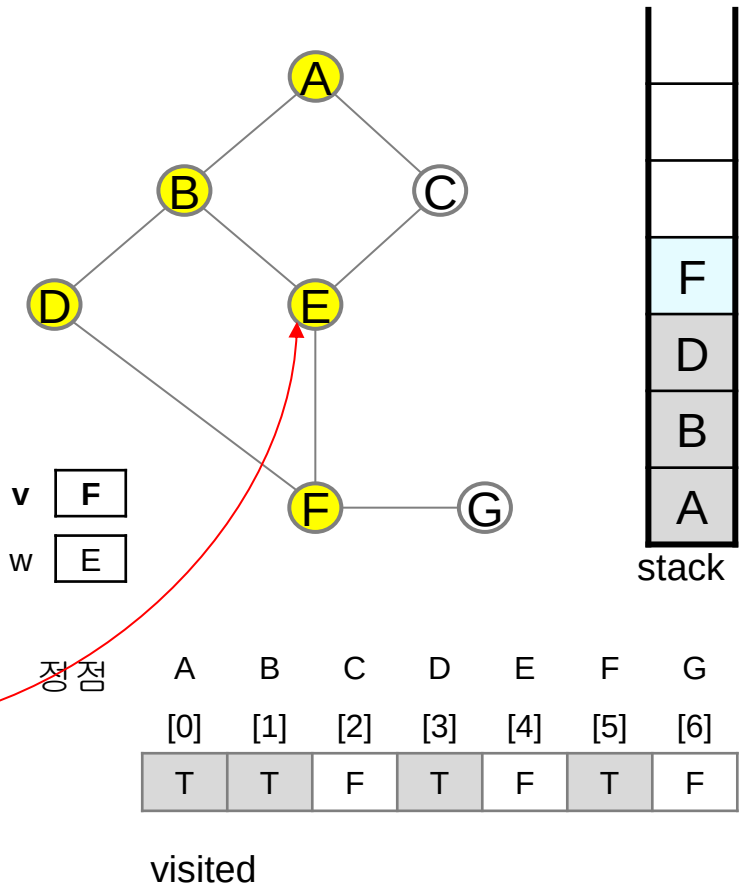


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	F	T	F	T	F

출력 : A – B – D – F

DFS 알고리즘 동작 예

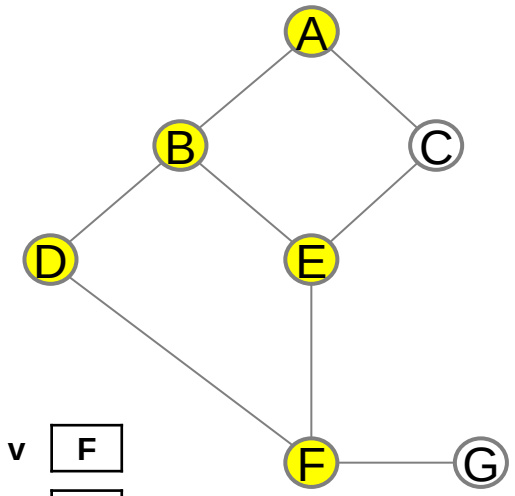
```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
       w 방문
       visited[w] = true
       push(w)
       v = w
       w = v의 미방문 인접정점

    v = stack.pop()
```

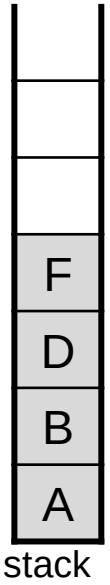


출력 : A – B – D – F

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```



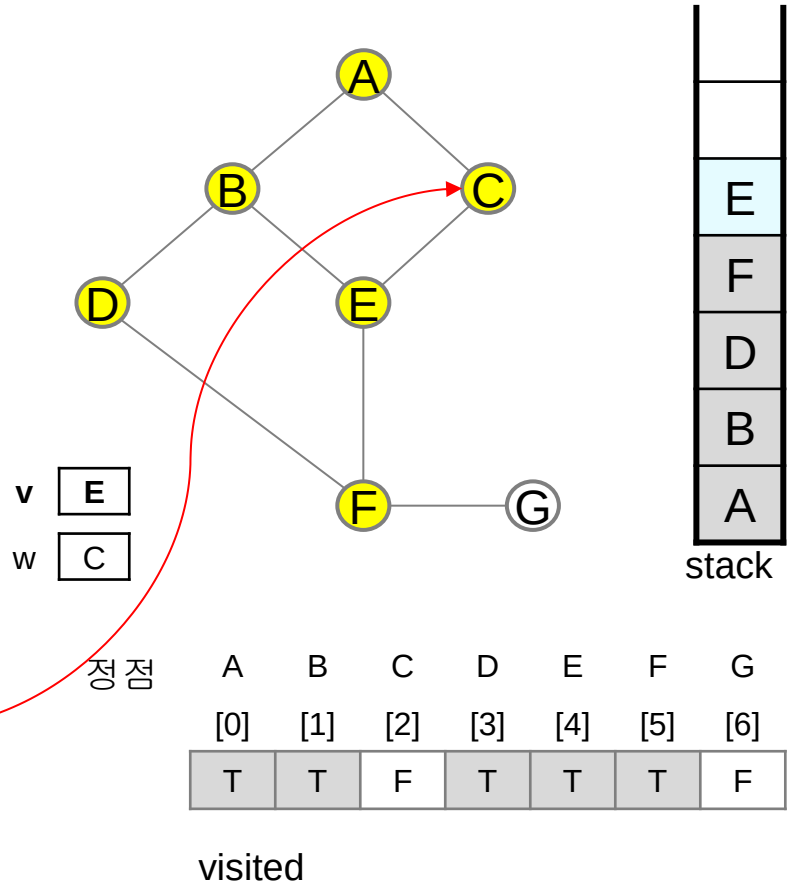


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	F	T	T	T	F

출력 : A – B – D – F – E

DFS 알고리즘 동작 예

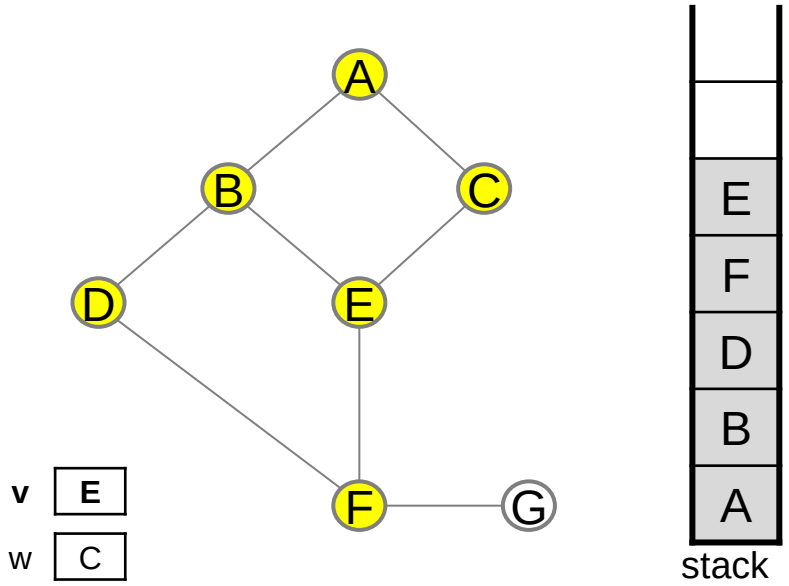
```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```



출력 : A – B – D – F – E

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```

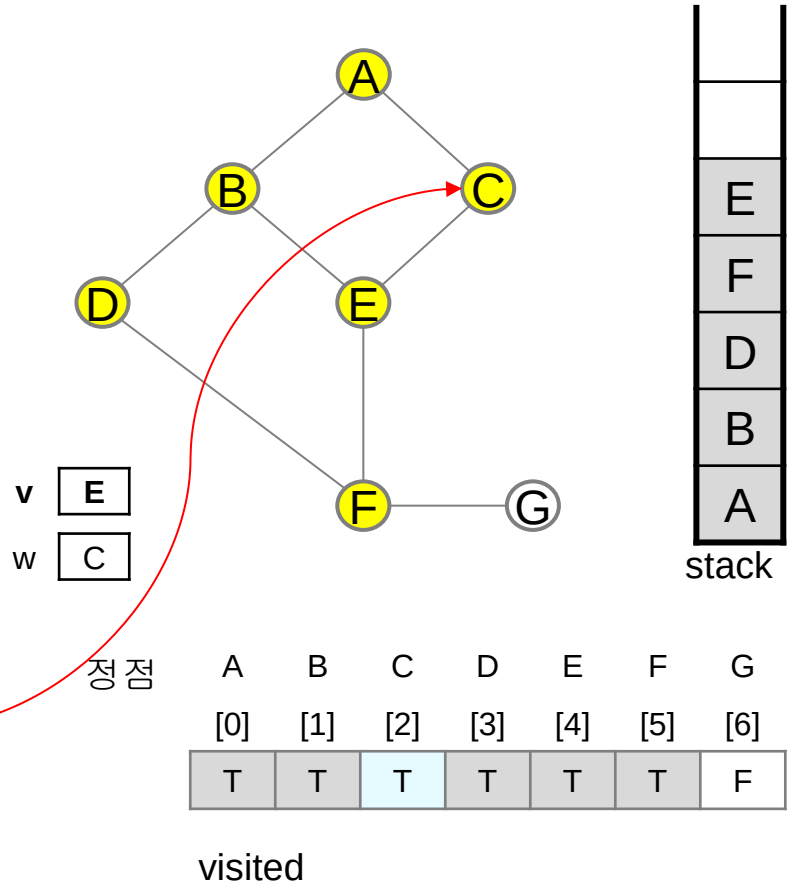


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	T	F

출력 : A – B – D – F – E – C

DFS 알고리즘 동작 예

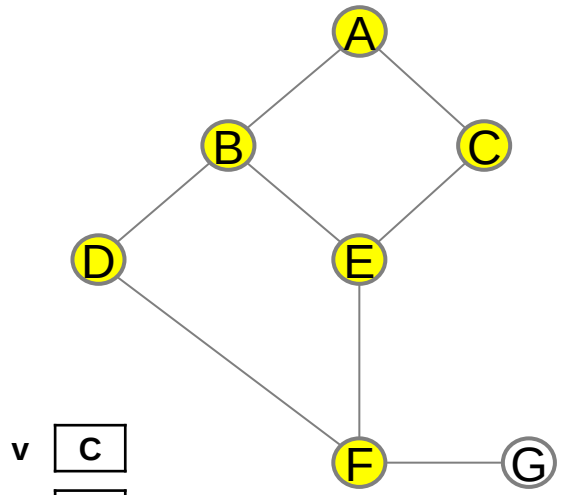
```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```

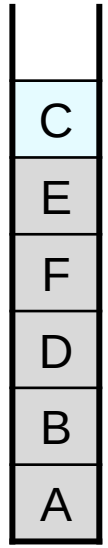


출력 : A – B – D – F – E – C

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
       w 방문
       visited[w] = true
       push(w)
       v = w
       w = v의 미방문 인접정점

    v = stack.pop()
```



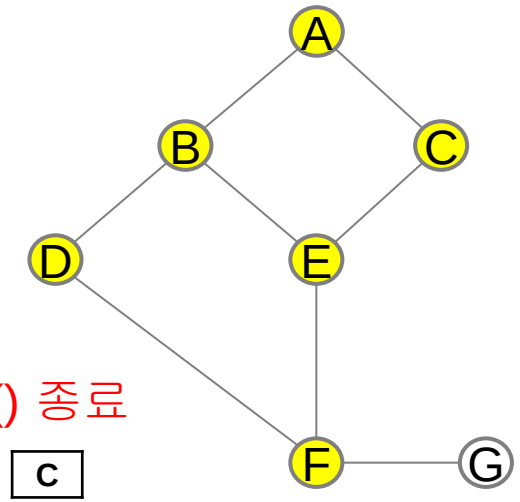


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	T	F

출력 : A – B – D – F – E – C

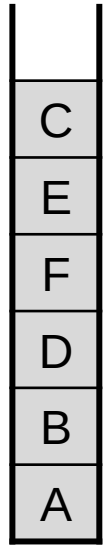
DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )  # w 가 없으므로 while() 종료
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```



v C

w



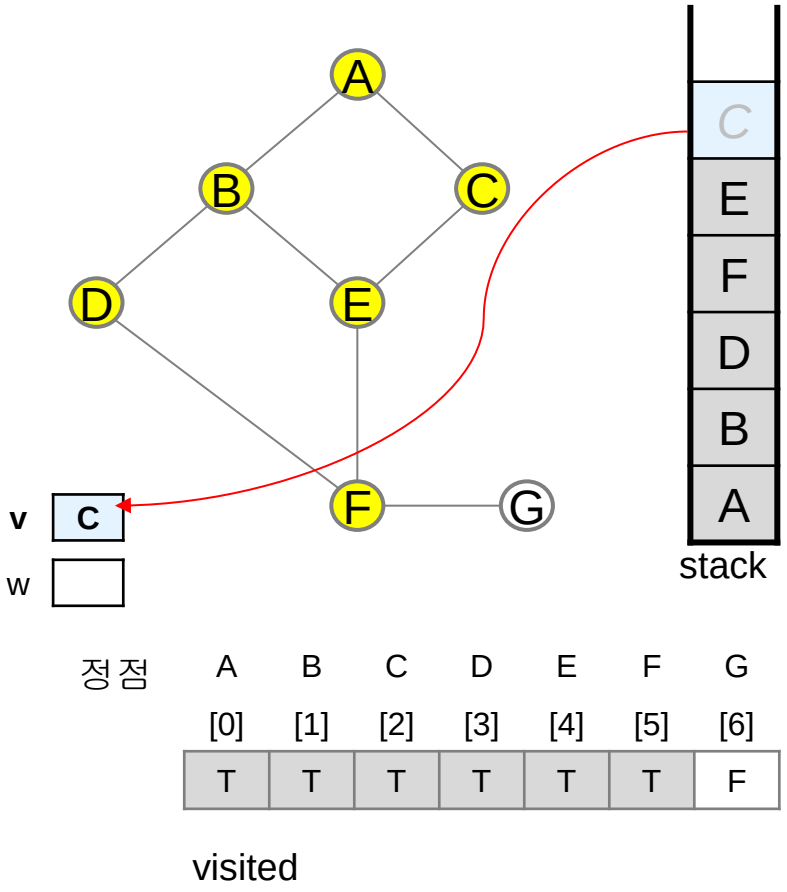
stack

정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	T	F

출력 : A – B – D – F – E – C

DFS 알고리즘 동작 예

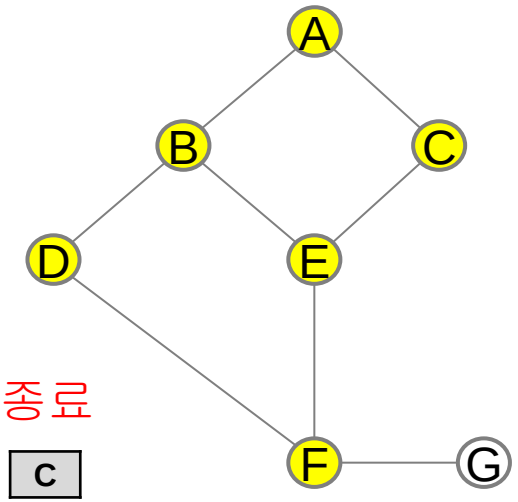
```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
       w 방문
       visited[w] = true
       push(w)
       v = w
       w = v의 미방문 인접정점

    v = stack.pop()
```

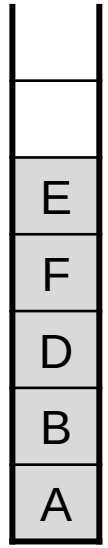


출력 : A – B – D – F – E – C

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )  # w 가 없으므로 while 종료
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```





v c

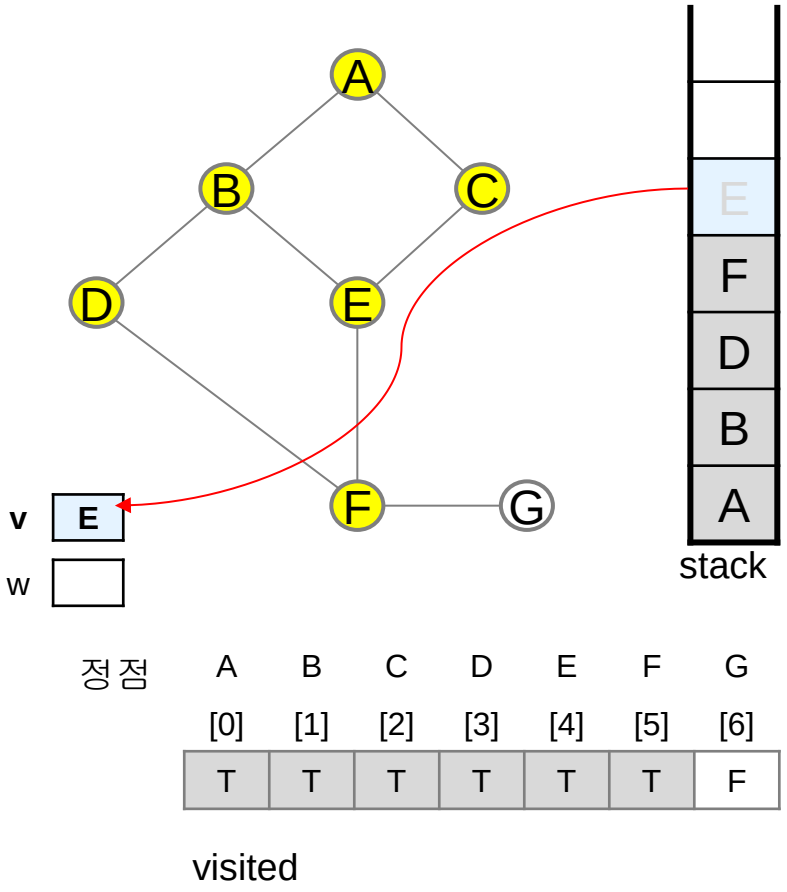
w

정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	T	F

출력 : A – B – D – F – E – C

DFS 알고리즘 동작 예

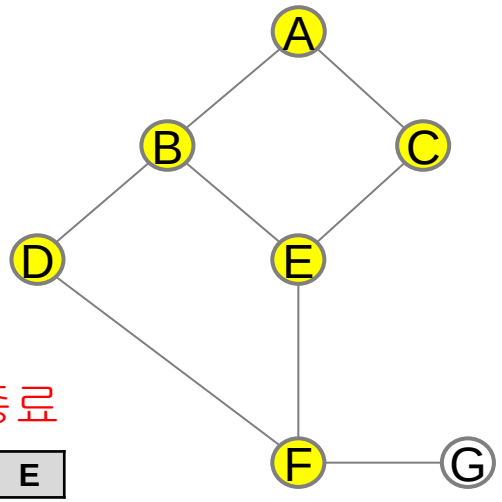
```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
       w 방문
       visited[w] = true
       push(w)
       v = w
       w = v의 미방문 인접정점

    v = stack.pop()
```

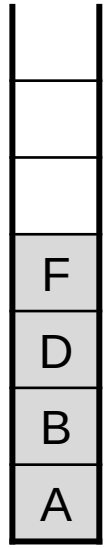


출력 : A – B – D – F – E – C

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w ) # w 가 없으므로 while 종료
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```





v E

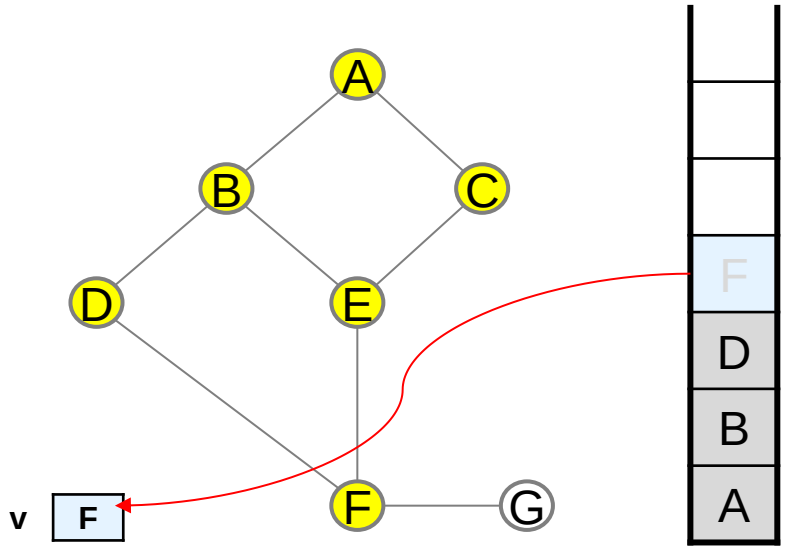
w

정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	T	F

출력 : A – B – D – F – E – C

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
       w 방문
       visited[w] = true
       push(w)
       v = w
       w = v의 미방문 인접정점

    v = stack.pop()
```

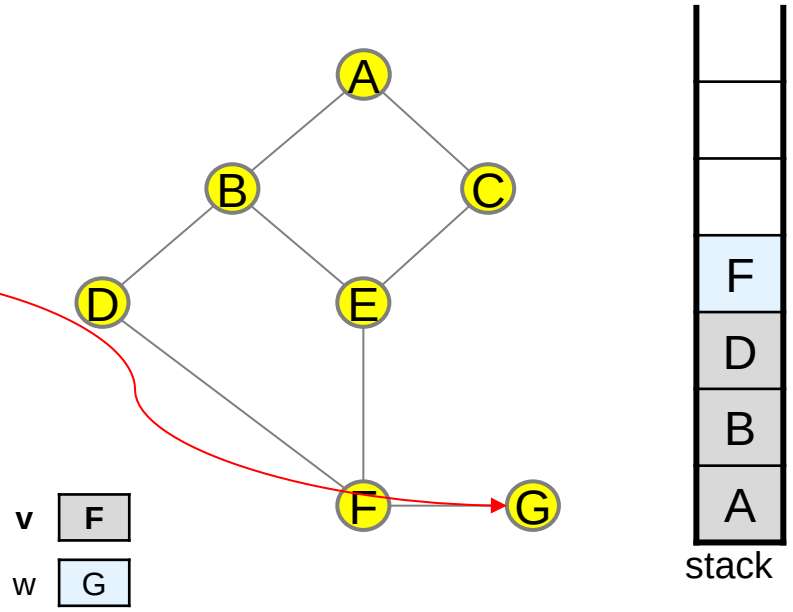


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	T	F

출력 : A – B – D – F – E – C

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```

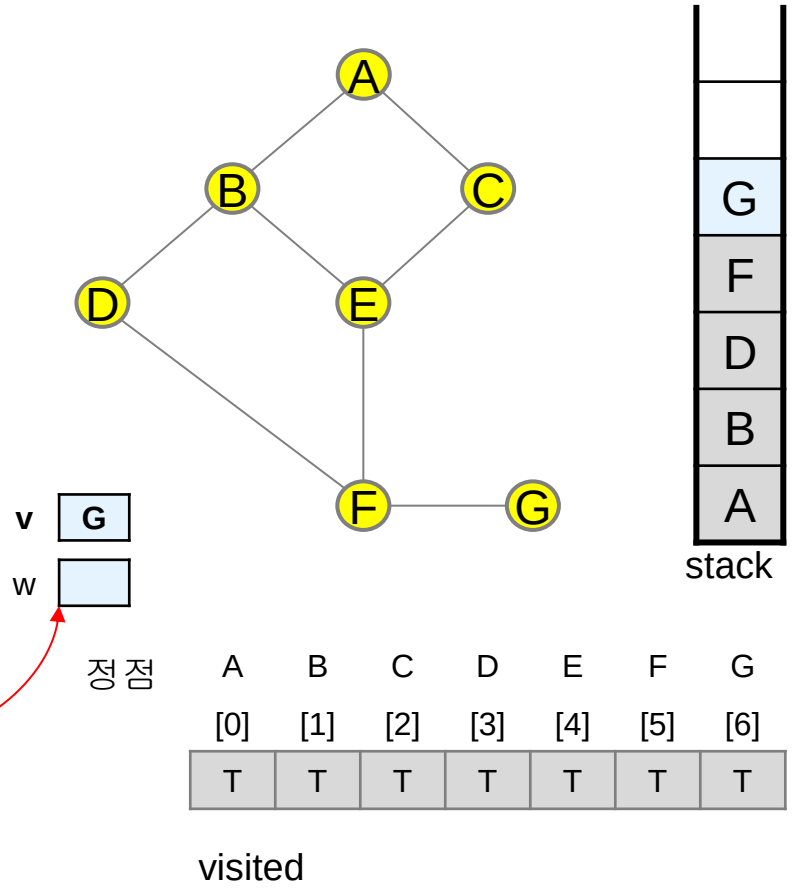


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	T	T

출력 : A – B – D – F – E – C – G

DFS 알고리즘 동작 예

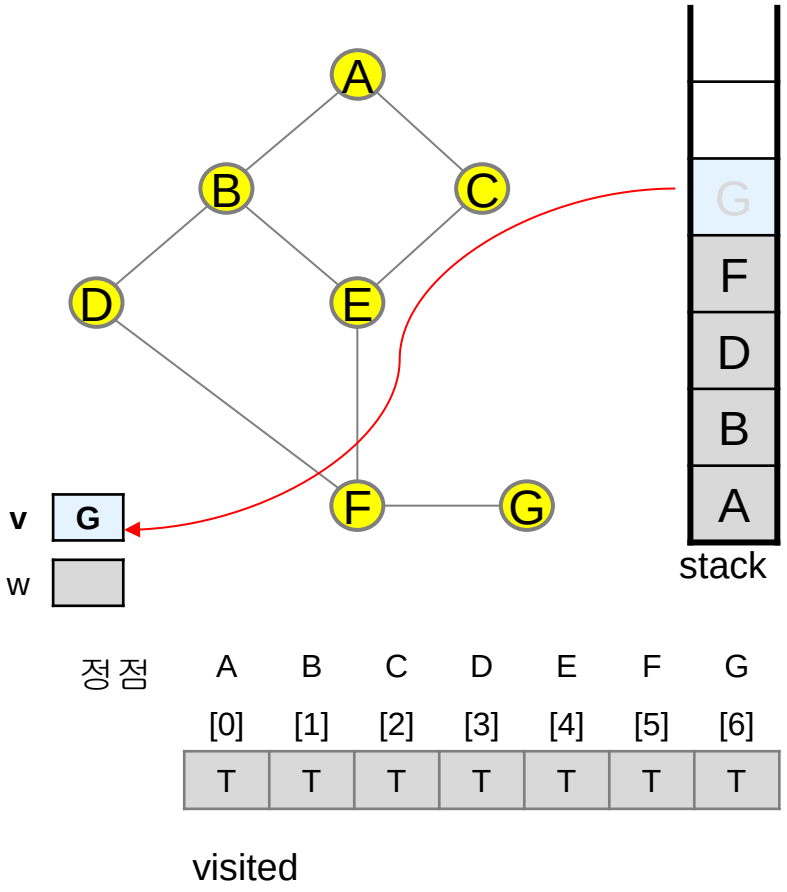
```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
       w 방문
       visited[w] = true
       push(w)
       v = w
       w = v의 미방문 인접정점

    v = stack.pop()
```



출력 : A – B – D – F – E – C – G

DFS 알고리즘 동작 예

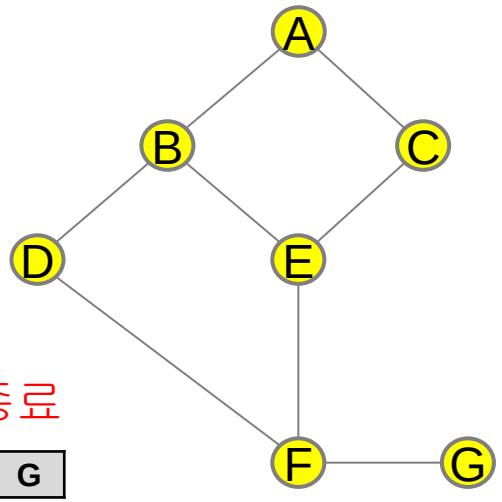
```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
       w 방문
       visited[w] = true
       push(w)
       v = w
       w = v의 미방문 인접정점

    v = stack.pop()
```



출력 : A – B – D – F – E – C – G

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w ) # w 가 없으므로 while 종료
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```

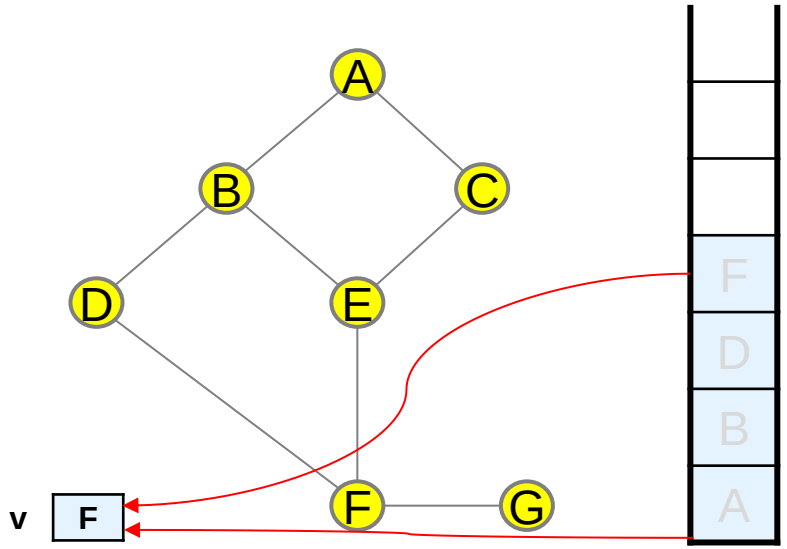


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	T	T

출력 : A – B – D – F – E – C – G

DFS 알고리즘 동작 예

```
DFS(v):
  v 방문 # 출력 처리
  visited[v] = true
  while(v):
    w = v의 미방문 인접정점
    if (w)
        push(v)
    while( w )
        w 방문
        visited[w] = true
        push(w)
        v = w
        w = v의 미방문 인접정점

    v = stack.pop()
```



정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	T	T

stack.pop() 결과 v 의 방문하지 않은 인접 정점 w 가 없으므로 스택이 빌 때까지 수행 후, 종료된다.

출력 : A – B – D – F – E – C – G

DFS 알고리즘 2

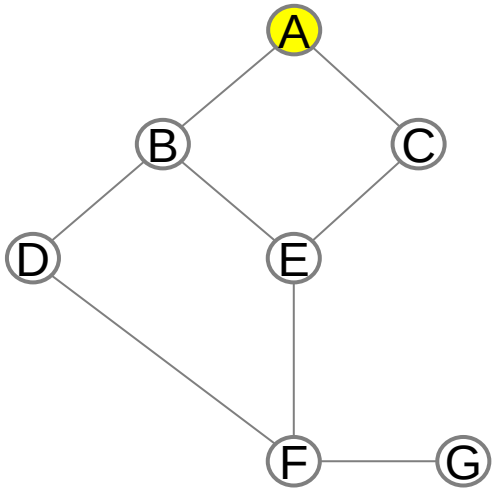
```
visited[] : 방문 여부 체크 배열
DFS(v):

    v 방문                  # 출력 or 계산 등의 처리
    visited[v]=1             # V 의 방문 표시

    for w in (v 의 인접정점)    # v 의 모든 인접 정점 w에 대해서
        if visited[w]            # w 가 방문하지 않았다면
            DFS(w)               # 정점 W 값으로 재귀 호출
```

DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	F	F	F	F	F	F

함수 호출 관계 : 상태 공간 트리



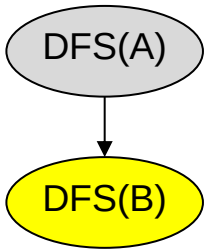
출력 : A

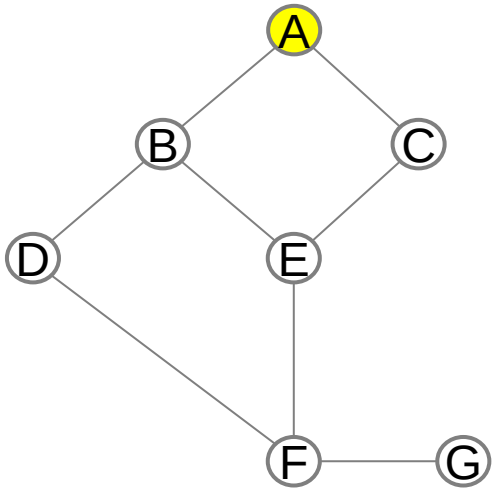
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점) # 인접 정점 B, C
          if visited[w]        # B 먼저 호출
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	F	F	F	F	F	F

함수 호출 관계 : 상태 공간 트리





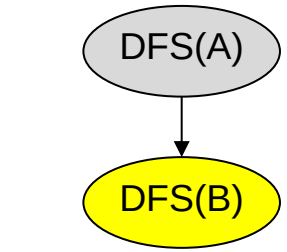
출력 : A

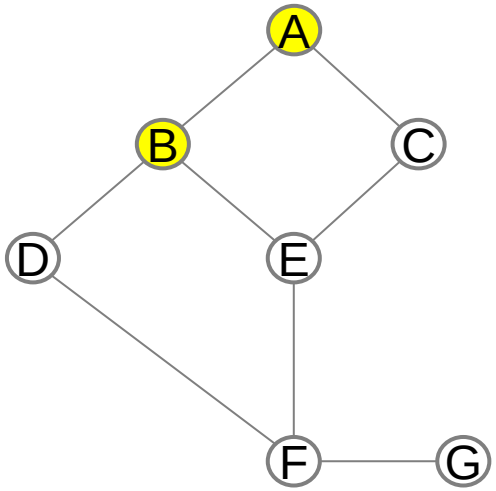
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	F	F	F	F	F

함수 호출 관계 : 상태 공간 트리





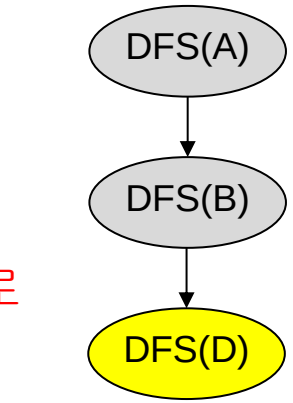
출력 : A - B

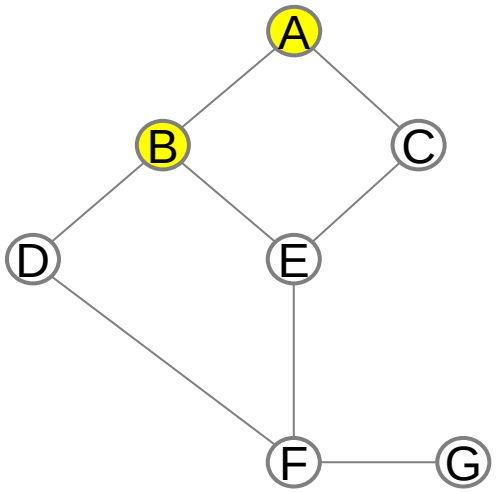
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	F	F	F	F	F

함수 호출 관계 : 상태 공간 트리





인접 정점 A, D, E
A 는 이미 방문했으므로
D 먼저 호출

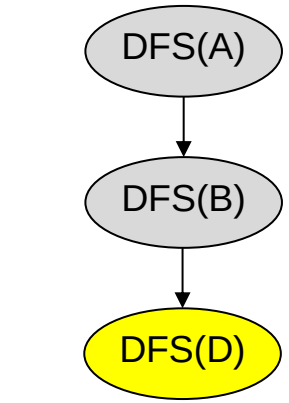
출력 : A - B

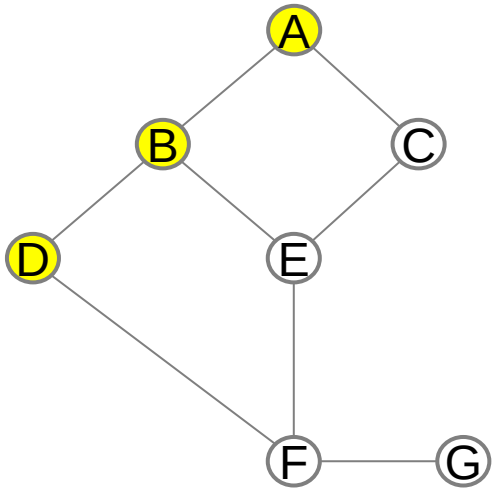
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	F	T	F	F	F

함수 호출 관계 : 상태 공간 트리





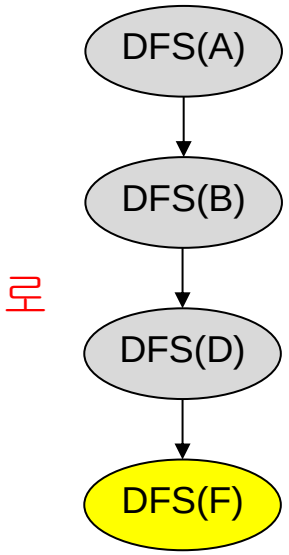
출력 : A – B - D

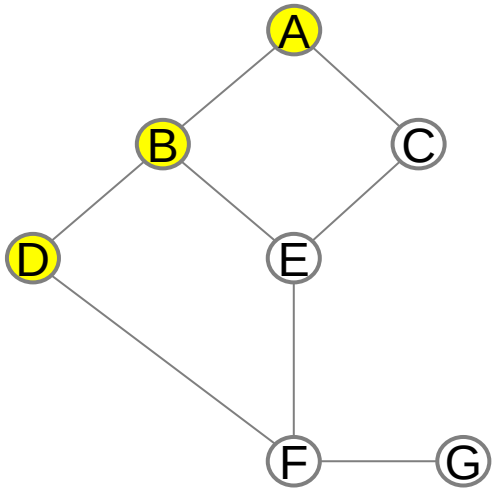
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	F	T	F	F	F

함수 호출 관계 : 상태 공간 트리





인접 정점 B, F
B는 이미 방문했으므로
F 호출

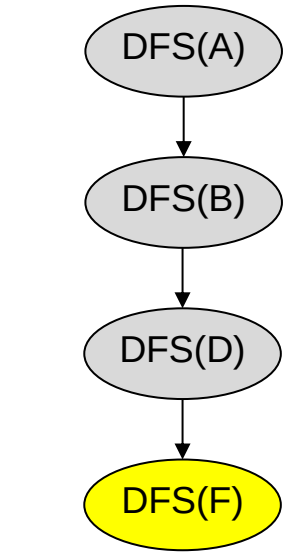
출력 : A – B - D

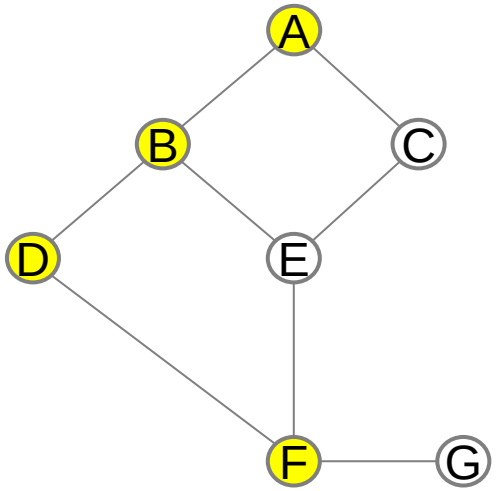
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	F	T	F	T	F

함수 호출 관계 : 상태 공간 트리





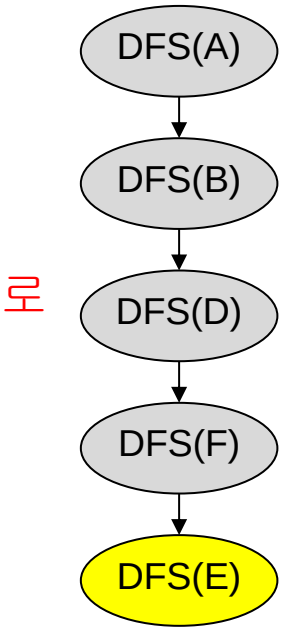
출력 : A – B – D - F

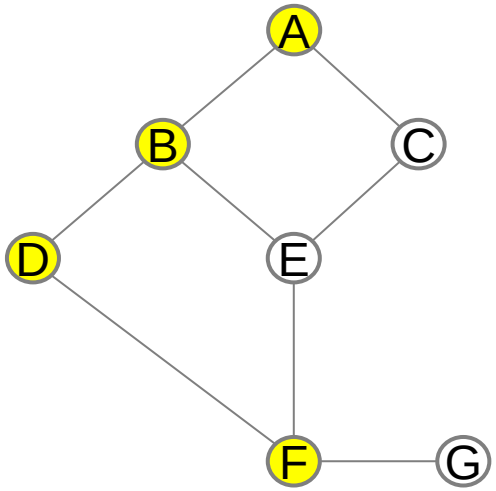
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	F	T	F	T	F

함수 호출 관계 : 상태 공간 트리





인접 정점 D, E, G
D는 이미 방문했으므로
E 호출

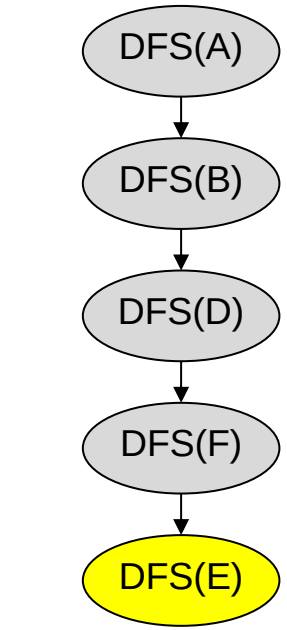
출력 : A – B - D - F

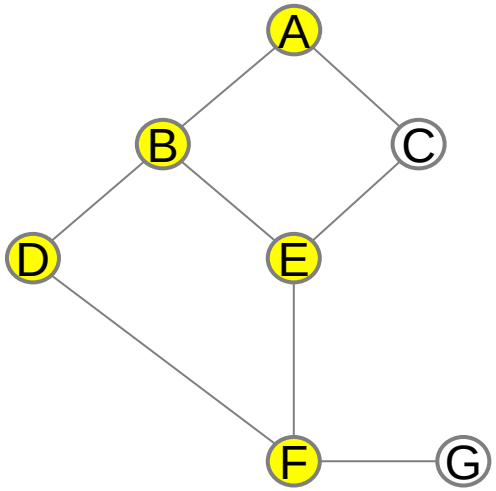
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	F	T	T	T	F

함수 호출 관계 : 상태 공간 트리





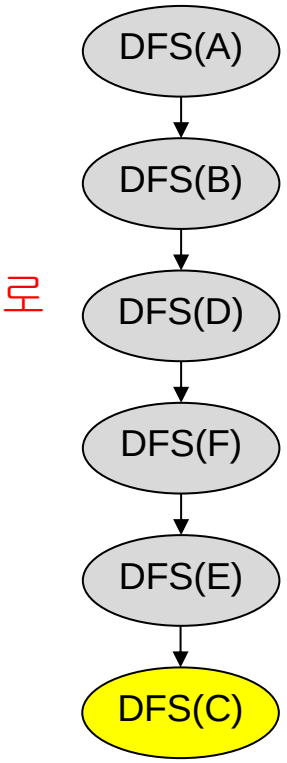
출력 : A – B - D – F - E

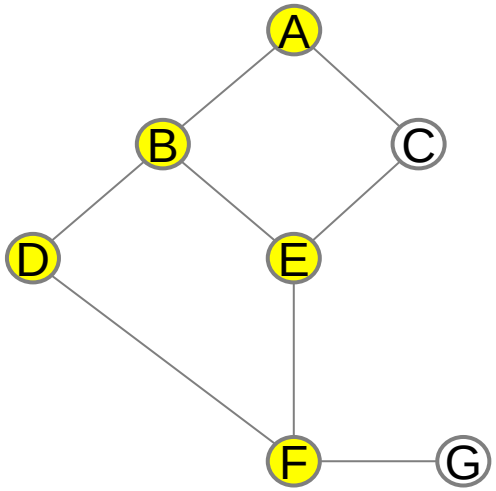
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	F	T	T	T	F

함수 호출 관계 : 상태 공간 트리





인접 정점 B, C, F
B는 이미 방문했으므로
C 호출

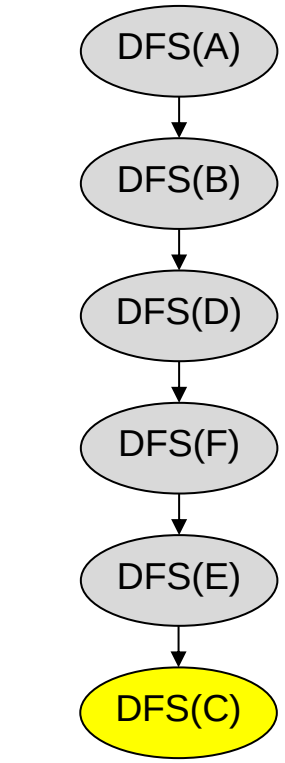
출력 : A – B - D – F - E

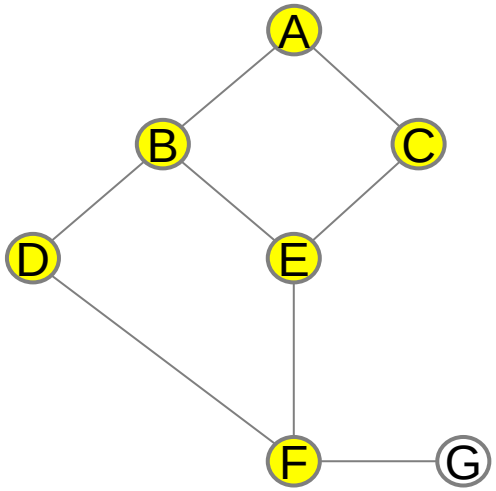
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	F

함수 호출 관계 : 상태 공간 트리





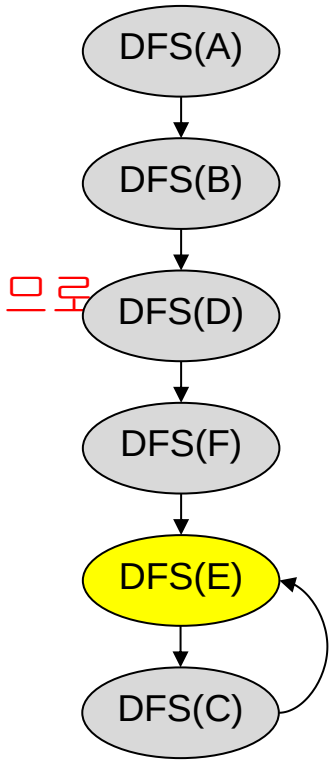
출력 : A – B - D – F – E - C

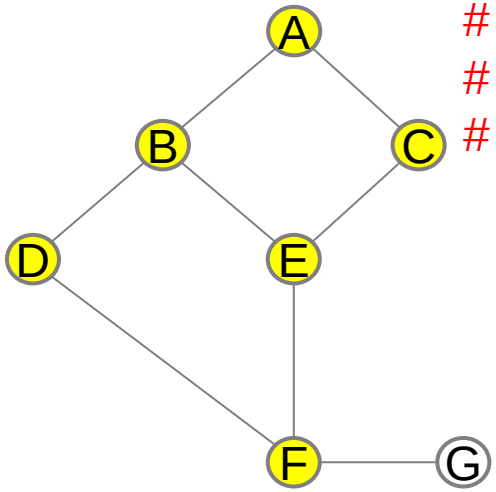
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	F

함수 호출 관계 : 상태 공간 트리





인접 정점 A, E
A, E 모두 이미 방문했으므로
리턴

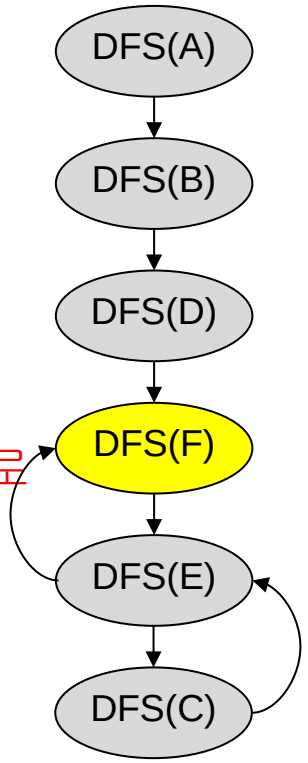
출력 : A – B - D – F – E - C

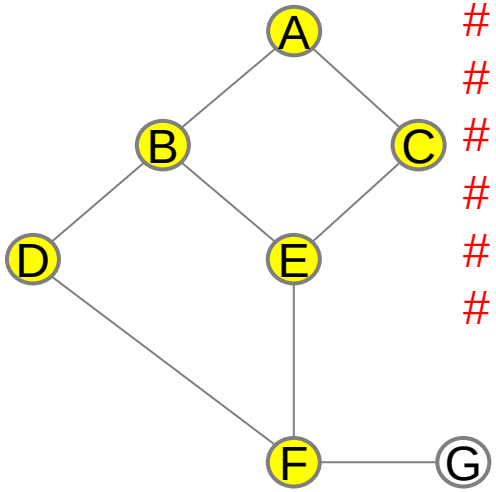
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	F

함수 호출 관계 : 상태 공간 트리





인접 정점 B, C, F
B 는 이미 방문하였고,
C는 방문하고 리턴한 상태
F는 이미 방문했으므로
더 이상 인접 정점 없으므로
리턴

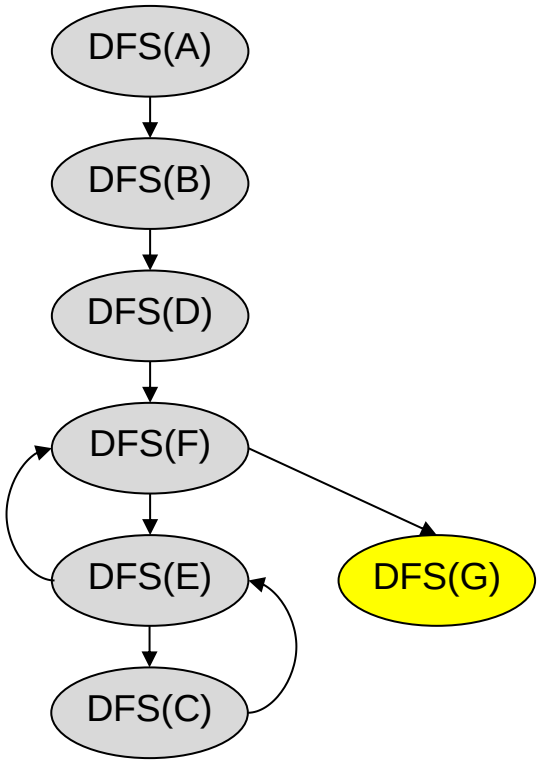
출력 : A – B - D – F – E - C

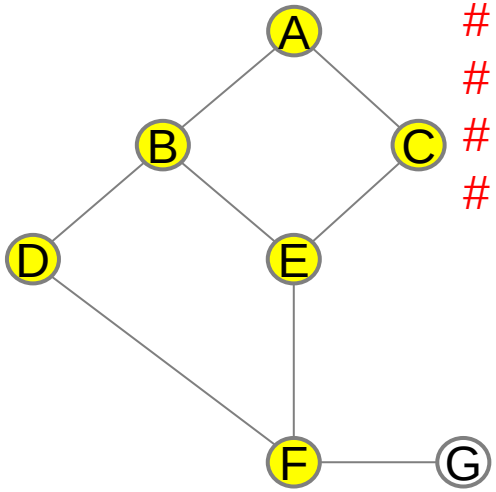
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	F

함수 호출 관계 : 상태 공간 트리





인접 정점 D, E, G
D 는 이미 방문하였고,
E는 방문하고 리턴한 상태
G 호출

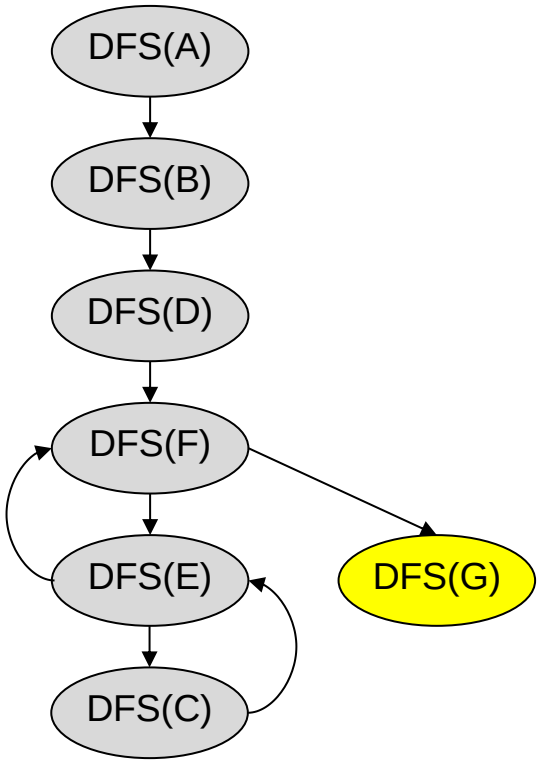
출력 : A – B - D – F – E - C

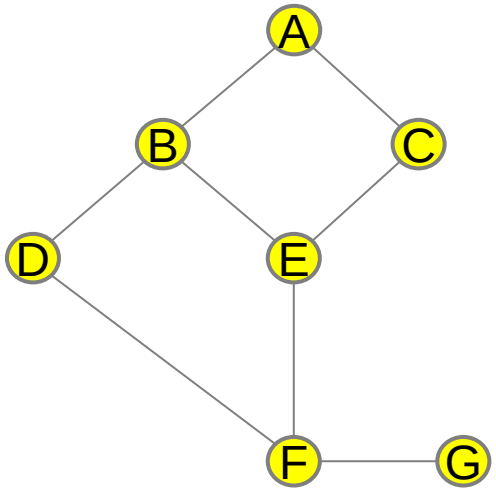
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

함수 호출 관계 : 상태 공간 트리





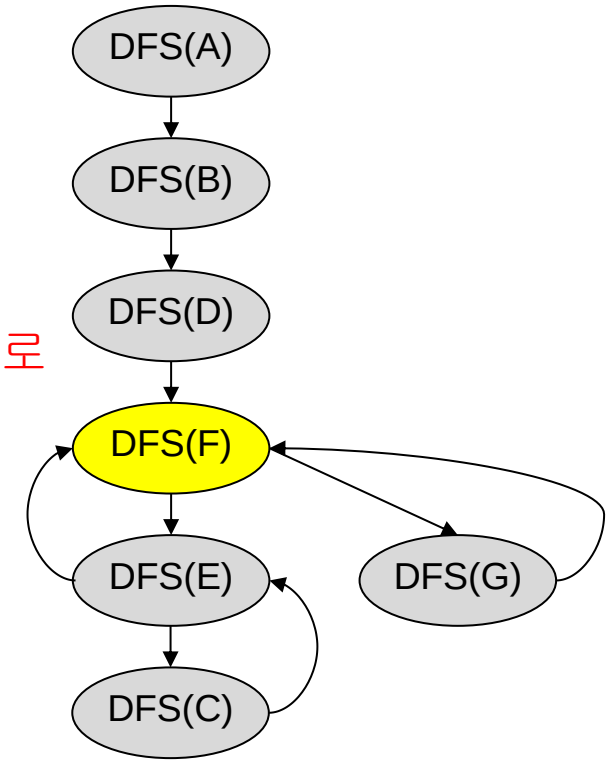
출력 : A – B - D – F – E – C - G

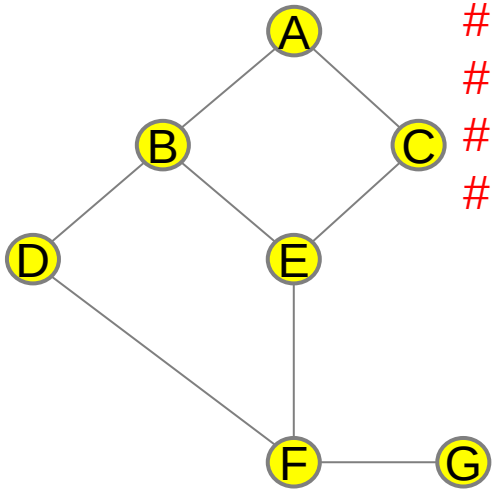
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

함수 호출 관계 : 상태 공간 트리






```
# 인접 정점 F
# F 는 이미 방문하였고,
# 더 이상 인접 정점 없으므로
# 리턴
```

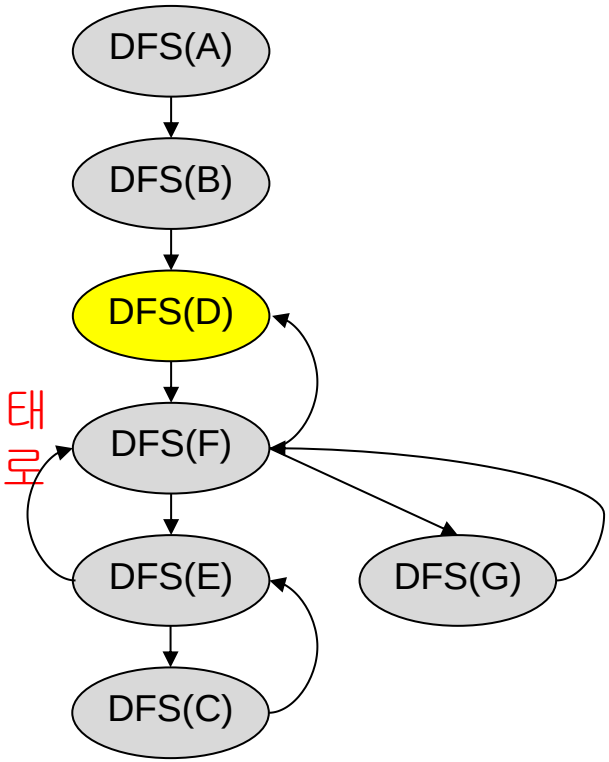
출력 : A – B - D – F – E – C - G

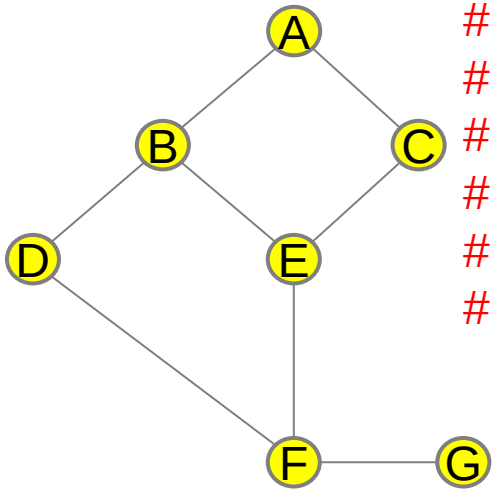
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

함수 호출 관계 : 상태 공간 트리





인접 정점 D, E, G
D 는 이미 방문하였고,
E 는 호출 후 리턴,
G 에서 호출 후 리턴한 상태
더 이상 인접 정점 없으므로
리턴

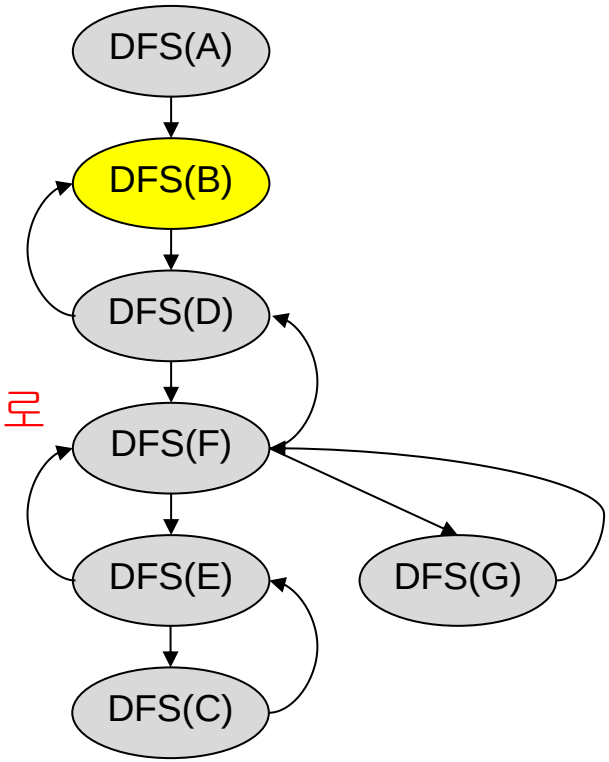
출력 : A – B - D – F – E – C - G

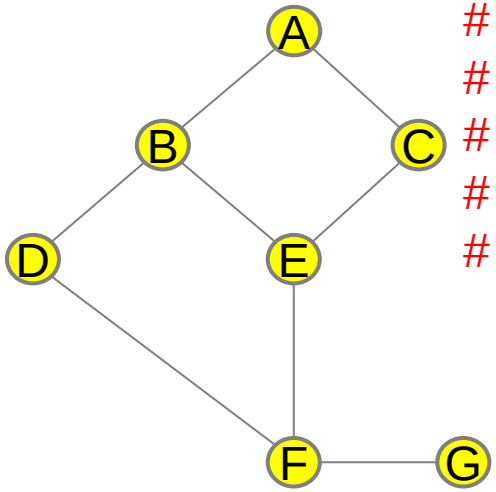
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

함수 호출 관계 : 상태 공간 트리





인접 정점 B, F
B 는 이미 방문하고 진입,
F 는 호출 후 리턴,
더 이상 인접 정점 없으므로
리턴

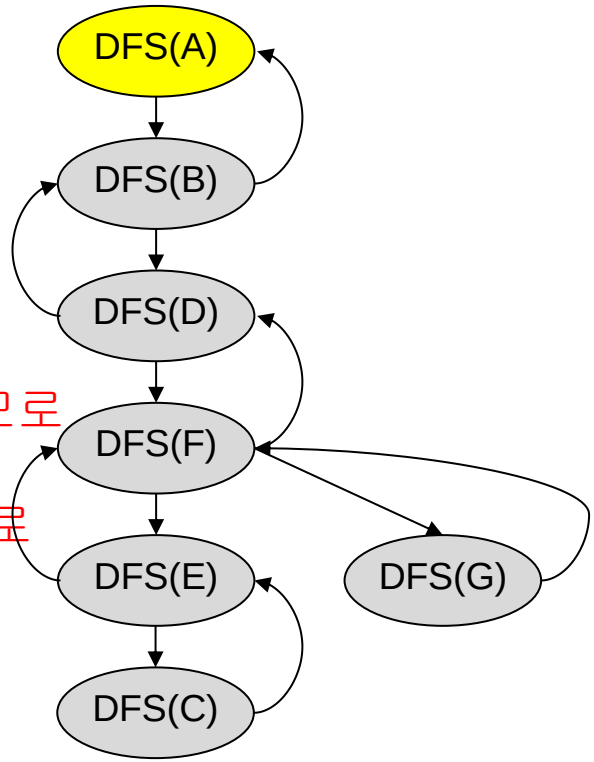
출력 : A – B - D – F – E – C - G

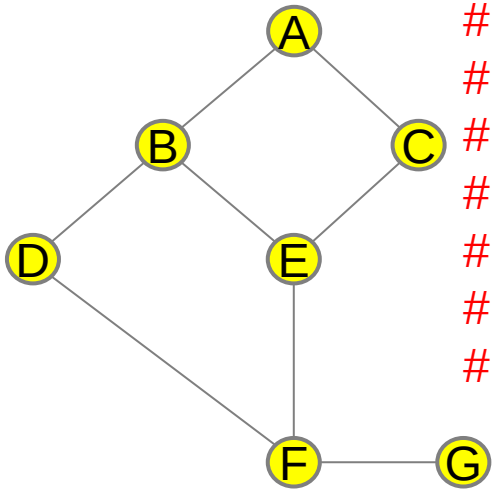
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

함수 호출 관계 : 상태 공간 트리





인접 정점 A, D, E
A 는 이미 방문하고 진입,
D 는 호출 후 리턴,
E 은 이미 방문한 상태이므로
E 를 호출하지 않는다.
더 이상 인접 정점 없으므로
리턴

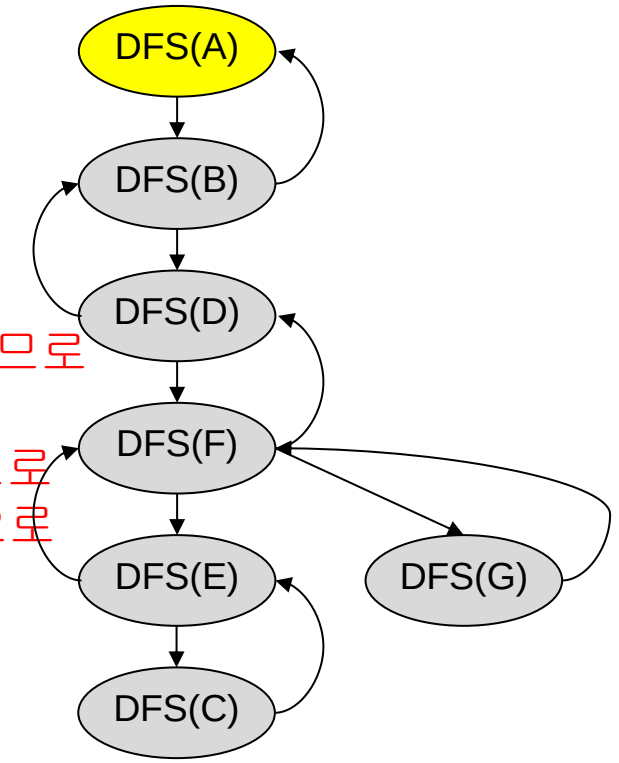
출력 : A – B - D – F – E – C - G

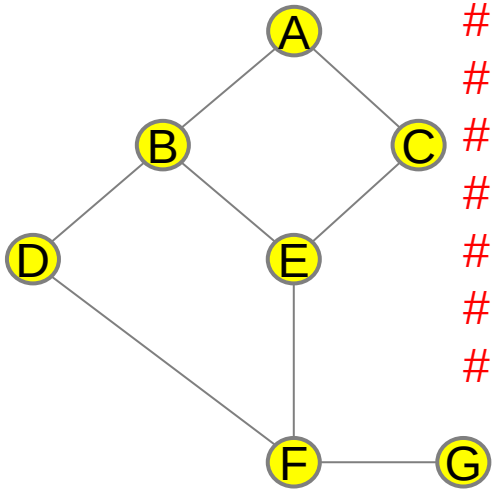
DFS 2알고리즘 동작 예

```
DFS(v):
      v 방문   # 출력 처리
      visited[v]=1

      for w in (v 의 인접정점)
          if visited[w]
              DFS(w)
```

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

함수 호출 관계 : 상태 공간 트리





인접 정점 B, C
B 는 호출 후 리턴,
C 은 이미 방문한 상태이므로
C 를 호출하지 않는다.
더 이상 인접 정점 없으므로
DFS(A) 를 호출한 지점으로
리턴하고 종료

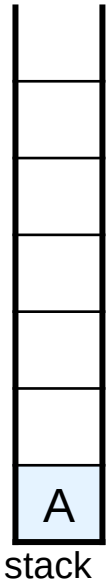
출력 : A – B - D – F – E – C - G

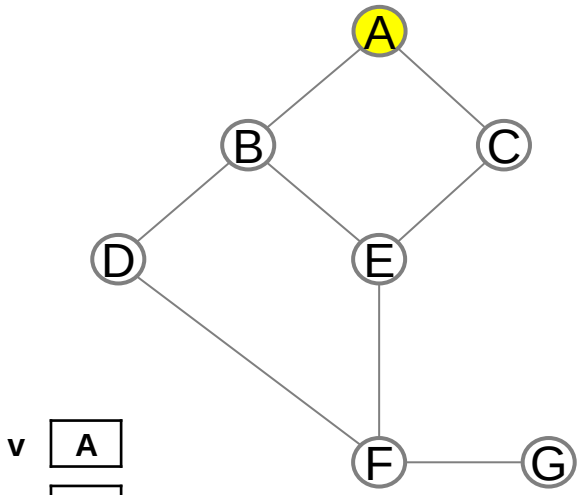
DFS 알고리즘 3

```
visited[] : 방문 여부 체크 배열, stack[]: 방문했던 정점들 저장
DFS(v):
     stack.push(v)                  # 스택에 시작 정점 저장

     while not stack.isEmpty()
        v = stack.pop()          # 최근 방문한 정점을 가져오게 된다.

        if Not visited[v]              # v 를 방문하지 않았다면
               visited[v]=1            # V 의 방문 표시
               v 방문                   # 출력 or 계산 등의 처리

        for w in (v 의 인접정점)      # v 의 모든 인접 정점에 대해서
           if visited[w]                 # w 가 방문하지 않았다면
               push(w)                   # 스택에 정점 W 저장
```

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```





v A

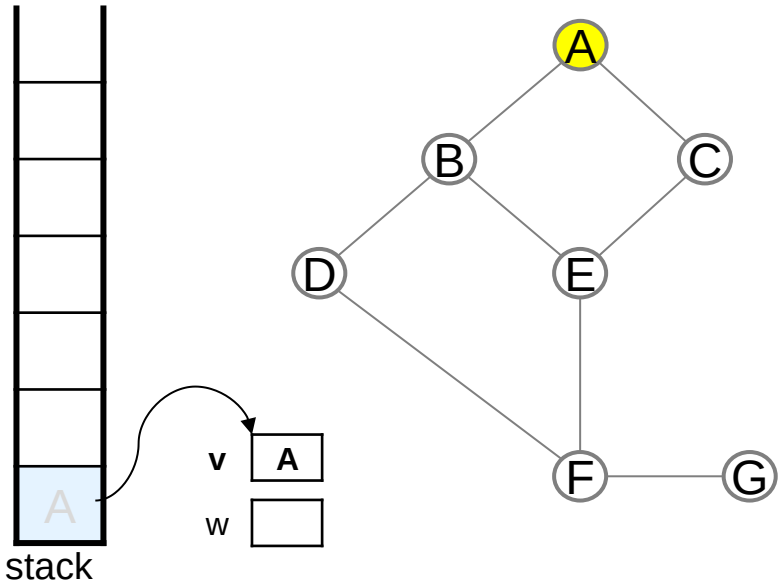
w

정점	A	B	C	D	E	F	G
visited	F	F	F	F	F	F	F

출력 :

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```

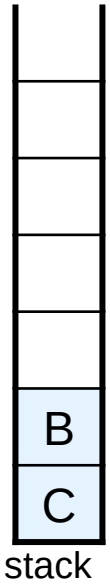


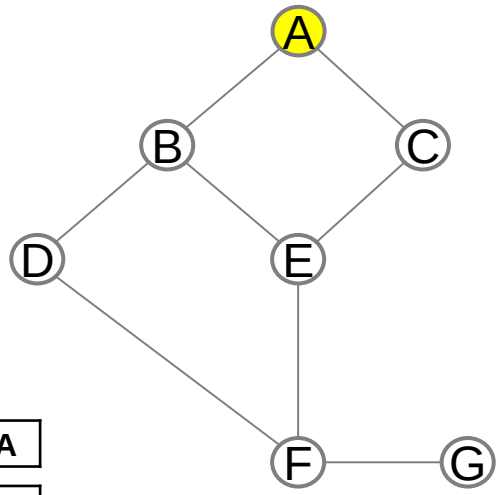
정점	A	B	C	D	E	F	G
visited	T	F	F	F	F	F	F

출력 : A

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```





v A

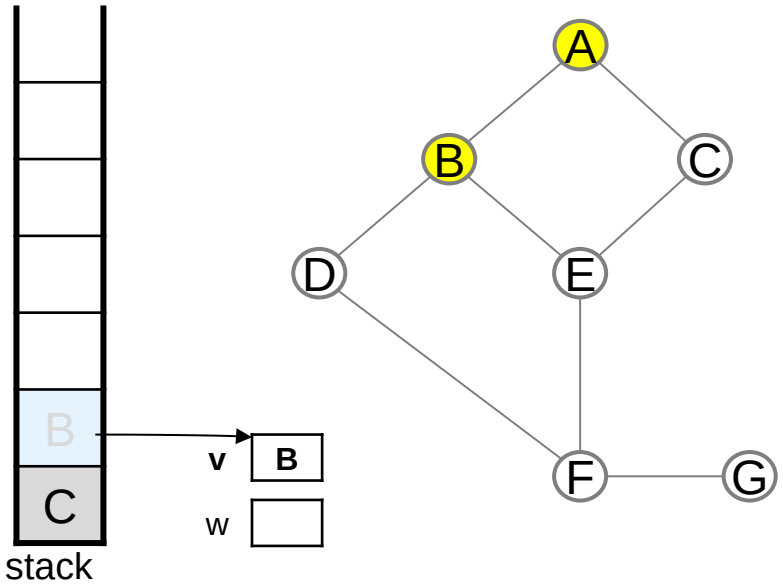
w B, C

정점	A	B	C	D	E	F	G
visited	T	F	F	F	F	F	F

출력 : A

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문  # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```

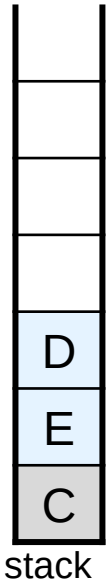


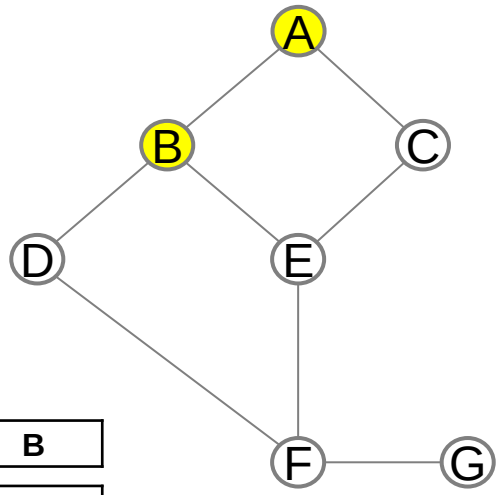
정점	A	B	C	D	E	F	G
visited	T	T	F	F	F	F	F

출력 : A - B

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```





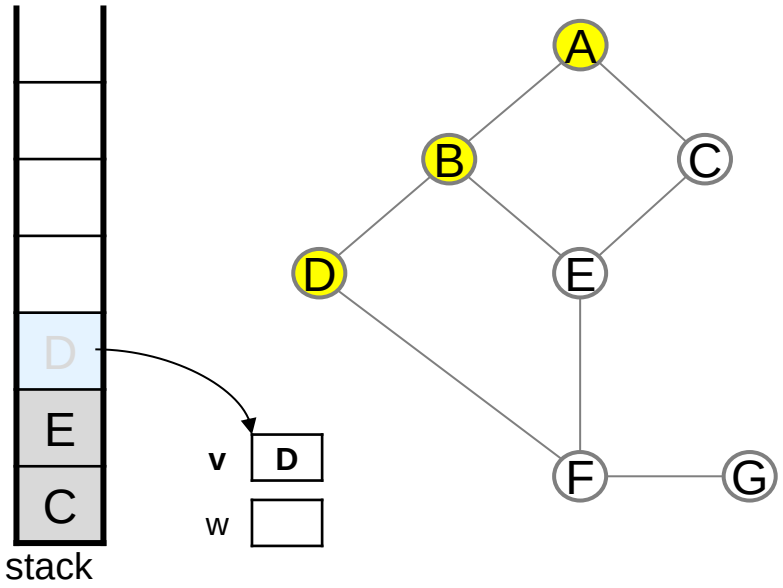
v	B
w	A, D, E

정점	A	B	C	D	E	F	G
visited	T	T	F	F	F	F	F

출력 : A - B

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```

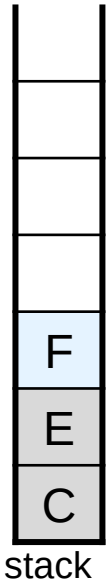


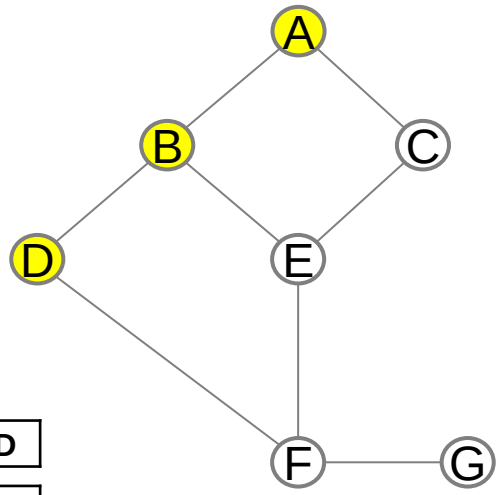
정점	A	B	C	D	E	F	G
visited	T	T	F	T	F	F	F

출력 : A – B - D

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```





v D

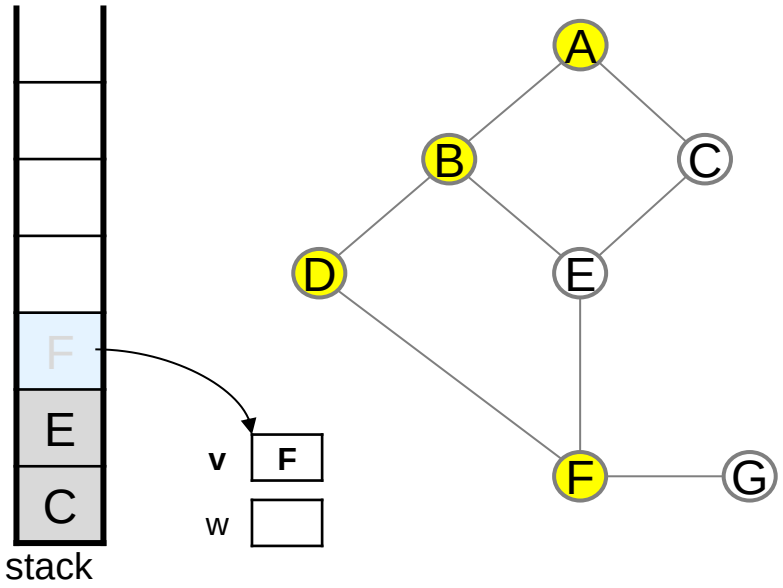
w B, F

정점	A	B	C	D	E	F	G
visited	T	T	F	T	F	F	F

출력 : A – B - D

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```

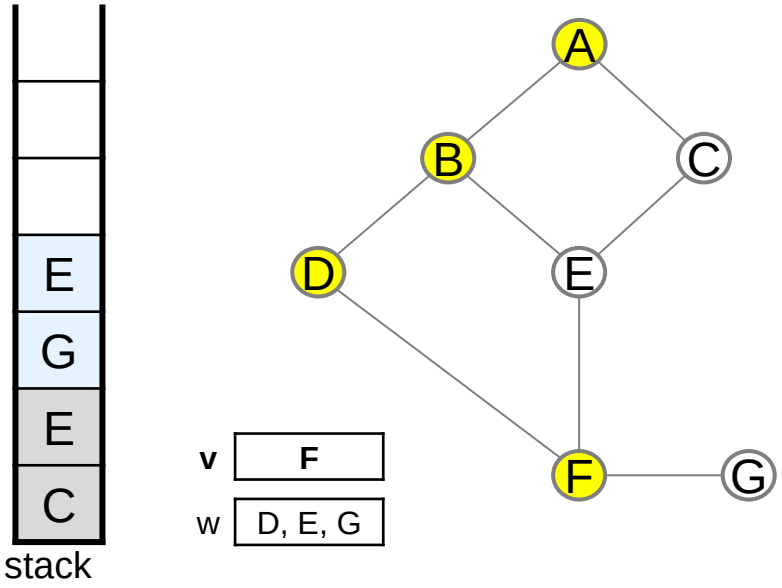


정점	A	B	C	D	E	F	G
visited	T	T	F	T	F	T	F

출력 : A – B – D - F

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```

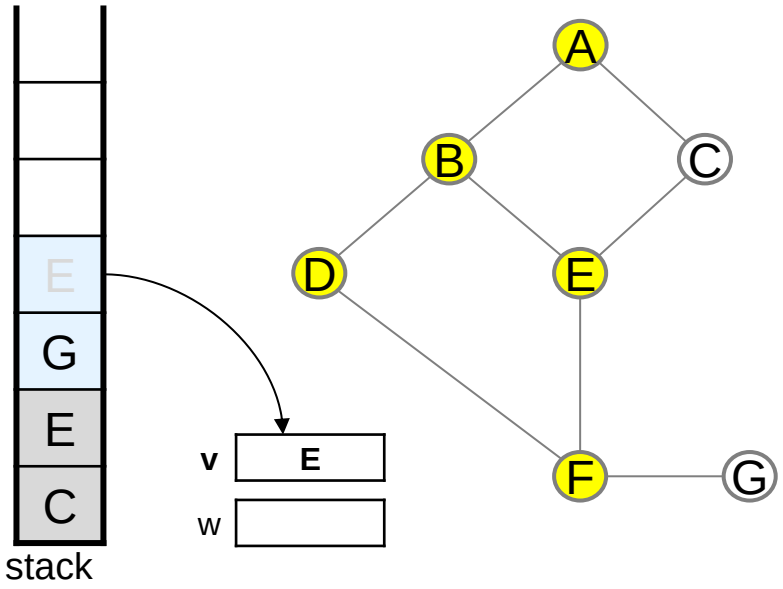


정점	A	B	C	D	E	F	G
visited	T	T	F	T	F	T	F

출력 : A – B – D - F

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```

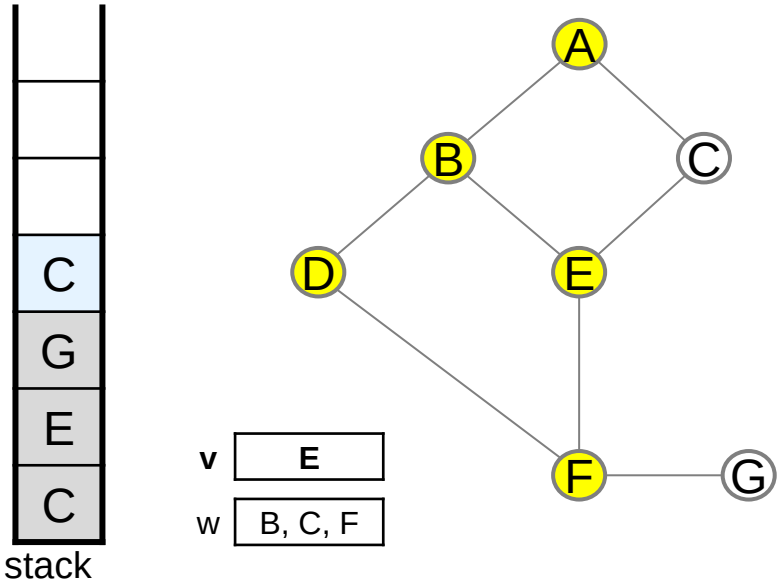


정점	A	B	C	D	E	F	G
visited	T	T	F	T	T	T	F

출력 : A – B – D – F - E

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```

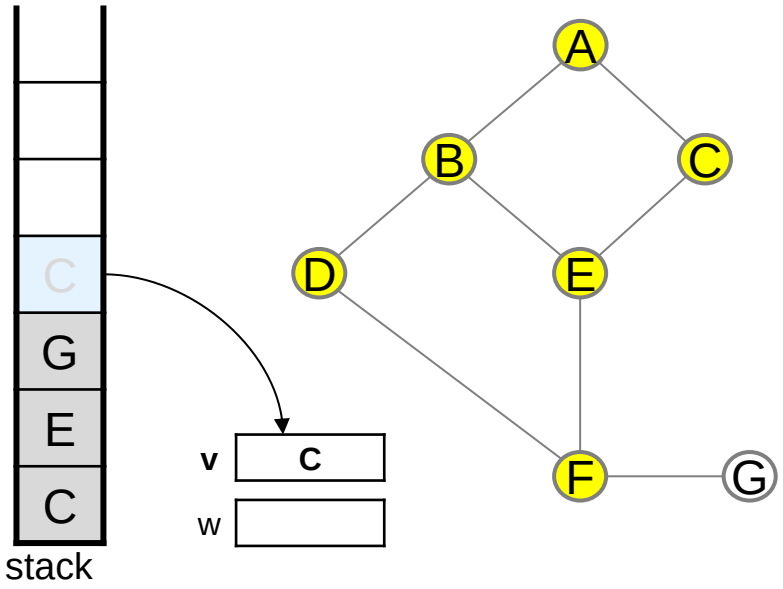


정점	A	B	C	D	E	F	G
visited	T	T	F	T	T	T	F

출력 : A – B – D – F - E

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```

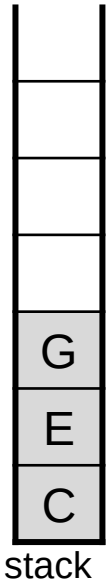


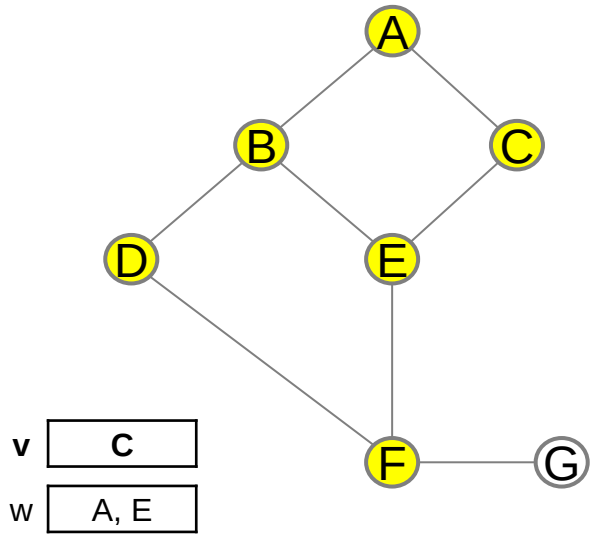
정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	F

출력 : A – B – D – F – E - C

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```





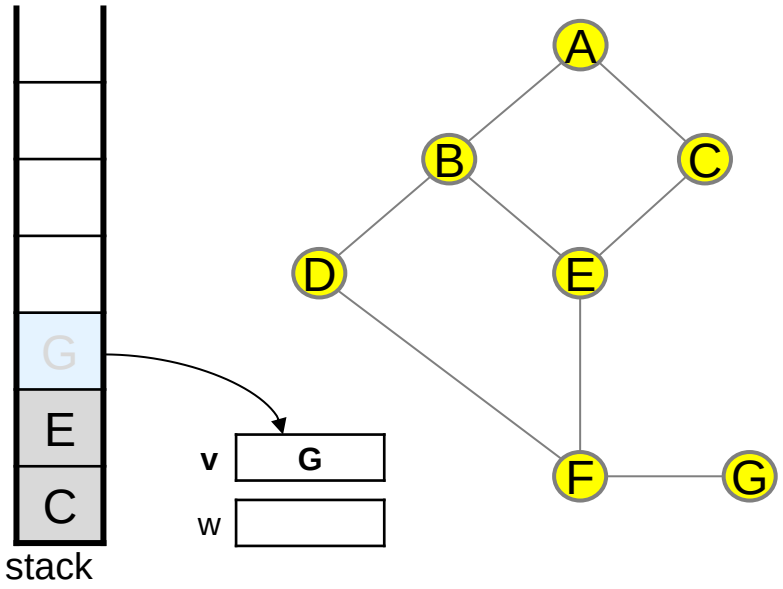
A, E 모두 방문한 정점이므로 push 하지 않는다.

출력 : A – B – D – F – E - C

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	F

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```

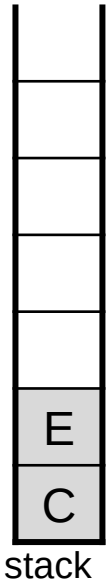


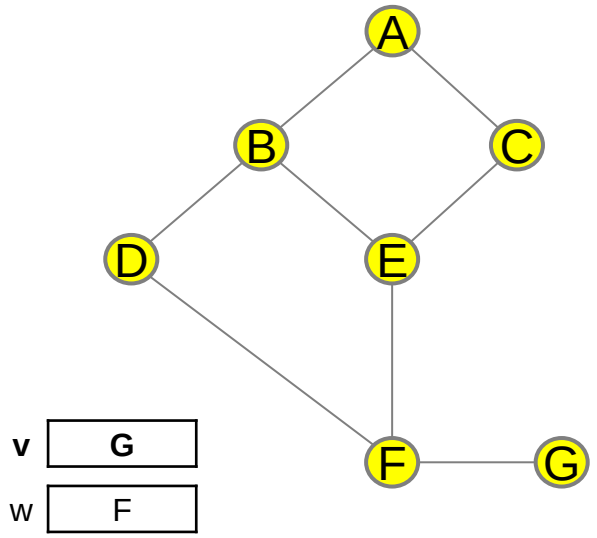
정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

출력 : A – B – D – F – E – C – G

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```





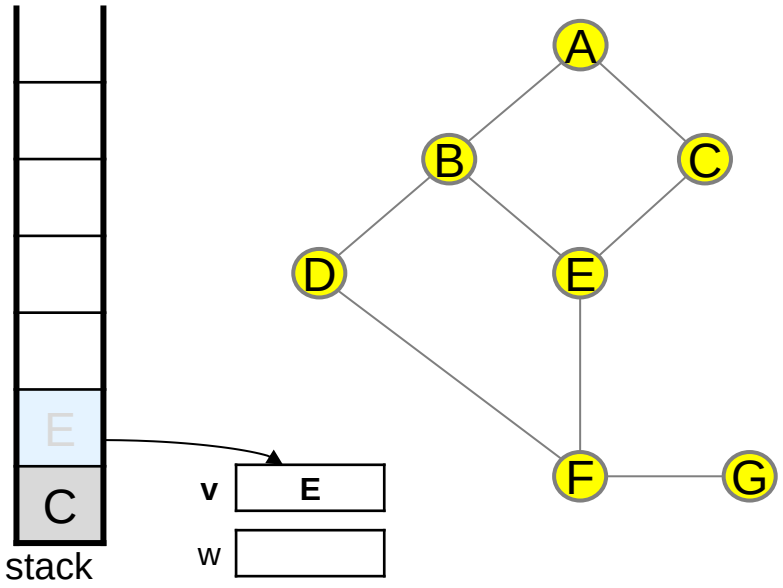
F 는 방문한 정점이므로
push 하지 않는다.

출력 : A – B – D – F – E – C – G

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```



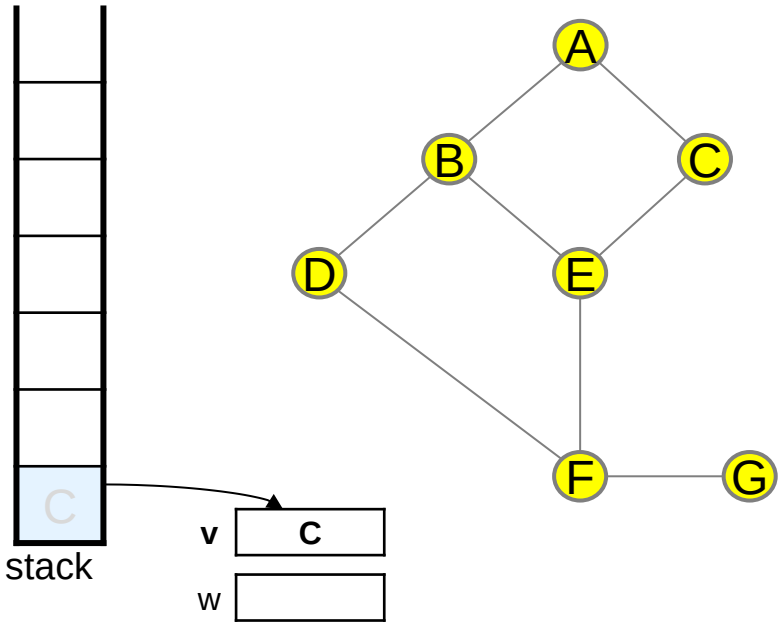
E 는 방문한 정점이므로 skip 한다.

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

출력 : A – B – D – F – E – C – G

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```



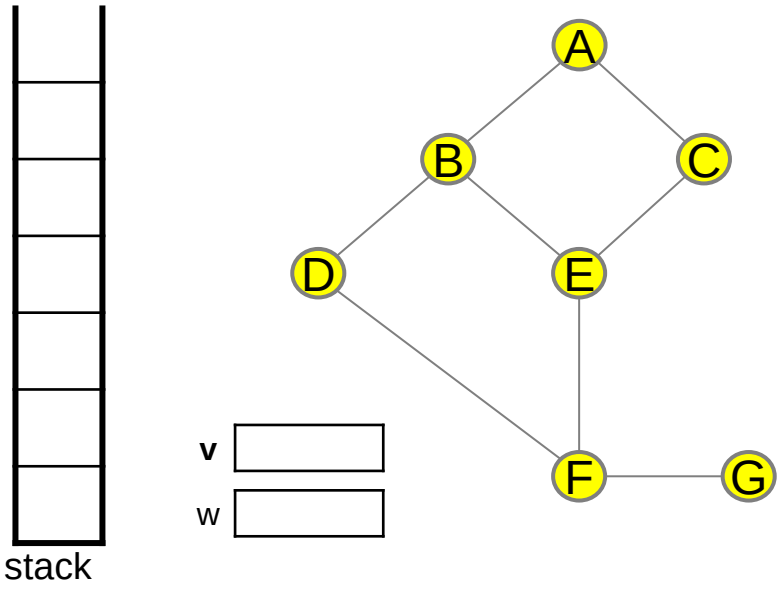
C 는 방문한 정점이므로 skip 한다.

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

출력 : A – B – D – F – E – C – G

DFS3 알고리즘 동작

```
DFS(v):
 stack.push(v)

 while not stack.isEmpty()
   v = stack.pop()

   if Not visited[v]
      visited[v]=1
      v 방문   # 출력

      for w in (v 의 인접정점)
         if visited[w]
               push(w)
```



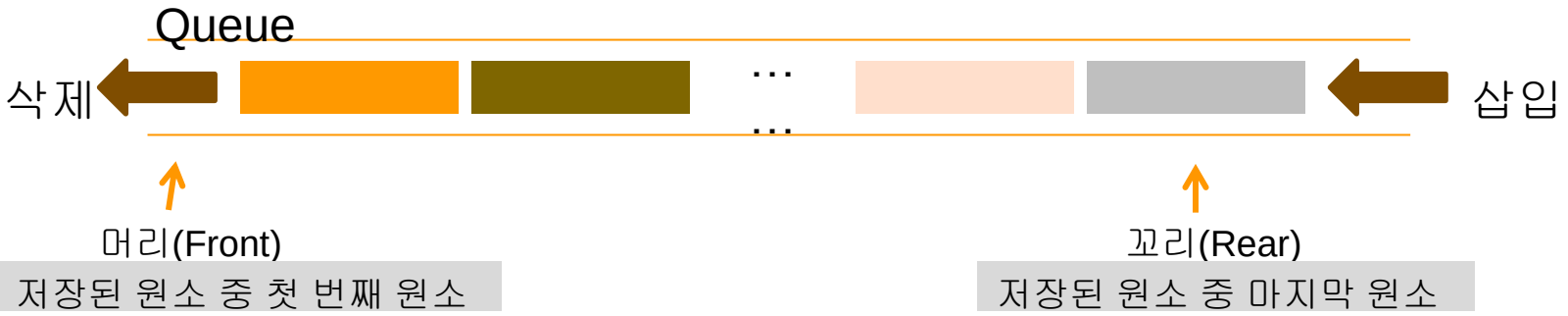
stack 이 비었으므로 종료한다.

출력 : A – B – D – F – E – C – G

정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

큐(Queue)

- 스택과 마찬가지로 삽입과 삭제의 위치가 제한적인 자료구조
- 선입선출구조(FIFO : First In First Out)
  - 큐에 삽입한 순서대로 원소가 저장되어, 가장 먼저 삽입(First In)된 원소는 가장 먼저 삭제(First Out)된다.
- 큐의 기본 연산
  - enQueue(item) : 큐의 뒤쪽(rear 다음)에 원소를 삽입하는 연산
  - deQueue() : 큐의 앞쪽(front)에서 원소를 삭제하고 반환하는 연산

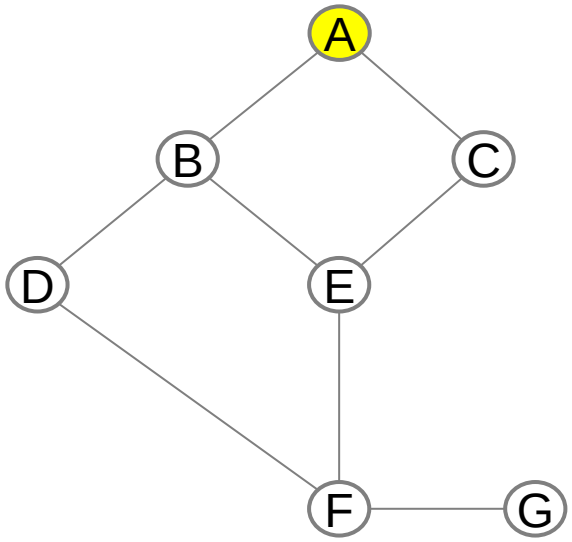


BFS 알고리즘

```
def BFS(v) :                          # 그래프 G, 탐색 시작점 v
    visited = [0]*n                     # n : 정점의 개수
    queue = []                          # 큐 생성
    queue.append(v)                     # 시작점 v를 큐에 삽입
    visited[v] = True                   # v 를 방문한 것으로 처리
    visit(v)                            # 출력 or 계산 등의 처리
    while queue :                       # 큐가 비어있지 않은 경우
        t = queue.popleft()             # 큐의 첫 번째 원소 반환
        for u in G[t] :               # t와 연결된 모든 정점에 대해
            if not visited[u] :       # 방문되지 않은 곳이라면
                queue.append(u)       # 큐에 넣기
                visited[u] = True     # 방문한 것으로 표시
                visit(u)              # 출력 or 계산 등의 처리
```

BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



정점	A	B	C	D	E	F	G
visited	T	F	F	F	F	F	F

출력 : A

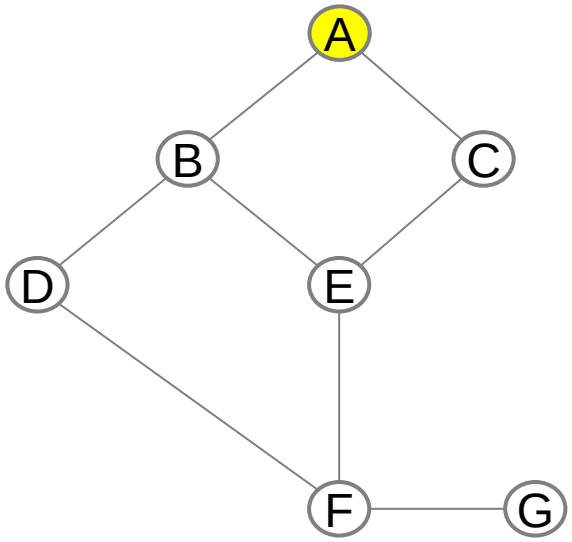
t

u

Queue | A | | | | | | |

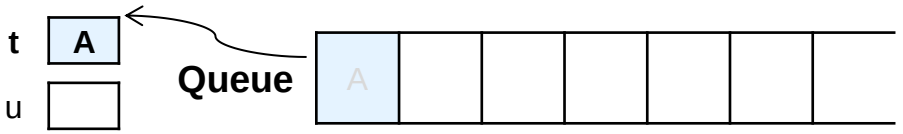
BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



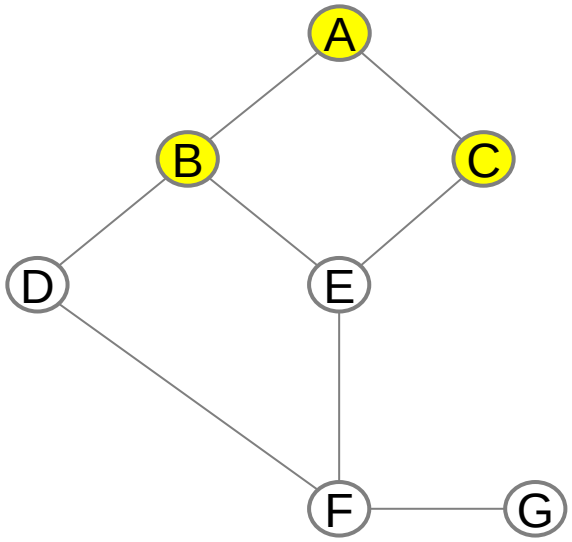
정점	A	B	C	D	E	F	G
visited	T	F	F	F	F	F	F

출력 : A



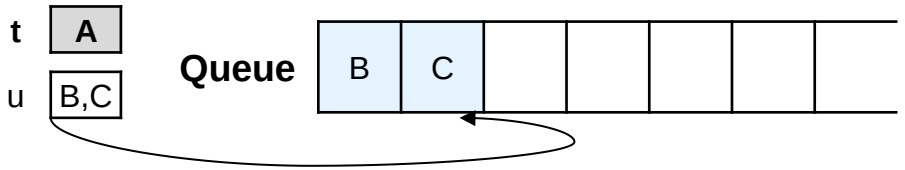
BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



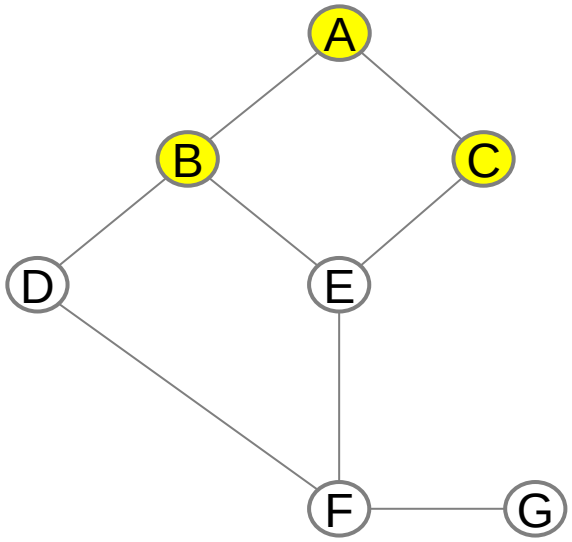
정점	A	B	C	D	E	F	G
visited	T	T	T	F	F	F	F

출력 : A – B – C



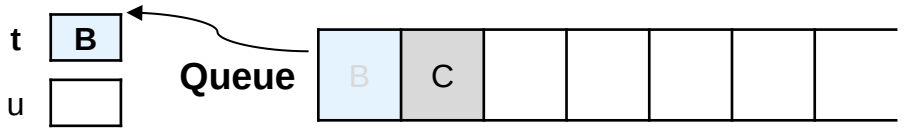
BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



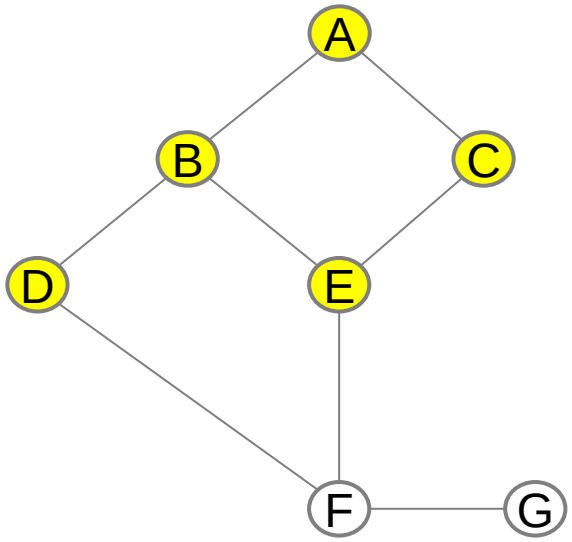
정점	A	B	C	D	E	F	G
visited	T	T	T	F	F	F	F

출력 : A – B – C



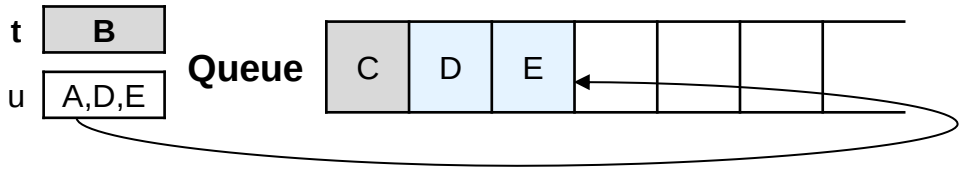
BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



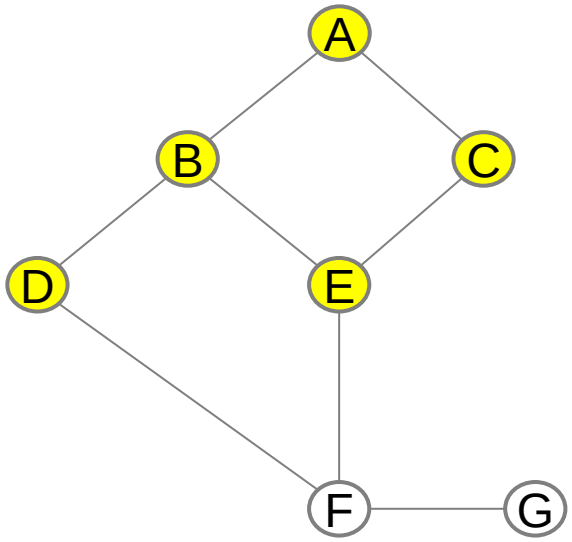
정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	F	F

출력 : A – B – C – D – E



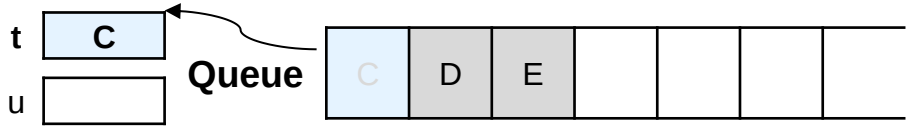
BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



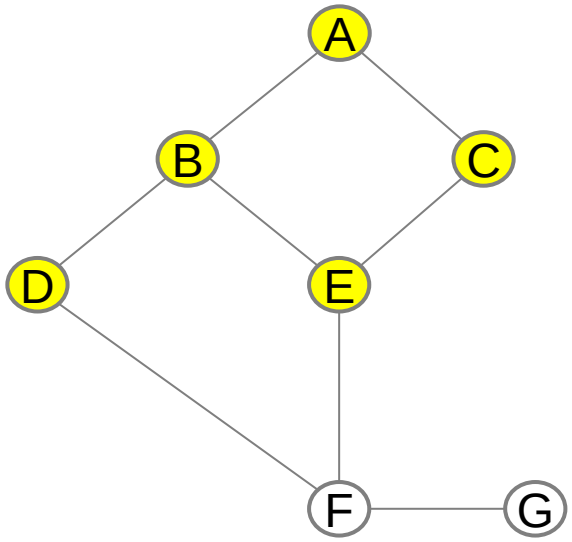
정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	F	F

출력 : A – B – C – D – E



BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	F	F

출력 : A – B – C – D – E

t	C
u	A,E

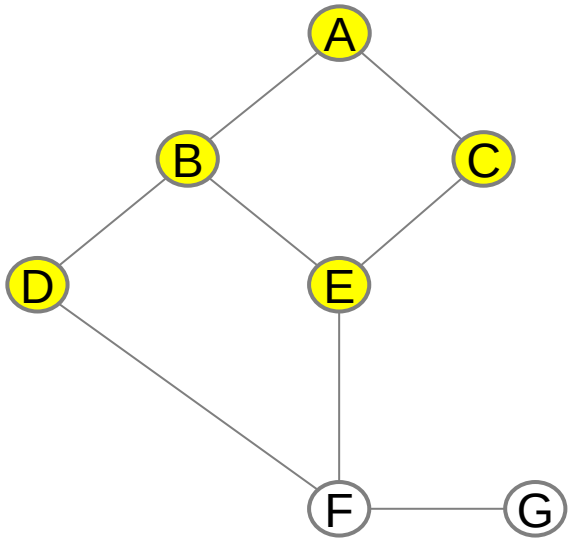
Queue

D	E					

C의 인접정점 A,E는 모두 방문했으므로 큐에 추가 안 함.

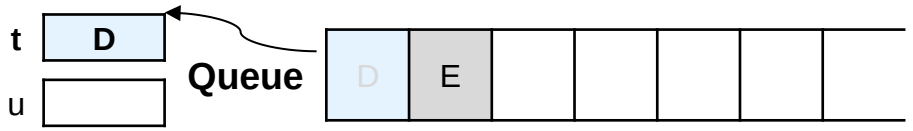
BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



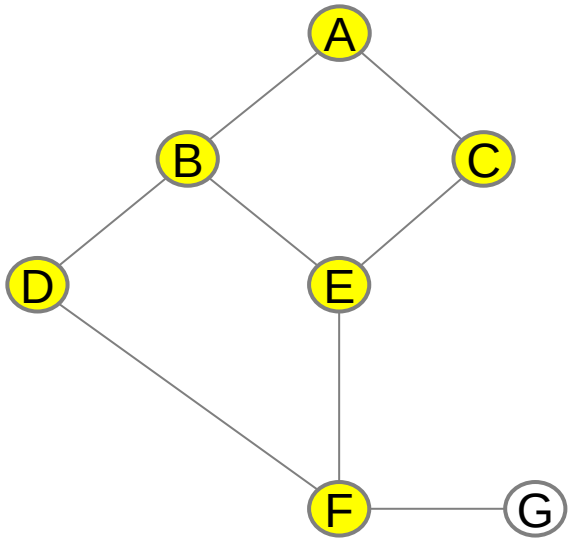
정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	F	F

출력 : A – B – C – D – E



BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	F

출력 : A – B – C – D – E – F

t D

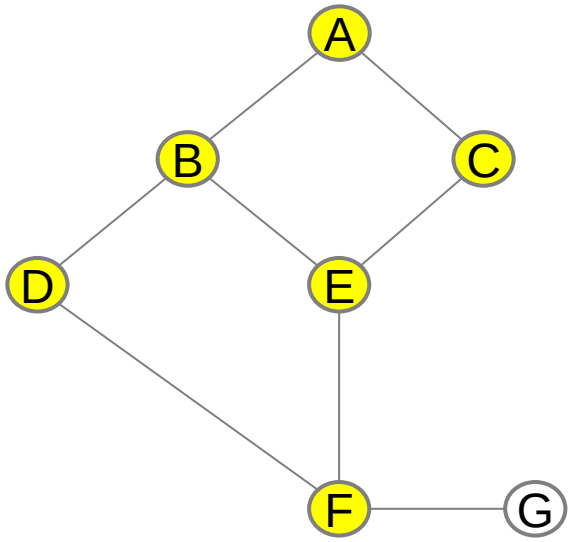
u B,F

Queue E F

D의 인접정점 B는 모두 방문했으므로 F 만 큐에 추가.

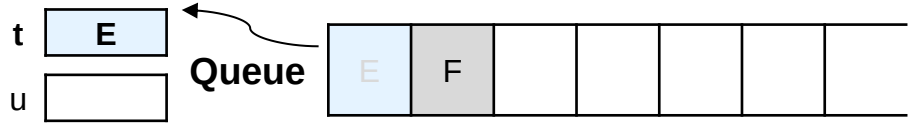
BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	F

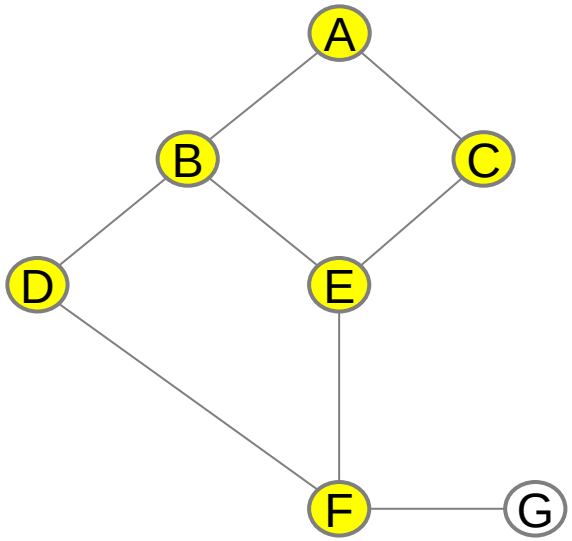
출력 : A – B – C – D – E – F



D의 인접정점 B는 모두 방문했으므로 F 만 큐에 추가.

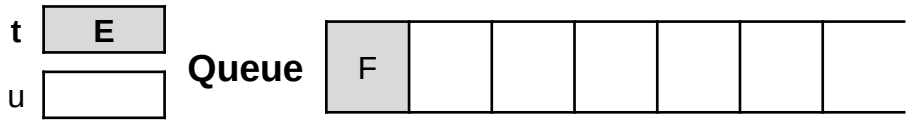
BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	F

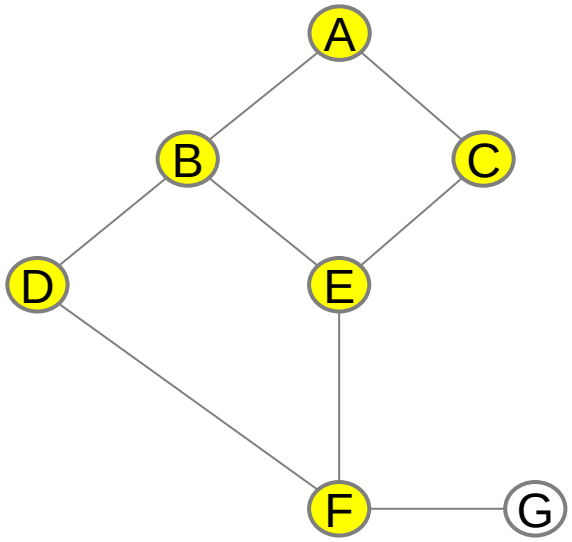
출력 : A – B – C – D – E – F



E의 인접정점 B,C,F는 모두 방문했으므로 큐에 추가 안 함.

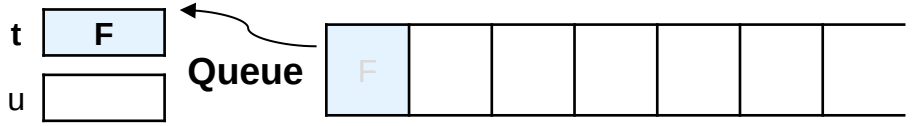
BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



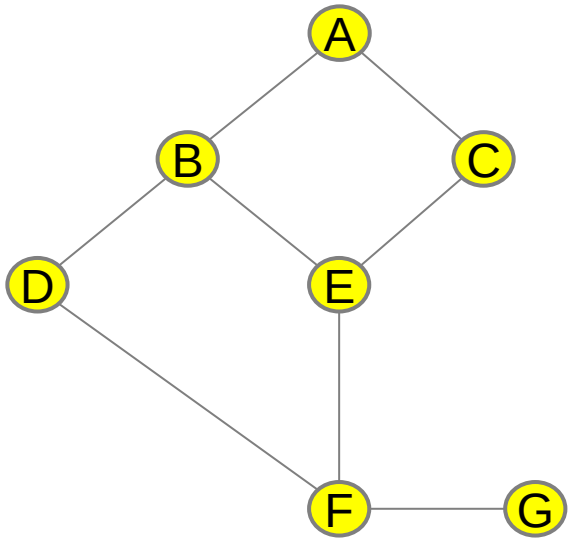
정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	F

출력 : A – B – C – D – E – F



BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

출력 : A – B – C – D – E – F – G

t F

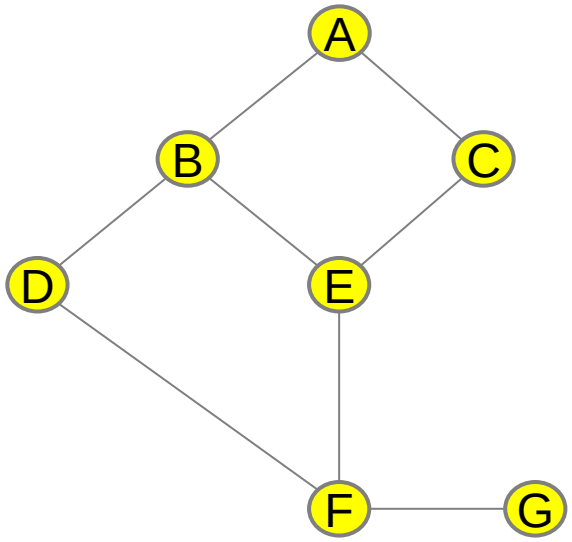
u D,E,G

Queue G

F의 인접정점 D,E는 방문했으므로 G 만 큐에 추가함.

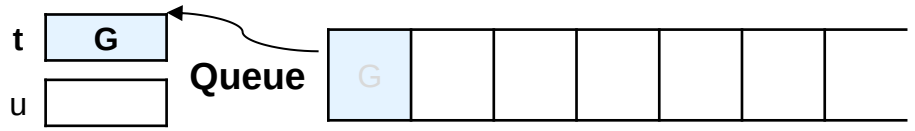
BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

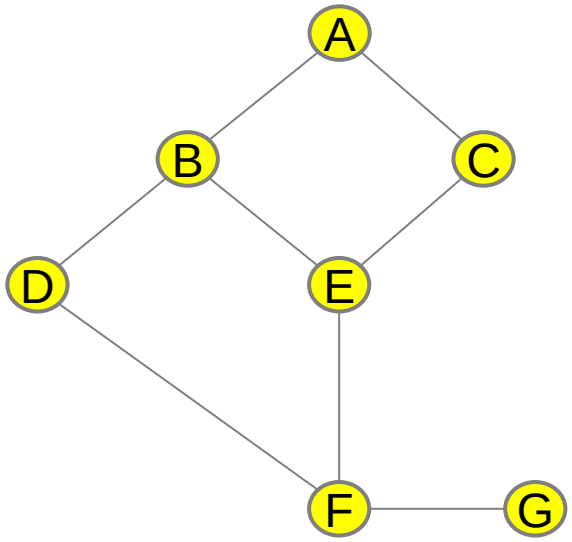
출력 : A – B – C – D – E – F – G



F의 인접정점 D,E는 방문했으므로 G 만 큐에 추가함.

BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

출력 : A – B – C – D – E – F – G

t G

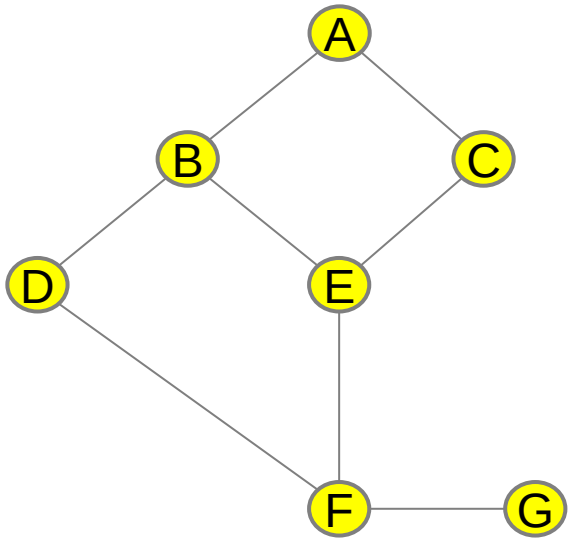
u F

Queue

G의 인접정점 F는 방문했으므로 큐에 추가 안 함.

BFS 알고리즘 동작

```
BFS(v):
 queue.append(v)
 visited[t] = True
 print(t)

 while queue:
   t = queue.popleft()

   for u in G[t]:
      if not visited[u]:
         queue.append(u)
         visited[u] = True
         print(u)
```



정점	A	B	C	D	E	F	G
visited	T	T	T	T	T	T	T

출력 : A – B – C – D – E – F – G

t G

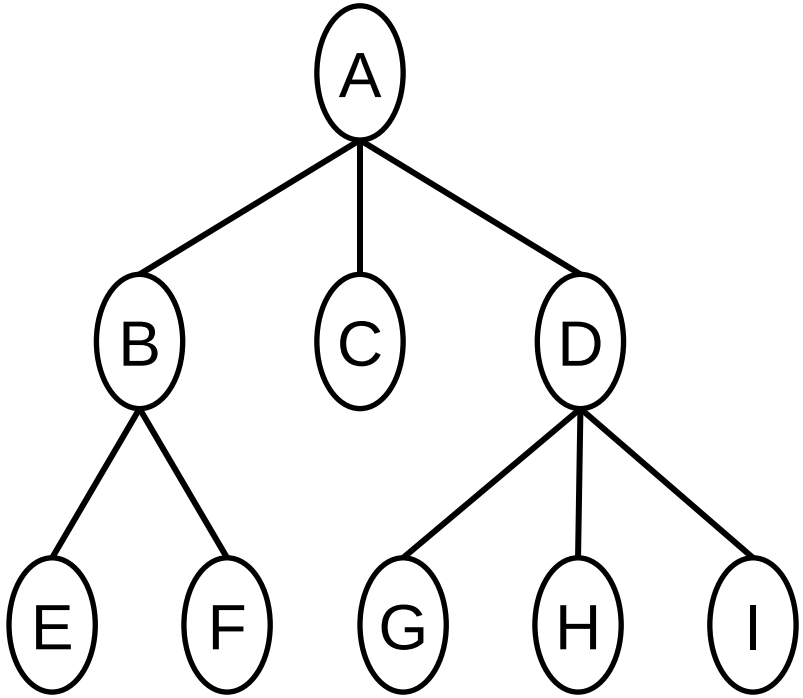
u

Queue

큐가 비었으므로 종료.

생각해 보기

- 아래의 그래프를 DFS 로 탐색했을 때와 BFS 로 탐색했을 때 탐색 순서를 시뮬레이션해 보자.



- DFS 탐색 결과
  - A-B-E-F-C-D-G-H-I
  - A-D-I-H-G-C-B-F-E
- BFS 탐색 결과
  - A-B-C-D-E-F-G-H-I
  - A-D-C-B-I-H-G-F-E
- 예시된 결과 외에 다른 결과도 가능.

질문

- BFS 는 재귀로 구현이 불가능한가?
  - BFS 도 반복적 처리되므로, 재귀로 구현이 가능하다.
    그러나, DFS 의 경우처럼 재귀로 구현했을 때의 이점이 없다.
  - DFS 를 재귀로 구현했을 때 이점
    - 재귀 호출이 스택의 역할을 대신한다. 즉 스택 제어가 필요 없다.
    - 구현이 간단해진다
  - BFS 를 재귀로 구현한다면?
    - 큐는 재귀로 구현해도 필요하다.
    - 재귀 호출로 하나의 반복문이 감소하지만, BFS 구현 코드가 그리 간단해지지는 않는다.

질문

- 어떤 경우엔 DFS 를, 어떤 경우엔 BFS 를 적용할까?
  - DFS, BFS 둘 다 그래프의 모든 정점을 탐색하는 알고리즘이다. 따라서 모든 정점을 다 탐색하는 문제에서는 어떤 알고리즘을 써도 무방하다.
  - DFS 는 그래프의 제일 끝 정점까지 가 봐야 답을 구할 수 있는 경우에 유용하다. 제일 끝 인접 정점까지 정점들은 탐색하고 연산하면서 답을 찾아가는 경우에 적용할 수 있다. 미로에 길이 존재하는 가와 같은 문제에서 적합하다.
  - BFS 도 그래프의 가장 끝 정점까지 탐색하면서 답을 구하는 경우에도 적용이 가능하다. 그러나, 대체적으로 시작 정점을 기준으로 같은 layer(또는 depth) 에서의 연산이 필요한 경우에 DFS 로 구현하는 것보다 구현이 간단해진다. 또, 주어진 문제에서 정해진 layer(depth) 까지만 탐색을 요구하는 경우에는 BFS 가 더 좋은 해결 방법이다.