Chapter 3. Decompositions of graphs(그래프 분해)

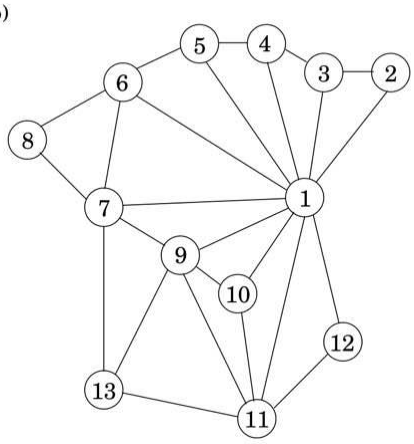
3.1 Why graphs?

-Graph 는 vertex(정점)와 edge(간선)로 이루어져 있음

ex. V = {1, 2, ⋯, 13}

(a)

(b)



(1) Undirected graphs, *e = {x, y}*

"An edge between x and y" = "x shares a border with y" = "y shares a border with x"

= "symmetric relation" (대칭 관계)

(2) Directed graphs, *e = (x, y)*

- Use edges with directions

- Directed edges e from x to y

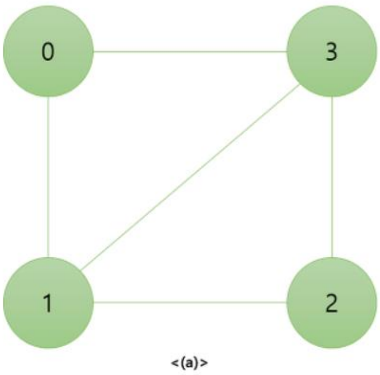
3.1.1 How is a graph represented?

-그래프를 <u>adjacency matrix(인접 행렬)</u>로 대표함

인접행렬: 그래프의 연결 관계를 **이차원 배열**로 나타내는 방식

vertex 가 v1, v2, ⋯, vn 이고 n=|v|이면, n*n 배열로 나타낼 수 있음

adj[i] [j] *: 노드 i 에서 노드 j 로 가는 간선이 있으면 1, 아니면 0*



	[0]	[1]	[2]	[3]
[0]	0	1	0	1
[1]	1	0	1	1
[2]	0	1	0	0
[3]	1	1	1	0

출처: https://kingpodo.tistory.com/46

- Undirected graph: symmetric(대칭)
- 장점: 특정 edge(노드 i 와 j 가 연결되어 있는지) 확인하고 싶을 때
  노드 i 와 j 가 연결되어 있는지 확인하고 싶을 때 adj[i] [j]가 1 인지 0 인지만 확인하면 됨
  ⇨ O(1)이라는 시간 복잡도
- 단점: 전체 노드의 개수를 V 개, 간선의 개수를 E 개.
  Matrix(한 노드에 연결된 모든 노드들)에 방문해보고 싶은 경우 adj[i] [1]부터 adj[i] [V]를 모두 확인해보아야 함 ➔ O(V)의 시간 소요
  각 노드에 연결된 노드를 방문 ➔ 모든 노드에 대해 확인해보아야 함 ➔ O(V*V)의 시간 소요

대안: adjacency list(인접 리스트)

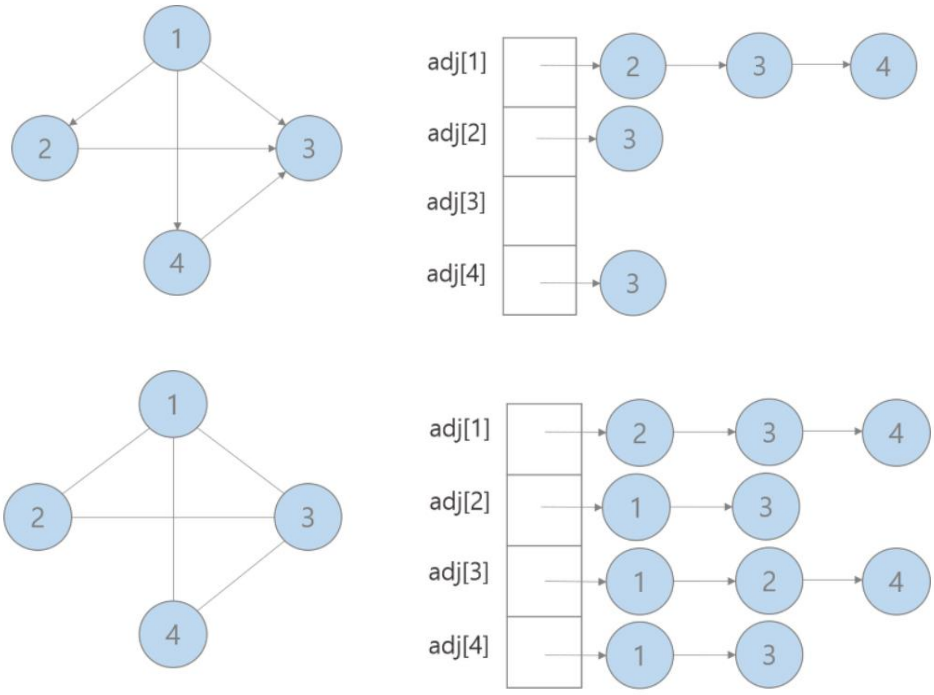
각각의 vertex 에 인접한 vertex 들을 linked list 로 표시한 것

정점 u 는 u 에서 나가는 정점들과 연결되어 있음

adj[i] *: 노드 i 에 연결된 노드들을 원소로 갖는 vector*

Directed(방향) graph 라면 edge 가 linked list 에서 한 번, undirected graph 라면 두 번 나옴

Total size: O(|E|)



출처: https://sarah950716.tistory.com/12

하나의 edge (u, v)를 확인하는 것은 더 이상 constant time 이 아님 (왜냐하면 adjacency list 를 체크해야 되니까)

그런데 linked list 에서 반복되기 쉬움

-장점: 인접 리스트는 인접 행렬과 달리, **실제로 연결된 노드들에 대한 정보만 저장**하기 때문에, 모든 벡터들의 원소의 개수의 합이 간선의 개수와 같음, 즉, 간선의 개수에 비례하는 메모리만 차지

-인접 리스트의 경우에는 각 노드마다 연결된 노드만 확인하는 것이 가능하기 때문에, 전체 **간선의 개수**만큼만 확인해 볼 수가 있음. 따라서, **O(E)의 시간복잡도**를 가짐.

3.2 Depth First Search in undirected graphs

3.2.1 Exploring mazes

“ Which parts of the graph are reachable from a give vertex? ”

-미궁을 탐색하기 위해서는 실과 분필이 필요하다?

분필: 방문한 교차점들을 표시하여 looping 방지

실: 시작점으로 되돌아가서 아직 조사하지 않은 곳을 찾을 수 있도록 함

컴퓨터에 적용 시,

분필-〉 각각의 vertex 에 Boolean 으로 방문 여부 확인

실-〉 stack 활용 (1. 새 vertex 를 넣음, 2. 이전으로 돌아가기 위해 vertex 를 꺼냄)

Recursion 예방 가능

Previsit: 처음 발견될 때의 작업

Postvisit: 마지막으로 떠날 때 작업

Result of running explore

Figure 3.4 The result of explore(A) on the graph of Figure 3.2.

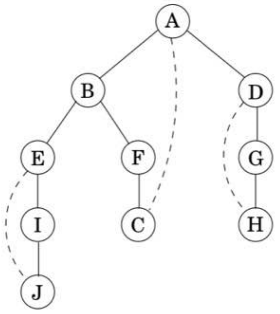


Figure 3.5 Depth-first search.

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v):  explore(v)
```

explore 함수가 정확한지 확인 하는 것이 우선

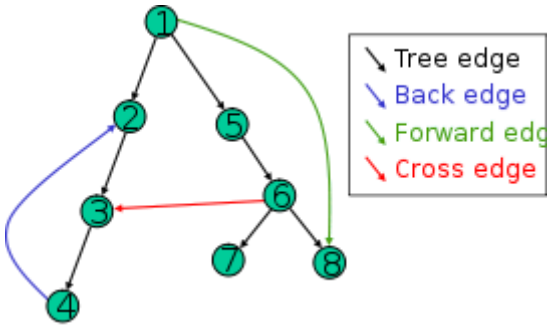
explore 함수는 이웃 노드로만 이동하고, 어떤 지역을 절대 뛰어넘을 수 없음

Ex.

노드 z 에서 이웃 노드인 w 를 알아차리고 움직였을 것임

Solid edges form a tree, called tree edges

Dotted edges led back to vertices previously visited, called back edges



출처: https://cs.stackexchange.com/questions/11116/difference-between-cross-edges-and-forward-edges-in-a-dft

3.2.2 Depth-first search

Explore 은 그래프의 부분만 방문하기에, 나머지 부분을 방문하기 위해서는 아직 방문하지 않은

vertex 에서 과정을 다시 시작해야 됨

이 알고리즘을 **DFS(Depth-first search)**라고 부르고, 전체 그래프를 순회할 때까지 반복됨

1. 각각의 vertex 는 한 번만 explore 됨(visited array, chalk 덕분에)

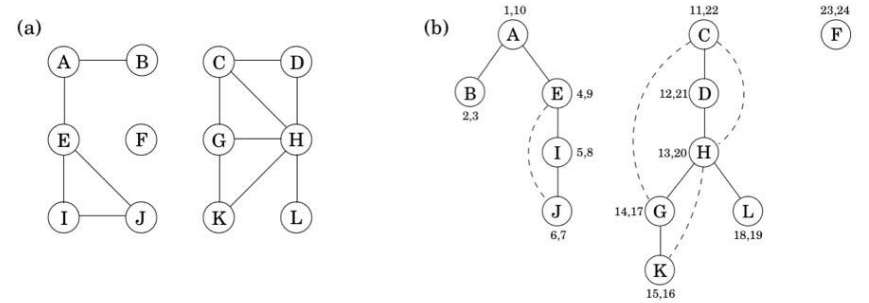
(1) 방문한 점을 표시하는 것, pre/post visit => **O(|V|)**

(2) adjacent edges(인접 간선)을 스캔하는 루프 => **O(|E|)**

각각의 간선 $\{x,y\} \in E$ 에서 두 번 조사됨(explore(x), explore(y))

⇨ 입력 크기가 linear 할 때 DFS has a running time of **O(|V|+|E|)**

Figure 3.6 (a) A 12-node graph. (b) DFS search forest.



3.2.3 Connectivity in undirected graphs

Figure 3.6 은 A-〉F 의 edge 가 존재하지 않으므로 그래프가 **not connected**

3 개의 분리된 연결 영역을 가지고 있음 (=connected component(연결 요소))

{A, B, E, I, J}, {C, D ,G, H, K, L}, {F}

-특정 vertex 에서 explore 이 시작되면 각 영역의 connected components 만 방문함

-DFS 는 그래프가 연결되어 있는지 확인하고, 각 노드 v 에 연결된 component 를 식별하는 정수 ccnum[v]를 할당하도록 조정됨

cc 는 DFS 과정이 explore 을 호출할 때 하나씩 증가함.

```
procedure previsit(v)
ccnum[v] = cc
```

3.2.4 Previsit and postvisit orderings

1. previsit: 처음으로 발견한 순간
2. postvisit: 마지막 떠남

1 부터 시작하는 clock counter 로 정의할 수 있음

procedure previsit(v)

Pre[v] = clock

clock = clock + 1

procedure postvisit(v)

post[v] = clock

clock = clock + 1

Property 노드 u 와 v 에 대하여 두 간격인 [pre(u), post(u)], [pre(v), post(v)]는 분리되어 있거나 하나가 다른 하나에 포함되어 있음

Why? [pre(u), post(u)]는 u 가 stack(스택)에 머무는 시간임. Stack 은 last in first out.

3.3 Depth-first search in directed graphs

3.3.1 Types of edges

A: root 이고 나머지는 자손.

E : F, G, H 가 자손이고 이 노드들의 조상임

C: D 의 부모임. D 는 C 의 가족

Undirected graph 에서 tree edges 와 nontree edges 는 구분됨

Tree edges 는 DFS forest 의 부분임

- Forward edge 들은 자식이 아닌 자손 노드와 연결됨

-Cross edge 들은 자손과 조상 둘 다 연결되지 않고 이미 방문한 노드로 이어짐

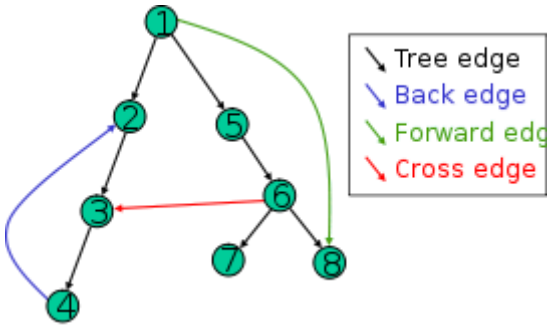
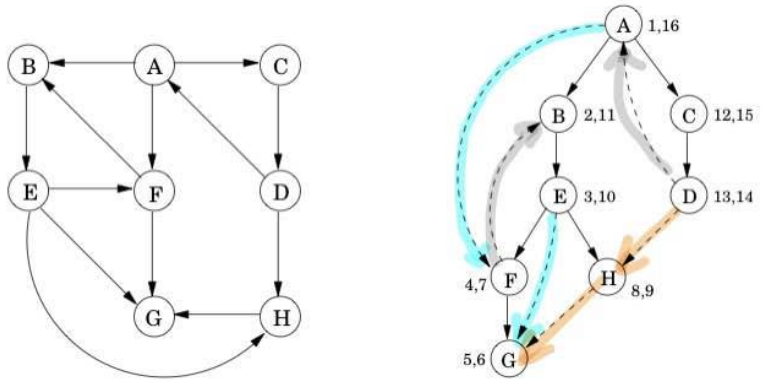


Figure 3.7 DFS on a directed graph.



조상과 자손 관계는 pre/post numbers 로부터 읽을 수 있음

U 가 먼저 발견되었고, v 는 explore u 중에 발견될 때 vertex u 는 vertex u 의 조상임

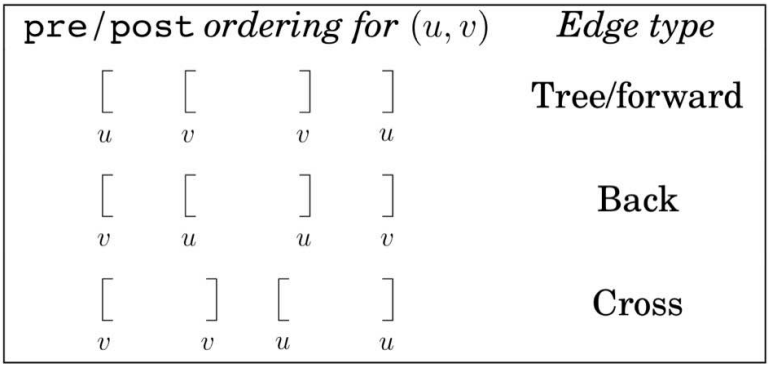
Pre(u) ⟨ pre(v) ⟨ post(v) ⟨ post(u)

[[ ]]

u v v u

v 가 u 의 조상일 경우에만 u 가 v 의 자손이므로 대칭임

edge 카테고리는 전적으로 조상 자손 관계이므로 pre/post number 에서도 읽을 수 있음



3.3.2 Directed acyclic graphs (cycle 이 없는 그래프. 방향 비순환 그래프)

순환경로: (v0->v1->v2->…->vk->v0) Ex. B->E->F->B

DAG 에서는 순환하는 cycle 존재하지 않고, 일방향성만 가짐.

작업의 우선순위를 표현하며, 블록체인의 핵심적인 특징이기도 함

Property Directed graph 는 DFS 에서 back edge(노드->조상)이 있는 경우에만 cycle 이 있음

Proof (u, v)가 back edge 이면, 검색 트리에는 v 에서 u 까지의 경로와 함께 cycle 로 구성됨

그래프가 cycle 이 있으면 cycle 의 첫 번째 노드가 있음. 이 노드를 vi 라고 가정하면 이 노드는 vj 들을 도달할 수 있음

Cycle 의 모든 v j 는 cycle 에서 도달할 수 있으므로, 검색 트리에서 자손이 됨

특히 vi-1 -> vi 는 노드에서 그 조상으로 이어지고 back edge 의 정의임

Directed acyclic graphs (Dags, 방향 비순환 그래프)

인과관계와 계층구조, 시간적 의존성과 같은 모델을 모델링 하기 좋음

Ex. 집을 지을 때 기둥을 세운 후에 인테리어를 할 수 있음

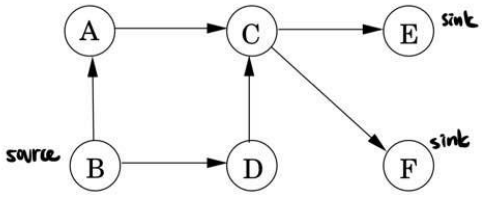
⇨ Directed graph 에서 각각의 task 가 node 이고, u 가 v 의 전제조건일 때, u->v 의 edge 가 존재하는 것

즉, 어떤 작업을 수행하기 전에 해당 작업을 향하는 모든 작업이 완료되어야 함

만약 cycle 이 생기면 순서를 생성할 수가 없음

DAG 는 이전 vertex 에서 다음 vertex 로 각 간선을 연결하므로 linearize(선형화)가 가능함

Figure 3.8 A directed acyclic graph with one source, two sinks, and four possible linearizations.



B, A, D, C, E, F

DAG 라면 linearize(선형화, topologically sort 위상정렬)하고자 함

- 위상 정렬은 유향 그래프의 꼭짓점들을 변의 방향을 거스르지 않도록 나열하는 것을 의미함. 위상정렬을 가장 잘 설명해 줄 수 있는 예로 대학의 선수과목 구조

모든 DAG 는 linearize(선형화)될 수 있음

How? post number 가 감소하도록 수행하면 됨

Post(u) 〈 post(v)인 간선 (u, v)만이 역방향 간선이 되고, DAG 는 역방향 간선을 가질 수 없음

Property DAG 에서 모든 간선은 더 낮은 post number 를 가진 vertex 에 다다름

➔ DAG 의 노드는 linear-time algorithm 이 됨
➔ Acyclicity(순환성) = linearizability(선형화) = absence of back edge

가장 작은 post number: sink(끝점), a node with outgoing edges

가장 큰 post number: source(출처), a node with no incoming edges

Property 모든 DAG 는 적어도 하나의 source 와 sink 가 있음

Source 가 존재하기에 linearization 에 대해 다른 방법으로 접근 가능함

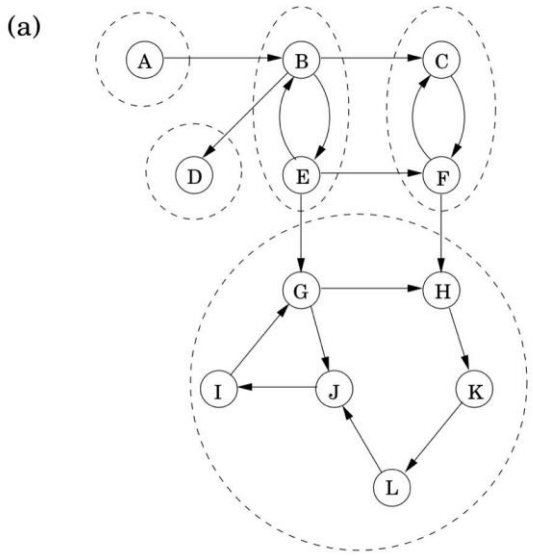
- Source 를 찾아서 출력하고 source 를 지움
- 그래프가 빌 때까지 이 과정 반복

3.4 Strongly connected components

3.4.1 Defining connectivity for directed graphs

Directed graph 에서는 연결성이 더 세밀함

연결되어 있지 않은 그래프는 여러 개의 connected components 로 분해될 수 있음



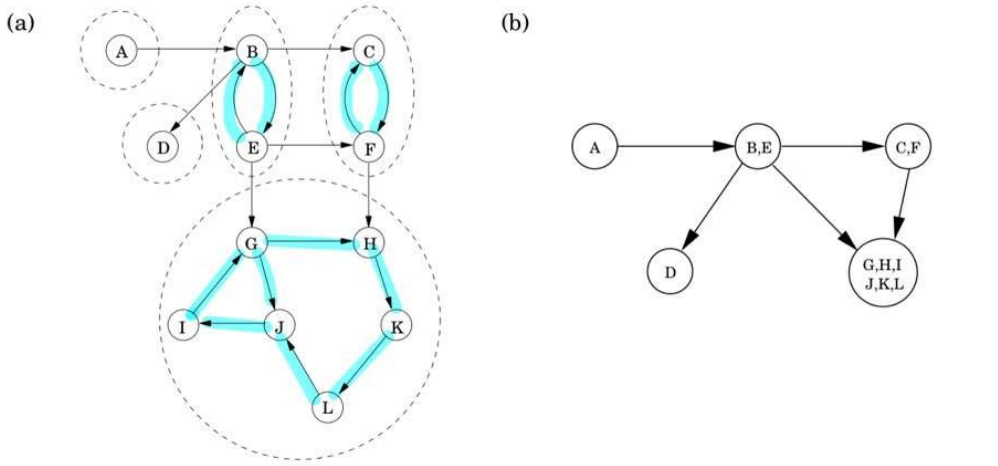
이 그래프는 연결되어 있지 않음(G->B, F->A 이동 불가능)

정의하자면, directed graph 의 노드인 u 와 v 는 u->v 경로와 v->u 경로가 있을 시 연결되어 있음

그래서 (a)가 5 개의 strongly connected components(강한 연결 성분)으로 나눠짐

각각의 Strongly connected component 을 meta-node 로 줄이고 meta-node 들을 연결하면 DAG 가 됨

Figure 3.9 (a) A directed graph and its strongly connected components. (b) The meta-graph.



Property 모든 directed graph 는 strongly connected component 의 DAG 임

Directed graph 의 연결 구조는 두 계층(two-tiered)으로 이루어짐

최상위 레벨에는 linearize 할 수 있는 DAG 가 존재

더 자세하게 알기 위해서는 DAG 의 노드를 보고 내부의 완전한 strongly connected component 를 검사할 수 있음

3.4.2 An Efficient Algorithm

Property 1 Explore subroutine 이 노드 u 에서 시작하면, u 로부터 이를 수 있는 모든 노드를 방문해야 완전히 종료될 수 있음

Strongly connect component 의 sink 에서 놓인 노드에서 explore 을 호출하면 해당 노드의 값을 추출함

Strongly connected component sink 가 2 개 있다

노드 k 에서 explore 시작하면, 노드 k 가 속한 strongly connected components 전체를 순회하고 멈춤

남은 2 개의 중요한 문제

(A) Strongly connected component 의 sink 에 있는 노드를 어떻게 찾을 것인가

(B) 첫 번째 component 를 발견한 후 어떻게 진행하는가

(A) strongly connected component 의 sink 에 놓인다고 보장되는 노드를 고를 수 있는 쉬운 방법은 없음. 대신 있는 노드를 얻을 수 있는 방법은 있음.

Property 2 DFS 에서 가장 높은 post number 를 가지는 노드는 strongly connected component 의 source 에 놓여져야 함

Property 3 만약 C 와 C'가 strongly connected component 이고 C 안의 노드에서 C'의 노드로 edge 가 있으면 C 에서 가장 높은 post number 는 C'에서의 가장 높은 post number 보다 크다.

Proof 2 가지 경우가 있음

1. DFS 가 C 를 먼저 방문하면 과정이 멈추기 전에 C 와 C'를 모두 순회함

그래서 C 에서 방문한 첫번째 노드가 C'의 어떤 노드들보다도 더 큰 post number 를 가짐

2. C'를 처음 방문하면, DFS 는 C'의 모든 노드를 살핀 후 C 를 하나도 보지 않고 멈출 것임

Property 3 는 strongly connected component 의 가장 높은 post number 를 줄어드는 순서로 배열하면 strongly connected component 를 linearize 할 수 있다는 말임

⇨ 이것으로 (A) 해결

(B) 풀이: 첫번째 sink 를 찾아서 지우면 남은 노드 중에서 가장 높은 post number 를 가진 노드는 G 에서 남은 strongly connected component 의 sink 에 속함

따라서, G'에 대해 첫번째 DFS 로부터 두번째 strongly connected component 등을 출력하는 데까지 post 숫자를 계속 사용할 수 있음

1. G' 에 대해 DFS 실행

2. G 에 대해 undirected connected components algorithm 을 실행하고, DFS 동안에 앞선 1 단계부터 post 숫자가 감소하는 순서로 노드 처리

➔ 해당 알고리즘은 linear 시간이고, linear term 의 constant 는 올바른 DFS 의 두 배임

질문: 선형 시간에 G'의 인접 리스트 표현을 어떻게 구축할 수 있나?

선형 시간에 어떻게 post 값을 감소시키면서 G 의 node 들을 linearization 할 수 있을까?

G 에 대해 최종 알고리즘을 실행해보면, (G'의 DFS 탐색에서 감소하는 post 숫자)에 대해 설정한 ordering 은 G, I, J, L, K, H, D, C, F, B, E, A 가 된다. 2 단계에서는 {G, H, I, J, K, L}, {D}, {C, F}, {B, E}, {A}와 같은 순서로 성분을 나타낼 수 있음