자바2

(알고리즘2 – 재귀/Graph)

Chapter 04
알고리즘2 – 재귀/Graph

Chapter04의 학습목표

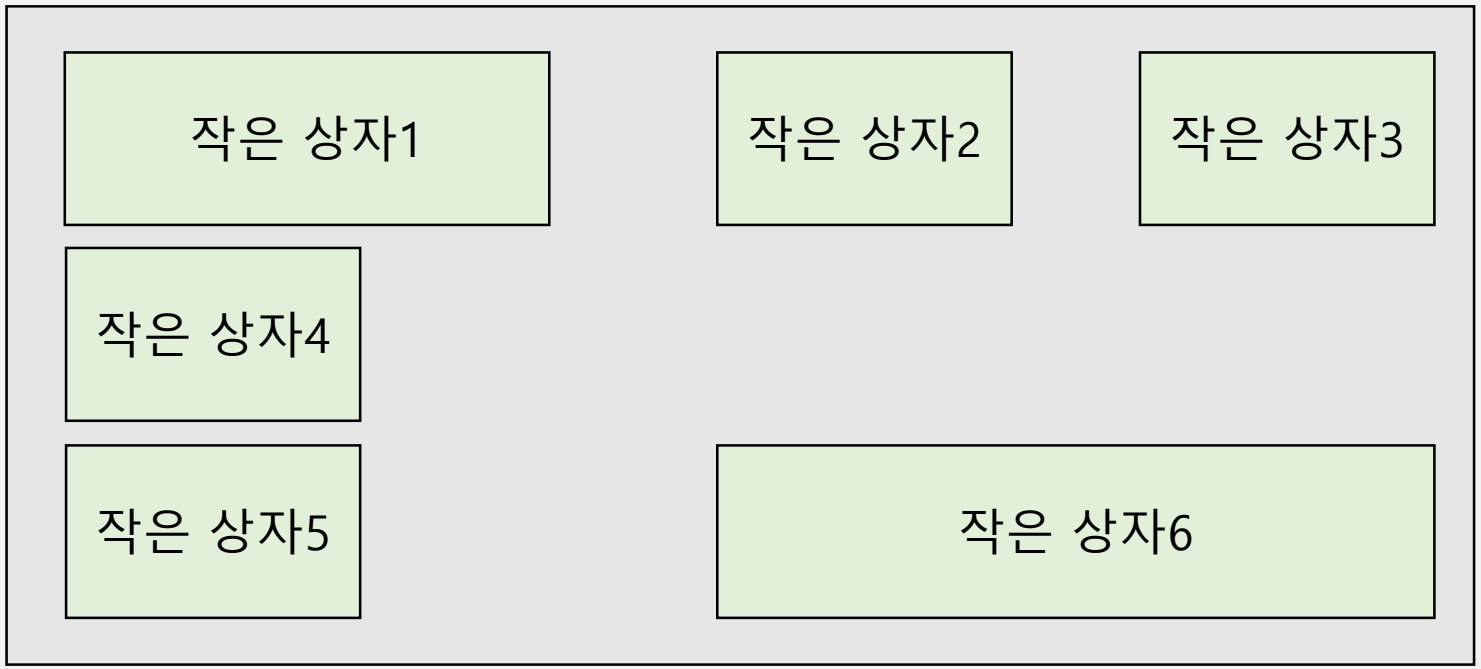
- 알고리즘과 자료구조에 대해 학습한다
- 재귀를 구현하는 방법을 학습한다
- 다익스트라/벨만 포드 알고리즘에 대해 알아보고 차이를 이해한다

재귀

재귀(Recursion)

- '원래 자리로 되돌아가거나 되돌아옴'이란 뜻으로, 어떠한 것을 정의할 때 자신을 참조하는 것을 의미
- 함수 안에서 함수 자기 자신을 호출하는 것

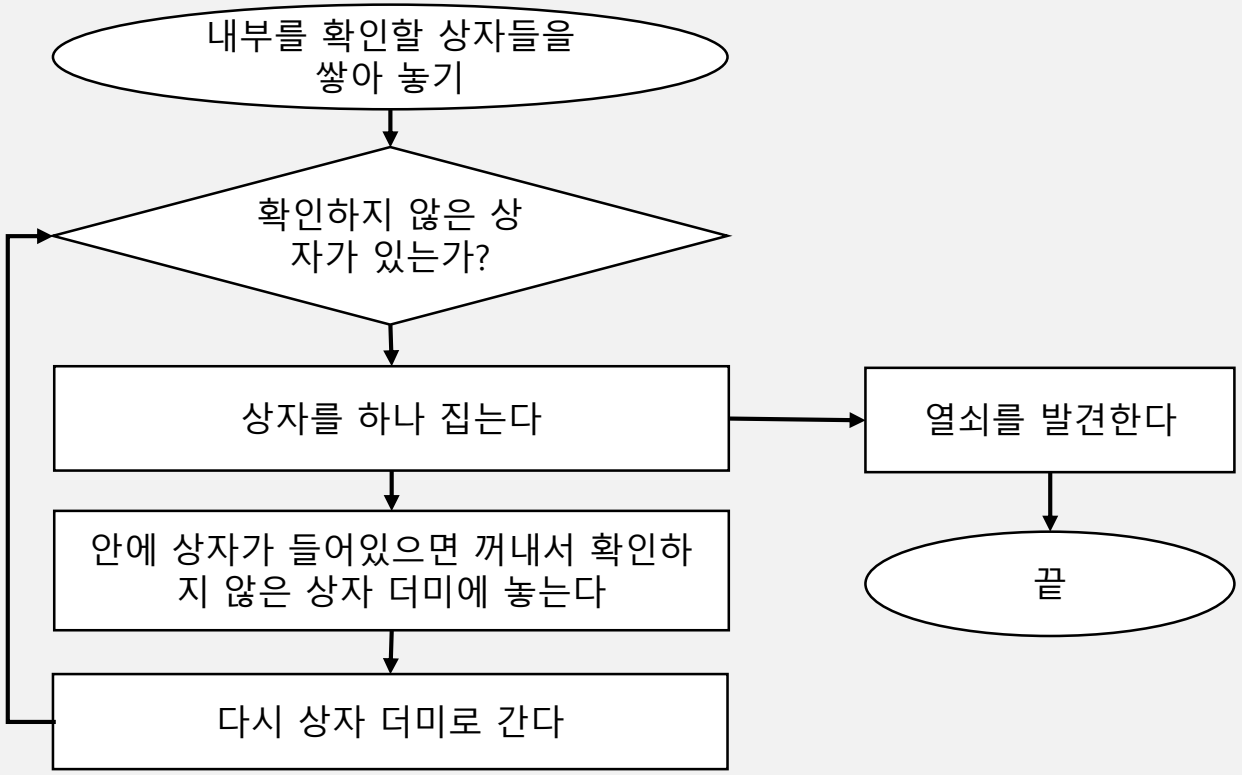
만약 큰 상자 안에서 열쇠를 찾을 때



재귀

재귀(Recursion)

방법1. while을 사용하는 방법

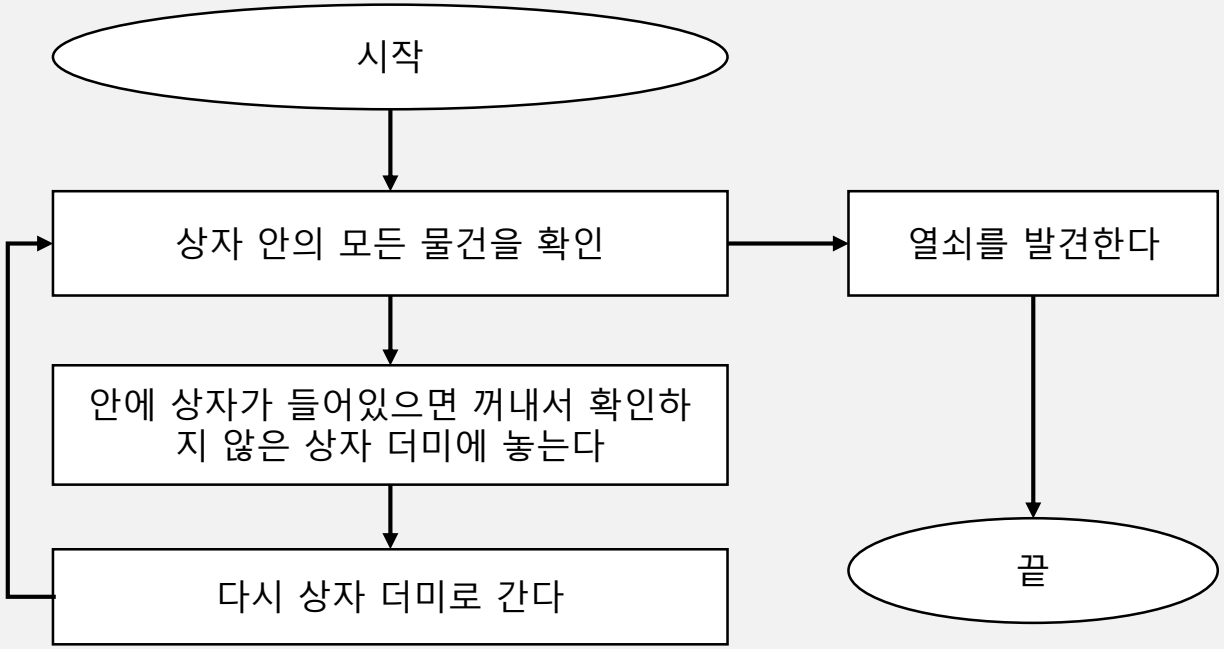



```
static void look_for_key(main_box) {
    pile = main_box.make_a_pile_to_look_through();

    while (pile is Not empty) {
        box = pile.grab_a_box();
        for(item : box) {
            if(item.is_a_box) {
                pile.add(item)
            }
            else if(item.is_a_key) {
                System.out.println("열쇠를 찾음!");
            }
        }
    }
}
```

재귀

재귀(Recursion)

방법2. 재귀를 사용하는 방법

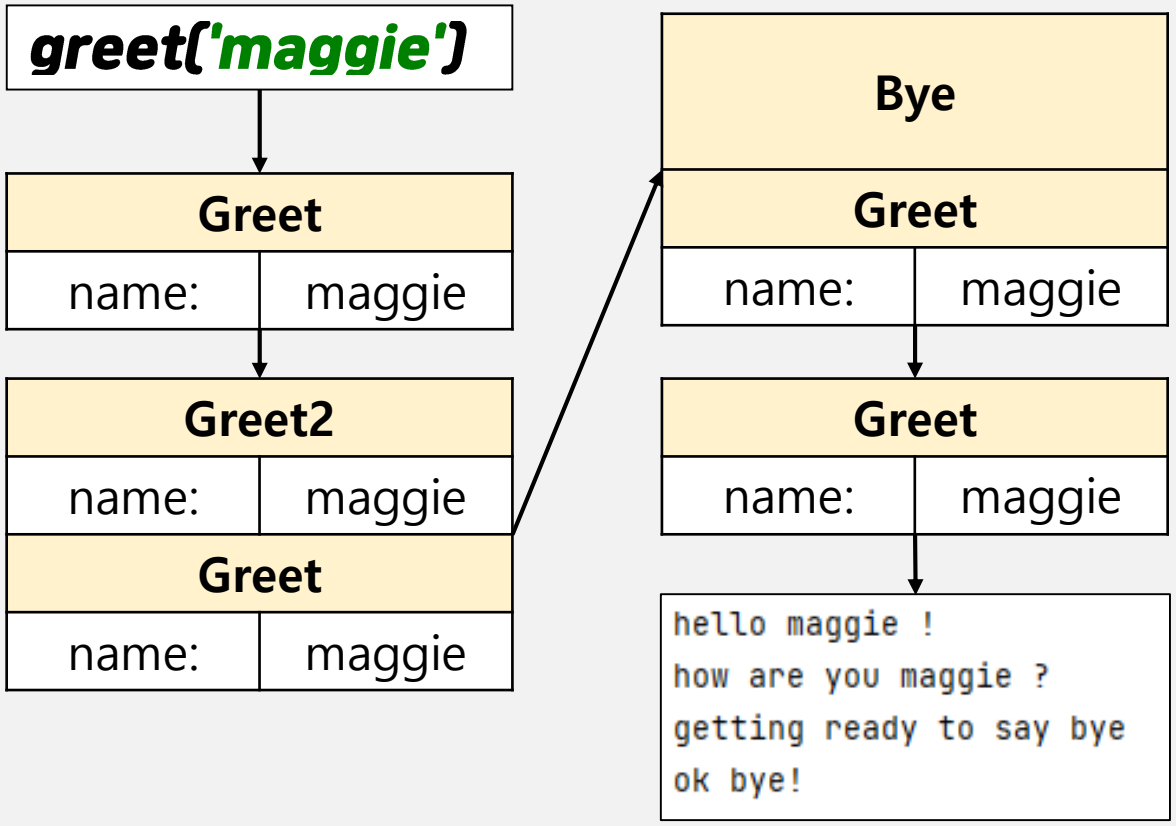



```
static void look_for_key(main_box) {
    for(item : box) {
        if(item.is_a_box) {
            look_for_key(item);
        }
        else if(item.is_a_key) {
            System.out.println("열쇠를 찾음!");
        }
    }
}
```

재귀

재귀(Recursion) – 호출 스택

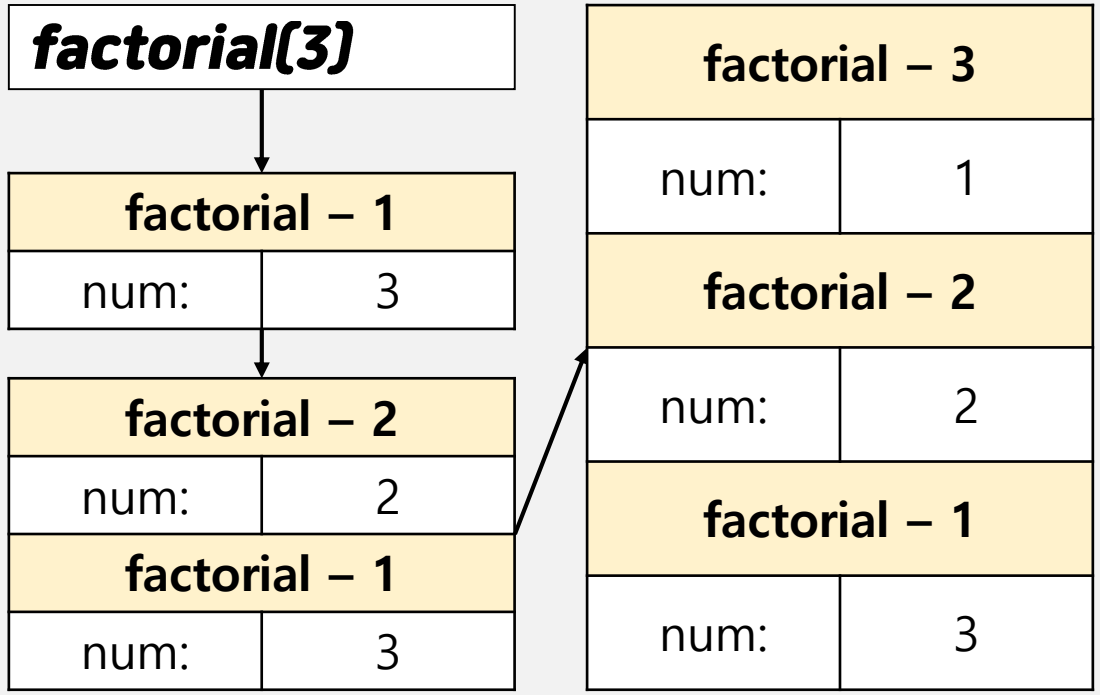
```
private static void greet(String name) {
    System.out.println("hello " + name + "!");
    greet2(name);
    System.out.println("getting ready to say bye");
    bye();
}
private static void greet2(String name) {
    System.out.println("how are you, " + name + "?");
}
private static void bye() {
    System.out.println("ok, bye!");
}
```



재귀

재귀(Recursion) – 팩토리얼 계산

```
private static int factorial(int num) {
    if (num == 1) {
        return 1;
    }
    return num * factorial(num-1);
}
```

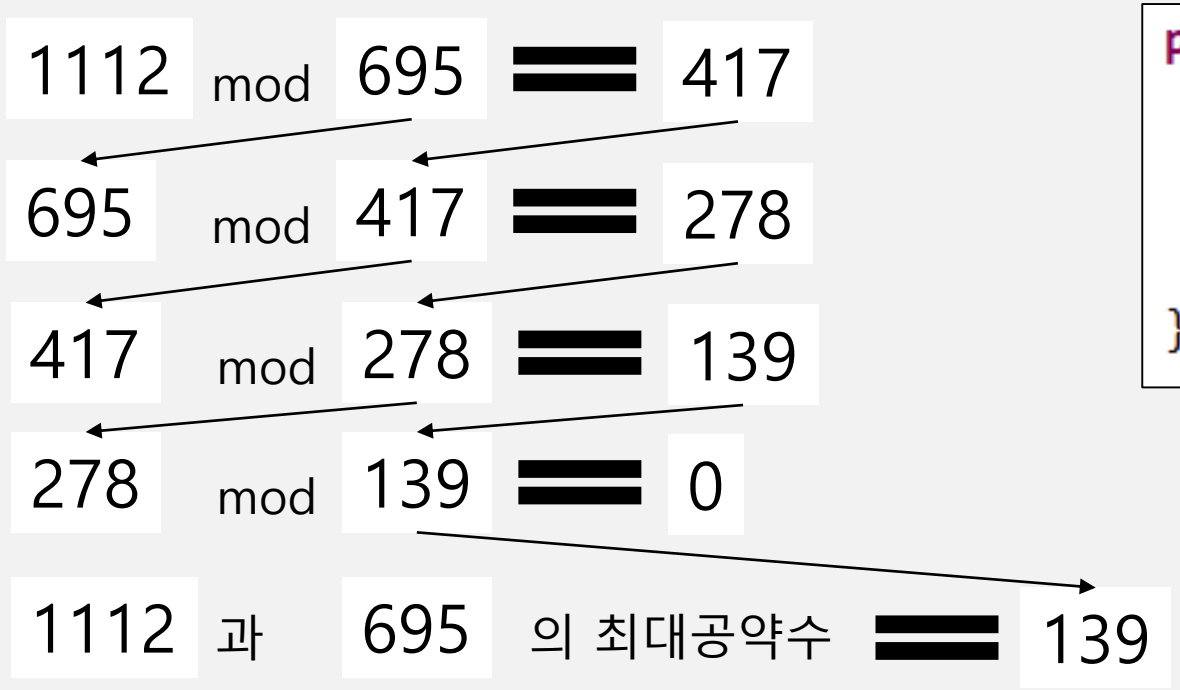


재귀

재귀(Recursion) – 유클리드 알고리즘

- 두 수의 최대공약수를 구하는 알고리즘
- 가장 오래된 기록이 기원전 300년에 작성된 유클리드 필서라는 것에서 명칭이 붙음

1112 과 695 의 최대공약수?

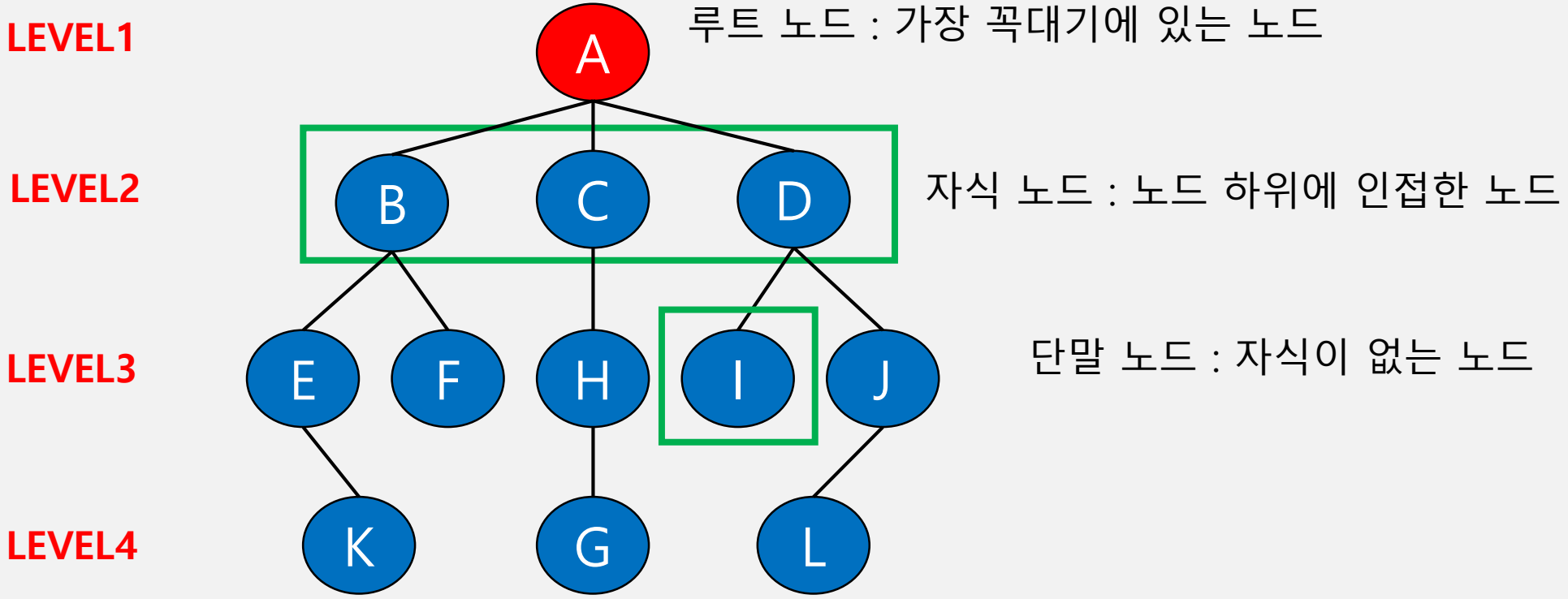



```
private static int Euclid(int num1, int num2) {
    if (num2 == 0) {
        return num1;
    }
    return Euclid(num2, num1 % num2);
}
```

그래프

트리 (Tree)

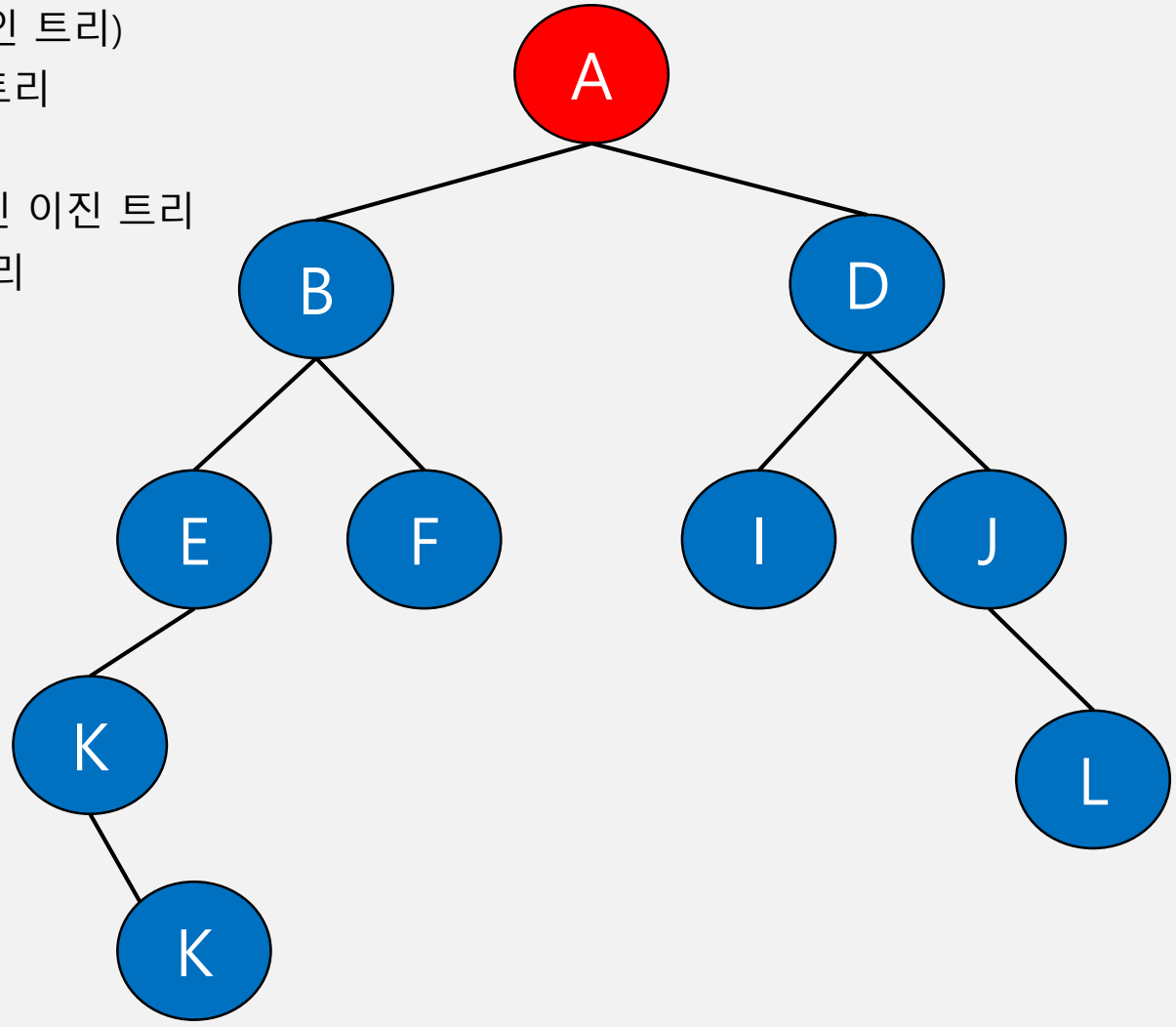
- 루트 노드를 시작으로 줄기를 뻗어 나가듯 원소를 추가해 나가는 비선형 구조
- 방향성이 있고 노드와 노드 간에 간선이 존재하여 간선으로 연결됨
- 차수 (Degree) : 노드의 하위 간선의 개수 (자식의 개수 = 간선의 개수)
- 깊이 (depth) : 루트에서 어떤 루트까지의 경로 길이



그래프

이진 탐색 트리 (BST : Binary Search Tree)

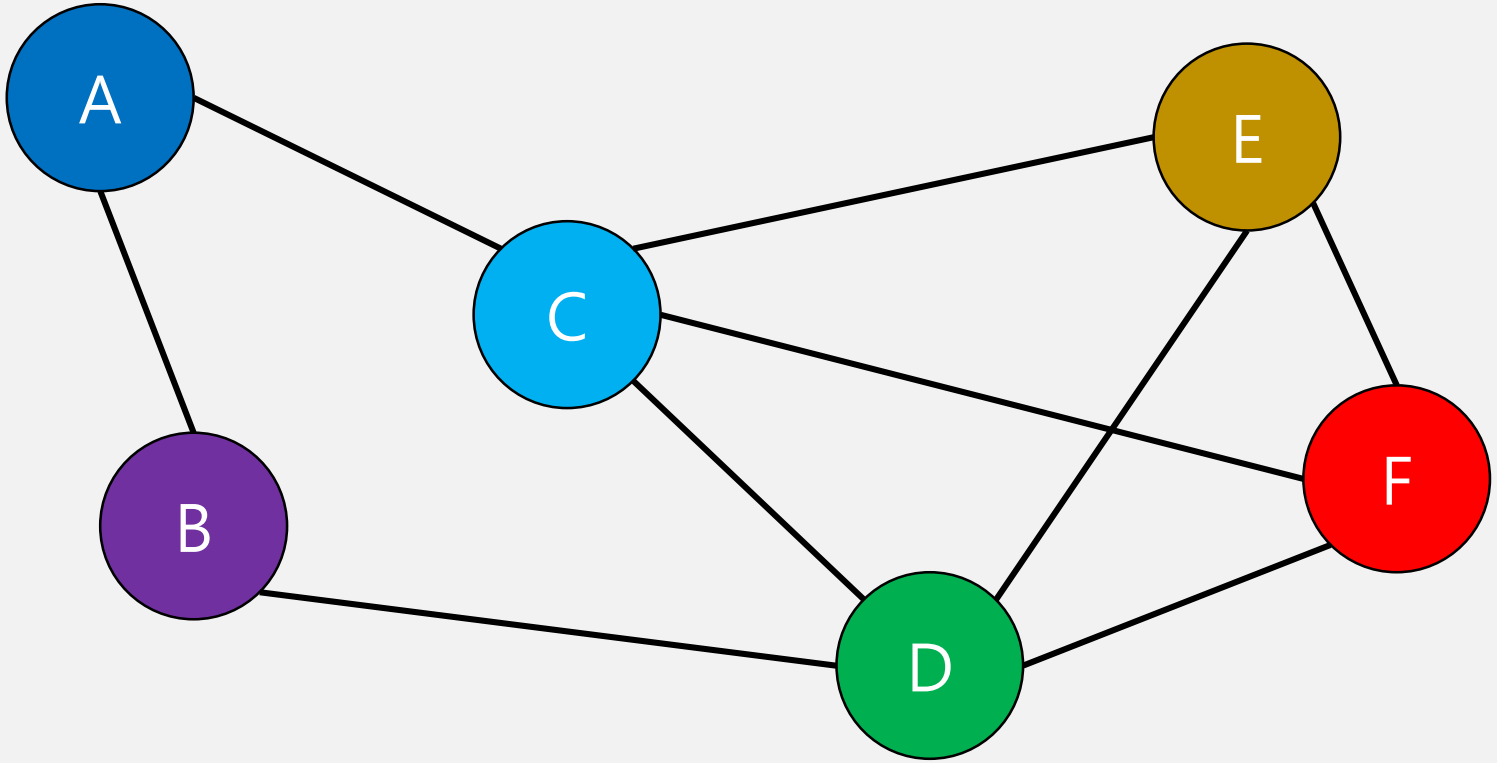
- 트리의 최대 차수가 2인 트리 (모든 노드의 자식이 최대 2개인 트리)
- 부모 노드보다 작으면 왼쪽, 크면 오른쪽 자식 노드가 되는 트리
- 중복된 데이터를 허용하지 않음
- 완전 이진 트리 : 마지막 노드를 제외하고 모든 노드가 채워진 이진 트리
- 포화 이진 트리 : 모든 단말 노드의 깊이가 같은 완전이진 트리
- 균형 이진 트리 : 좌우의 높이의 차가 같거나 최대 1인 트리



그래프

그래프 (Graph)

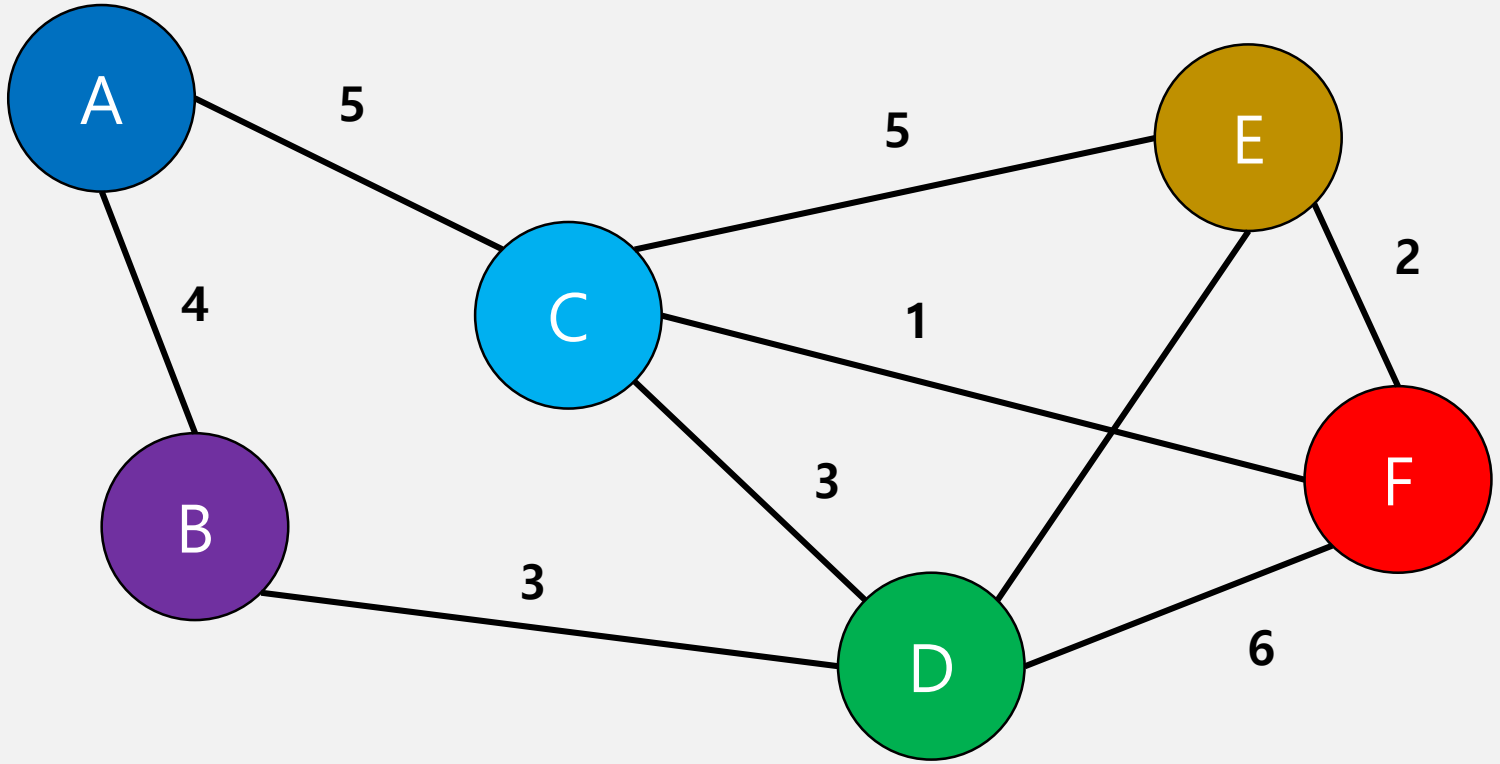
- 정점(Node)과 간선(Link)으로 이루어짐
- 연결됨, 연결되지 않음 두 개로 표현 가능



그래프

가중 그래프 (Weighted graph)

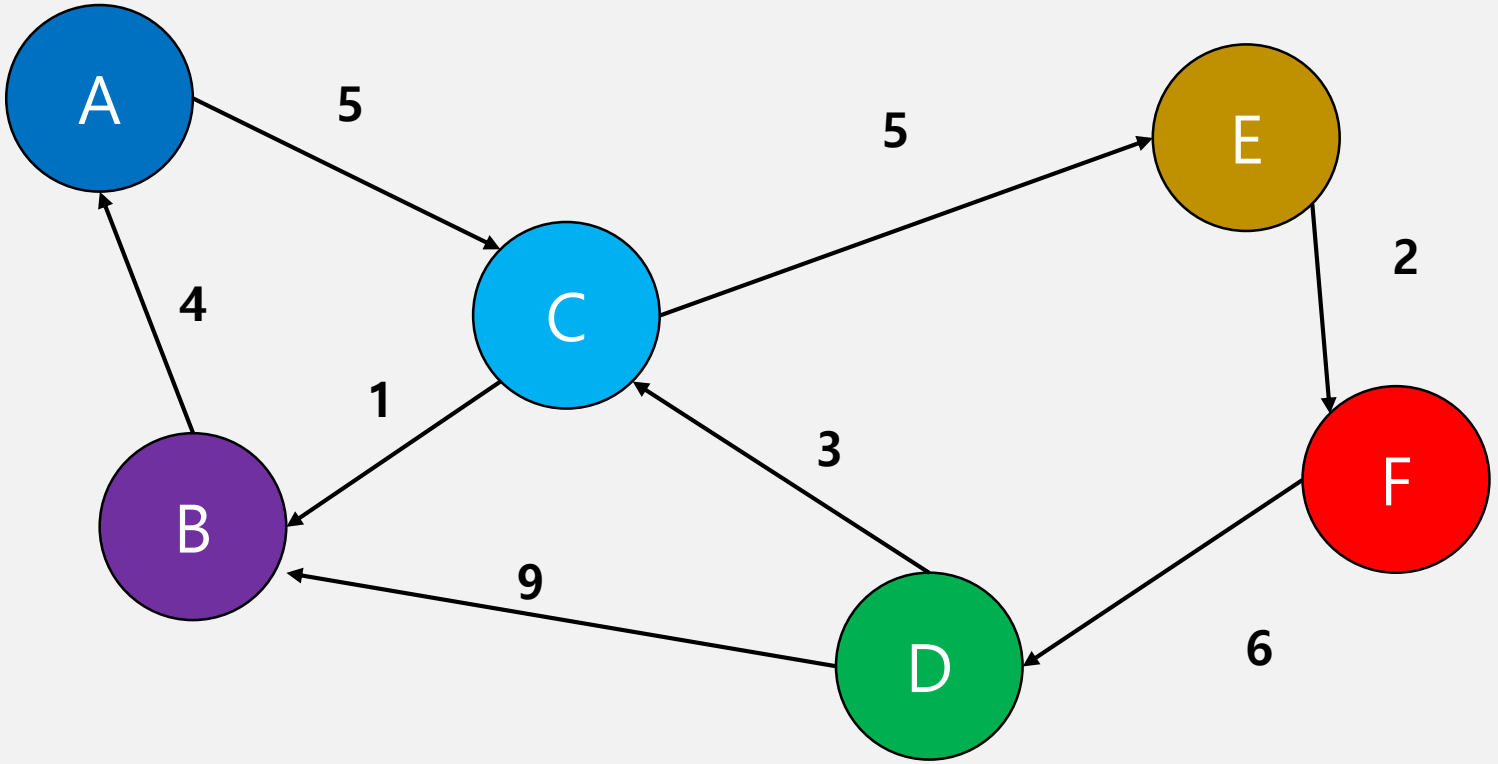
- 정점과 간선으로 이루어지고 간선에 가중치(Cost)가 부여되어 있음
- 연결의 정도를 표현 가능



그래프

방향성 그래프 (Directed graph)

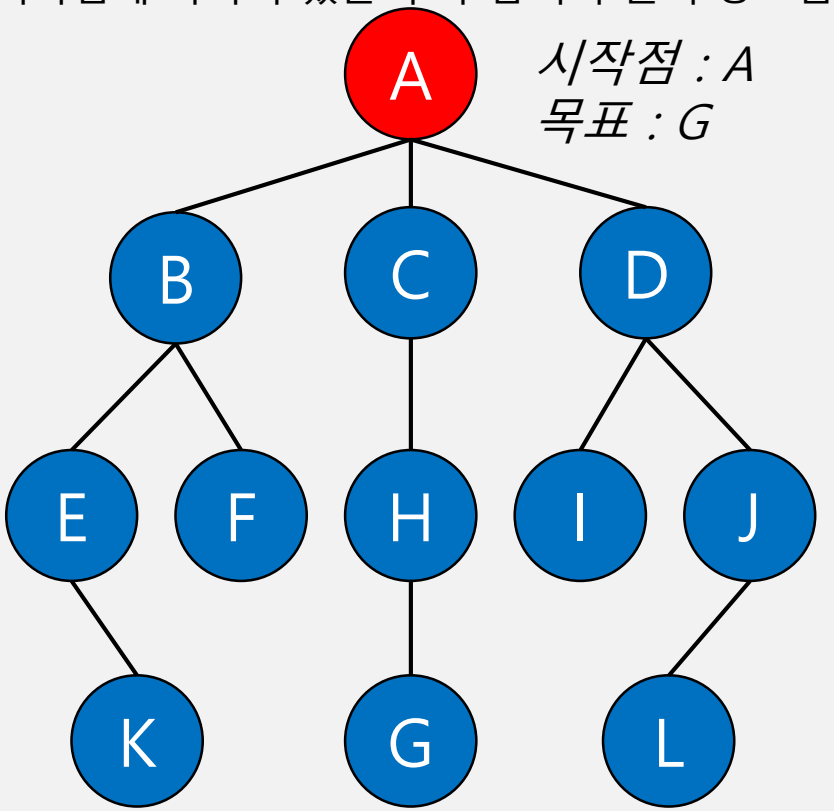
- 정점과 간선으로 이루어지고 간선에 노드 사이에 방향(direction)이 부여되어 있음
- 방향성 그래프에서는 방향과 가중치 둘 다 표현할 때도 있음



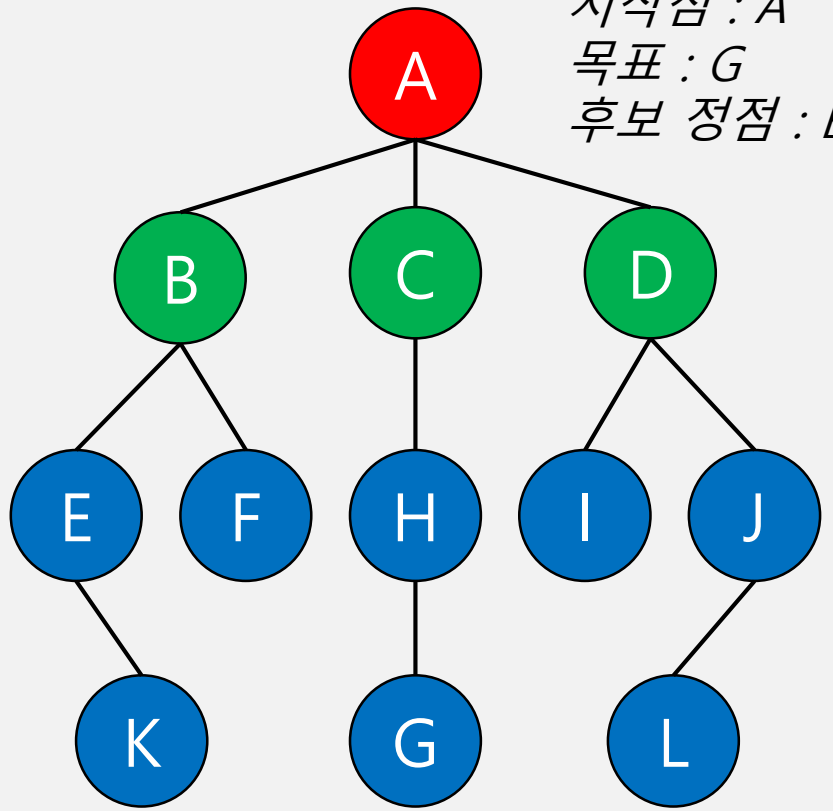
그래프

너비 우선 탐색 (Breadth First Search : BFS)

- 정점을 탐색할 때 시작점에 가까운 정점부터 차례로 탐색하는 방법 (Queue 구조를 사용)
- 목표가 시작점에 가까이 있을 수록 탐색이 빨리 종료됨



시작점 : A
목표 : G

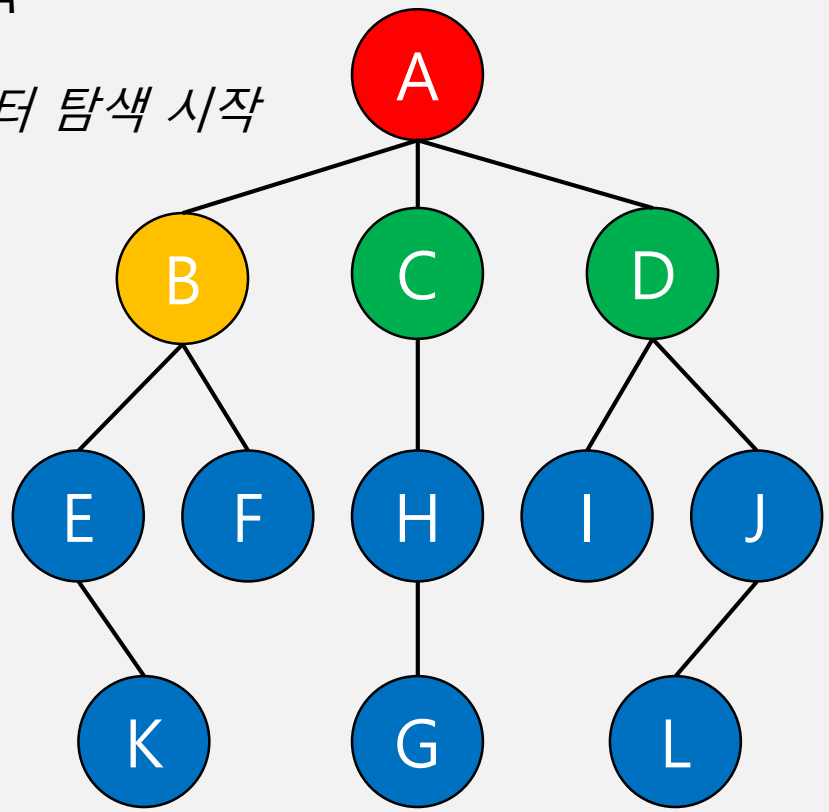


시작점 : A
목표 : G
후보 정점 : B, C, D

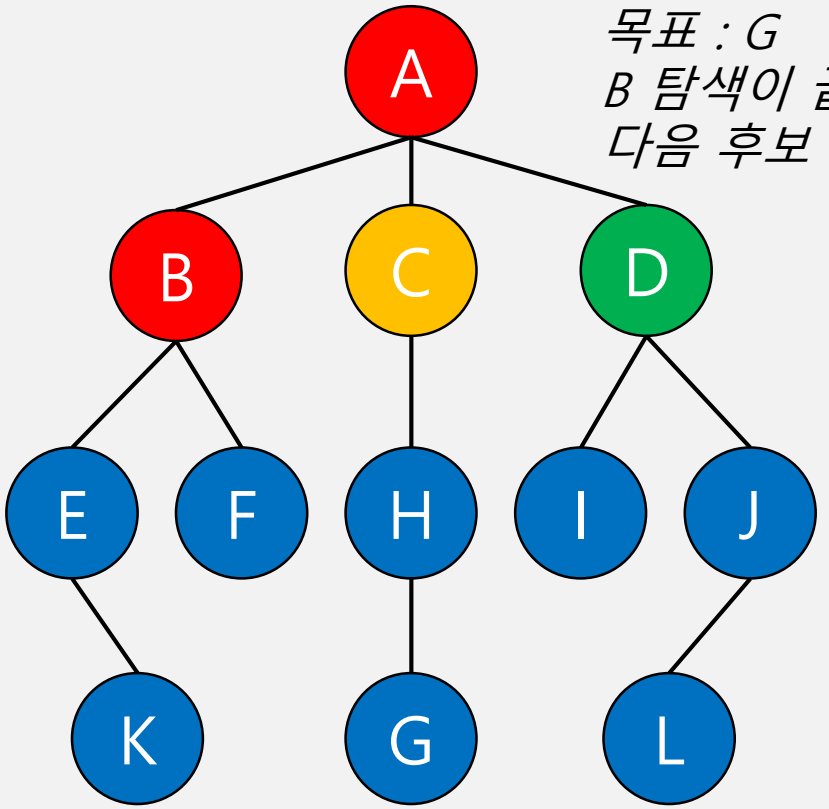
그래프

너비 우선 탐색 (BFS)

시작점 : A
목표 : G
후보 B부터 탐색 시작



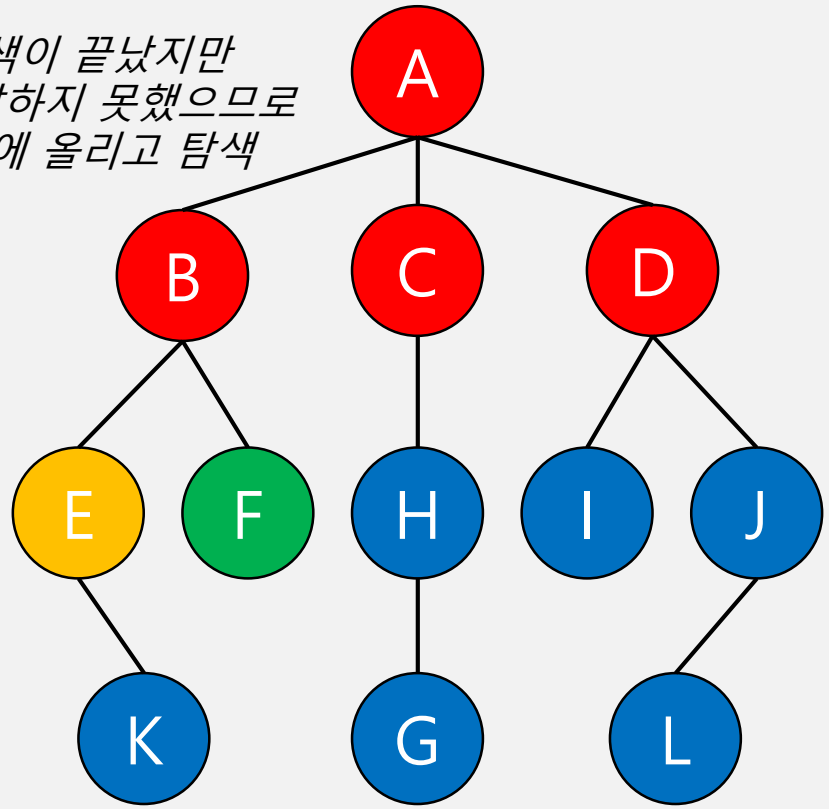
시작점 : A
목표 : G
B 탐색이 끝났으므로
다음 후보 C 탐색 시작



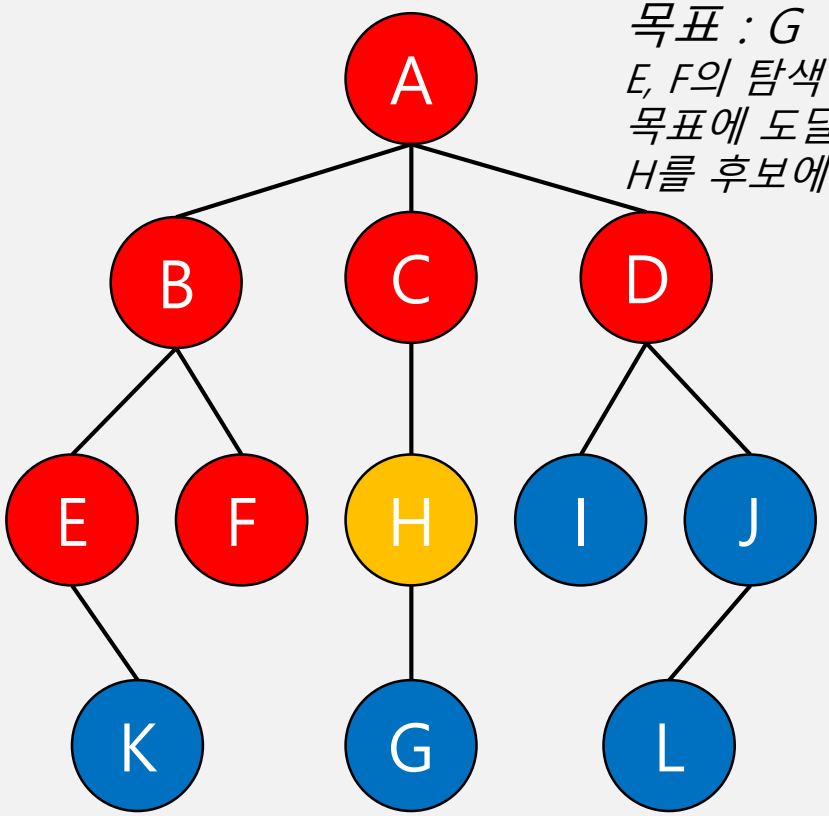
그래프

너비 우선 탐색 (BFS)

시작점 : B
목표 : G
B,C,D의 탐색이 끝났지만
목표에 도달하지 못했으므로
E, F를 후보에 올리고 탐색



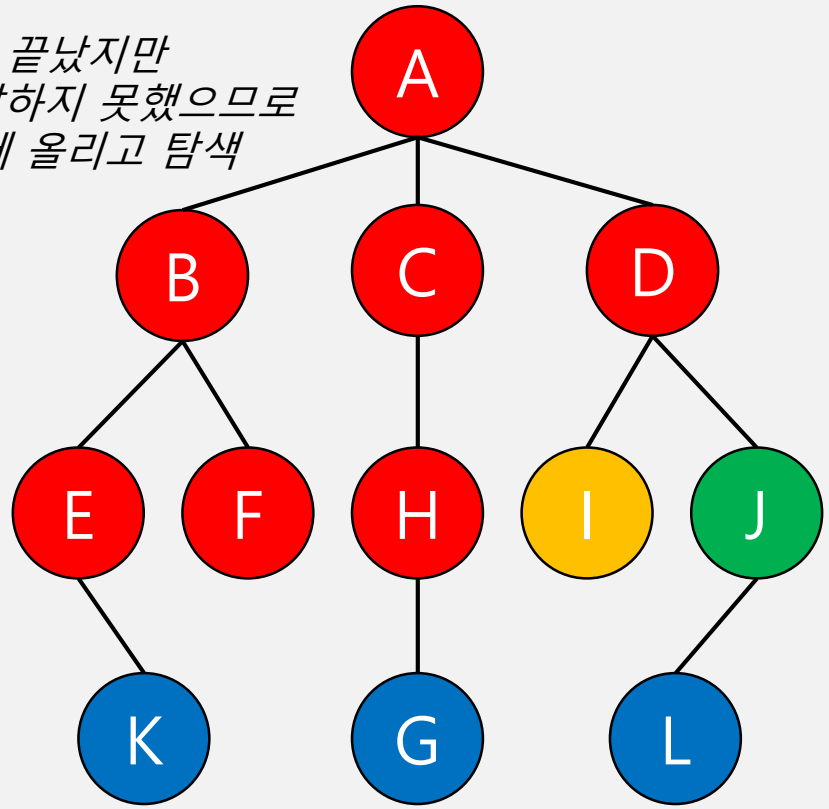
시작점 : C
목표 : G
E, F의 탐색이 끝났지만
목표에 도달하지 못했으므로
H를 후보에 올리고 탐색



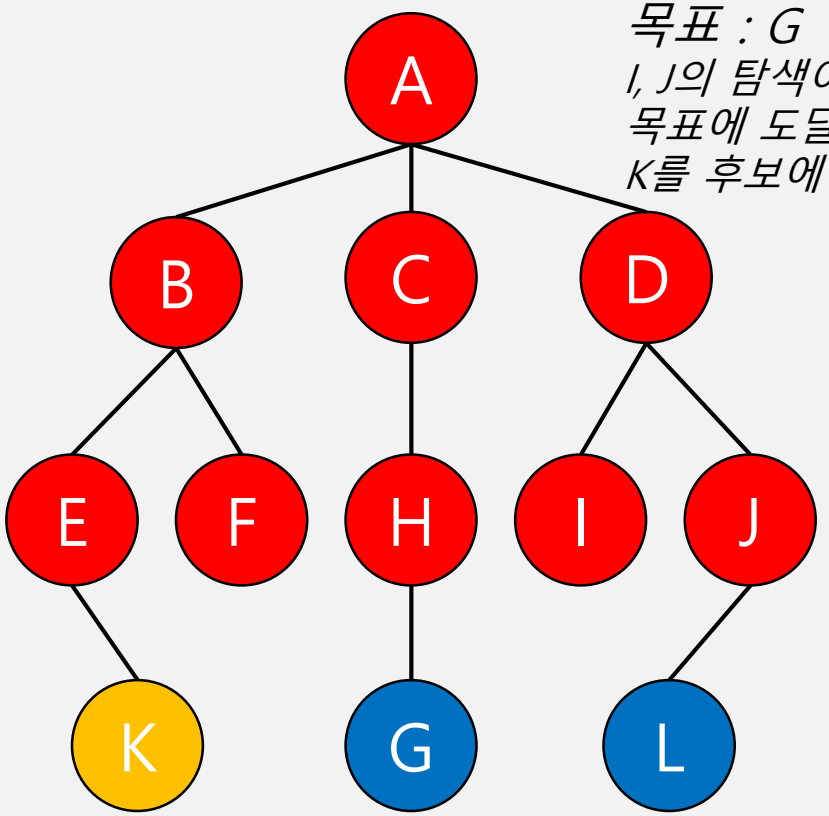
그래프

너비 우선 탐색 (BFS)

시작점 : D
목표 : G
H의 탐색이 끝났지만
목표에 도달하지 못했으므로
I, J를 후보에 올리고 탐색



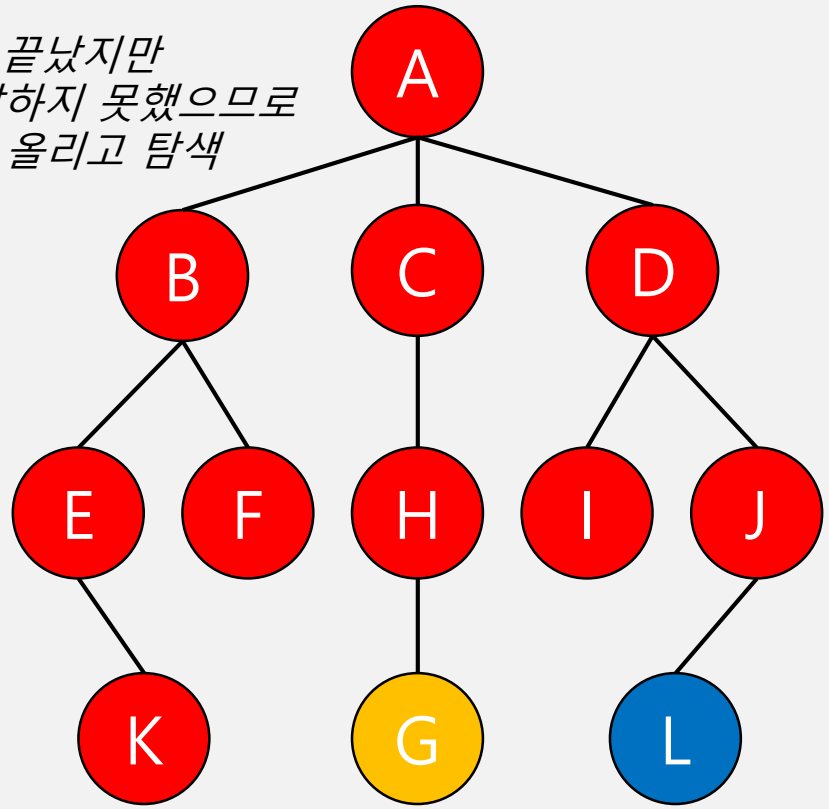
시작점 : E
목표 : G
I, J의 탐색이 끝났지만
목표에 도달하지 못했으므로
K를 후보에 올리고 탐색



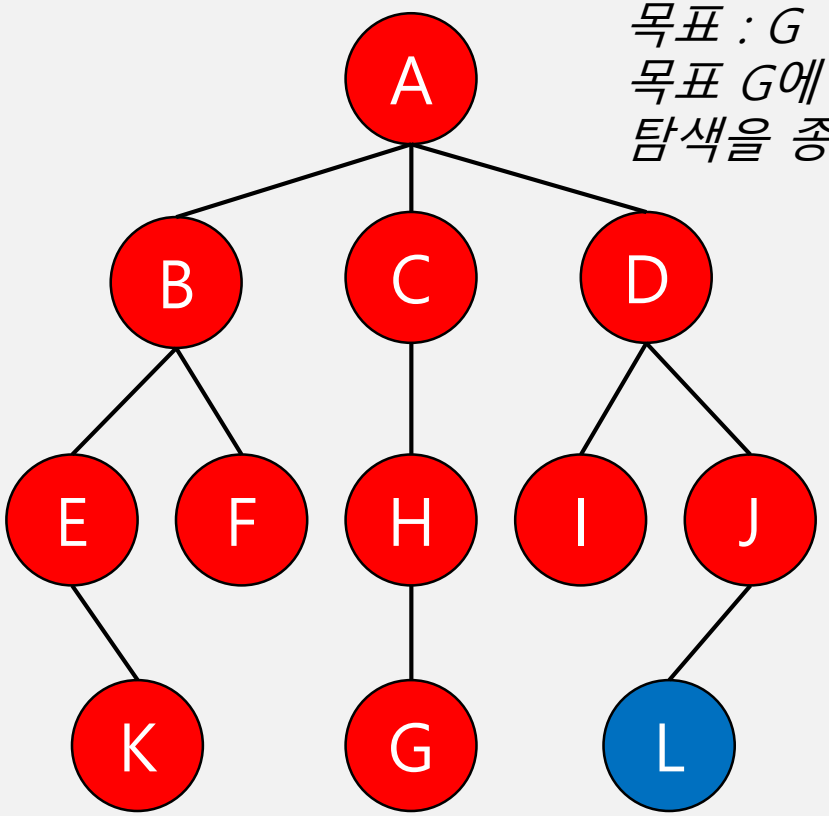
그래프

너비 우선 탐색 (BFS)

시작점 : H
목표 : G
K의 탐색이 끝났지만
목표에 도달하지 못했으므로
G를 후보에 올리고 탐색



시작점 : H
목표 : G
목표 G에 도달했으므로
탐색을 종료



그래프

너비 우선 탐색 (BFS)

Queue를 사용

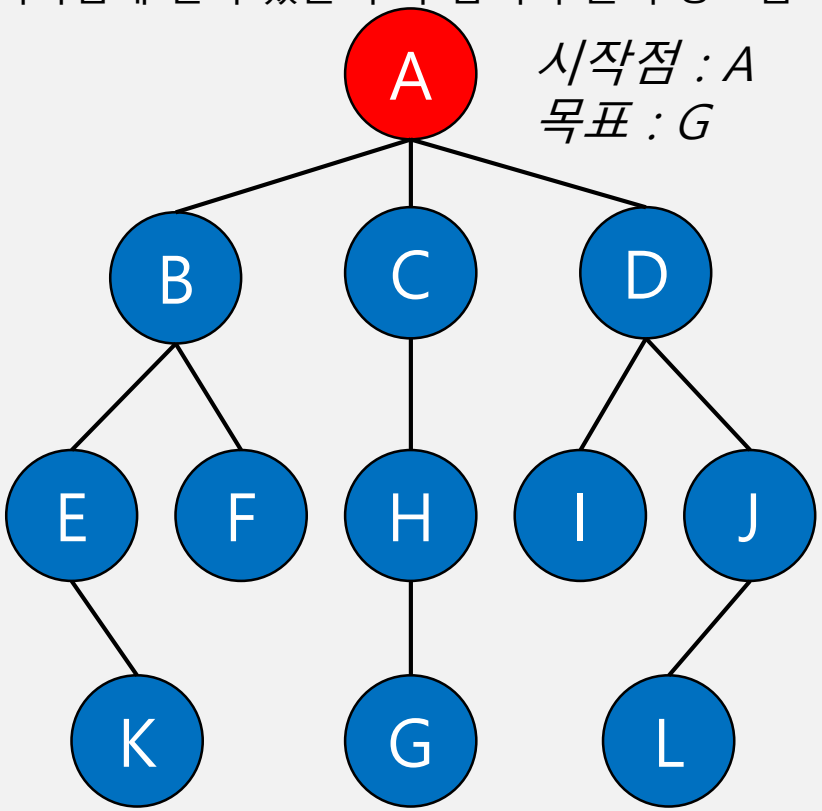
```
public static void main(String[] args) {
    int[][] graph = {
            {1, 2, 3}, //A노드
            {4, 5}, //B노드
            {6}, //C노드
            {7, 8}, //D노드
            {9}, //E노드
            {}, //F노드
            {10}, //H노드
            {}, //I노드
            {11}, // J노드
            {}, //K노드
            {}, //G노드
            {} //L노드
    };

    BFS(graph, 0);
}
```

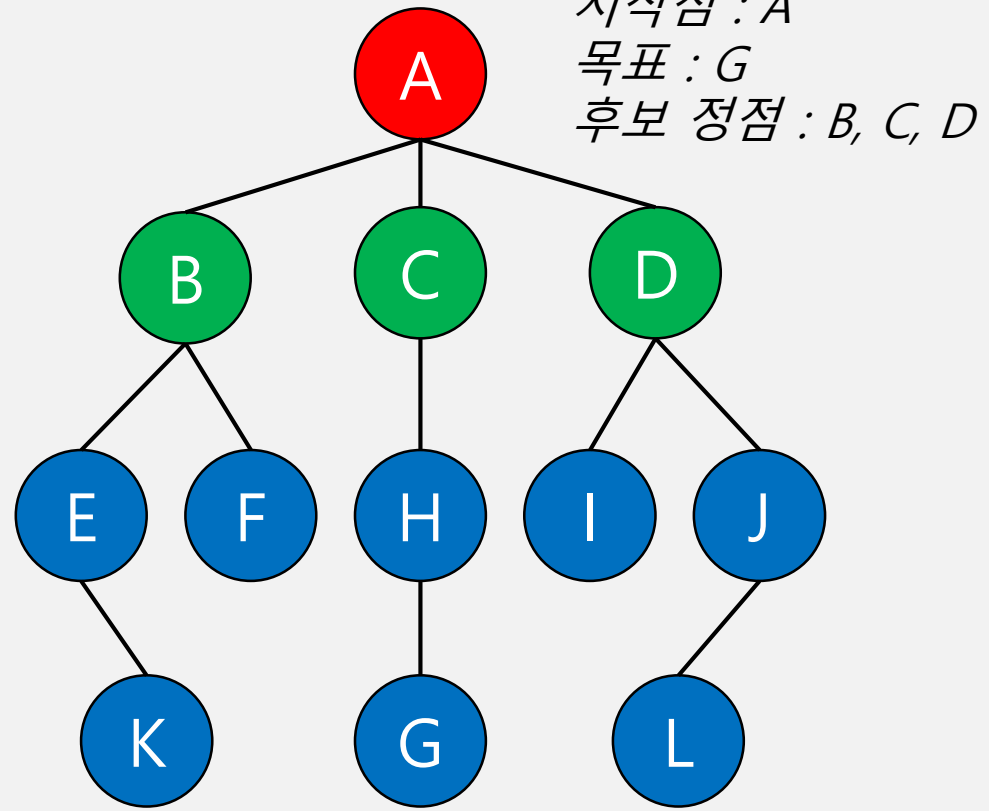
```
static void BFS(int[][] graph, int start_node) {
    ArrayList<Integer> visit = new ArrayList<>();
    Queue<Integer> queue = new LinkedList<>();

    queue.offer(start_node);

    while(!queue.isEmpty()) {
        int node = queue.poll();

        if(!visit.contains(node)) {
            visit.add(node);
            for (int i = 0; i < graph[node].length; i++) {
                queue.offer(graph[node][i]);
            }
        }
    }
    for (int i = 0; i < visit.size(); i++) {
        System.out.println(i + 1 +"번째 방문 : " + visit.get(i));
    }
}
```

그래프

깊이 우선 탐색 (Depth First Search : DFS)

- 정점을 탐색할 때 하나의 길을 끝까지 따라가며 차례로 탐색하는 방법 (Stack 구조를 사용)
- 목표가 시작점에 멀리 있을 수록 탐색이 빨리 종료됨

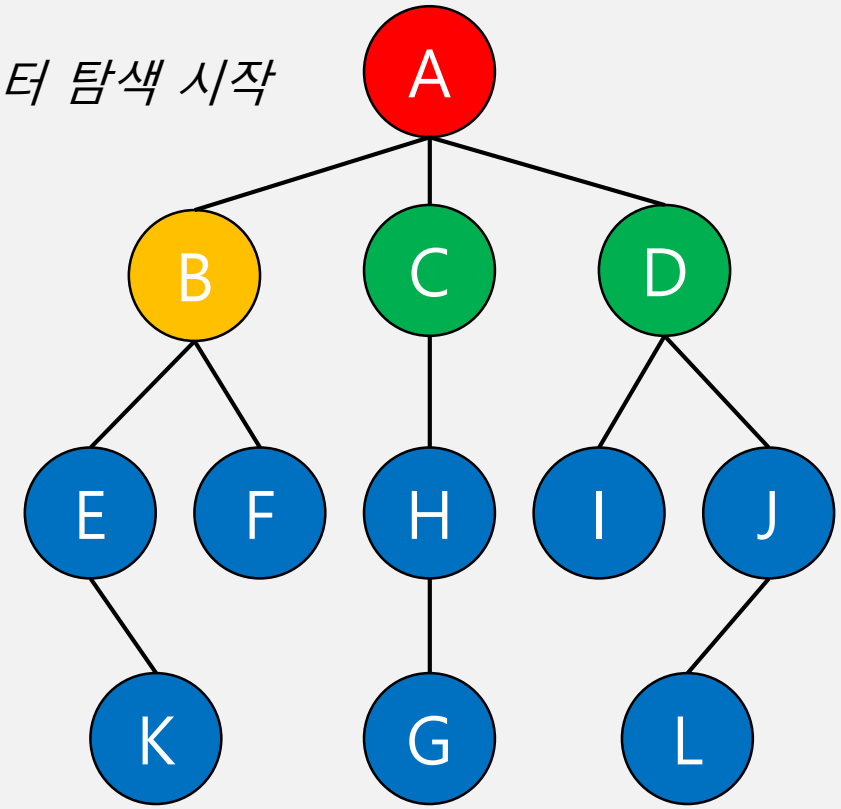




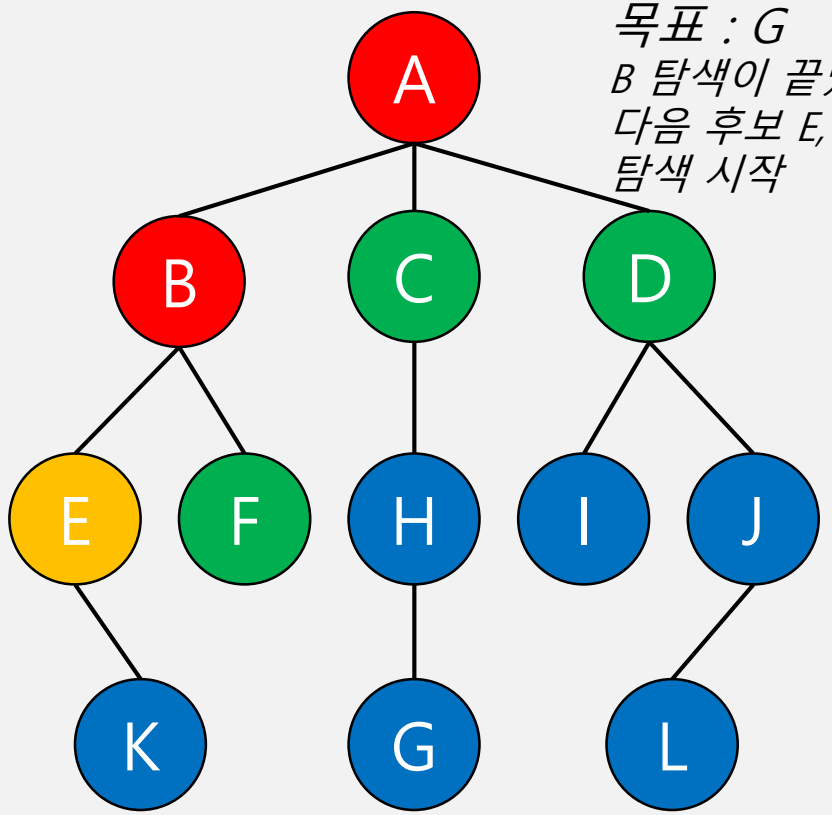
그래프

깊이 우선 탐색 (DFS)

시작점 : B
목표 : G
후보 B부터 탐색 시작



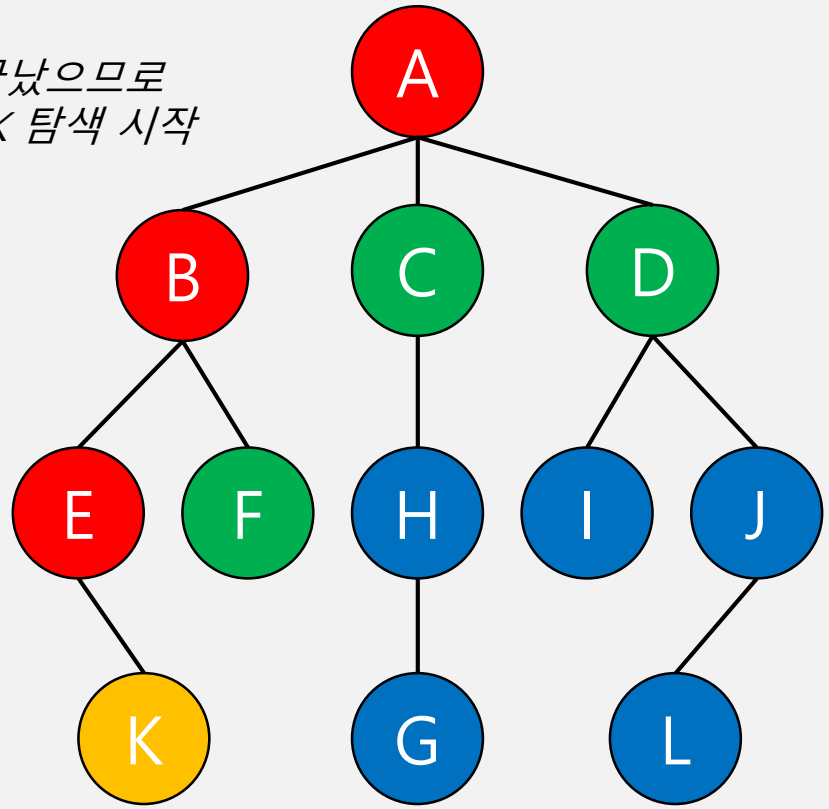
시작점 : B
목표 : G
B 탐색이 끝났으므로
다음 후보 E, F를 후보로 하여
탐색 시작



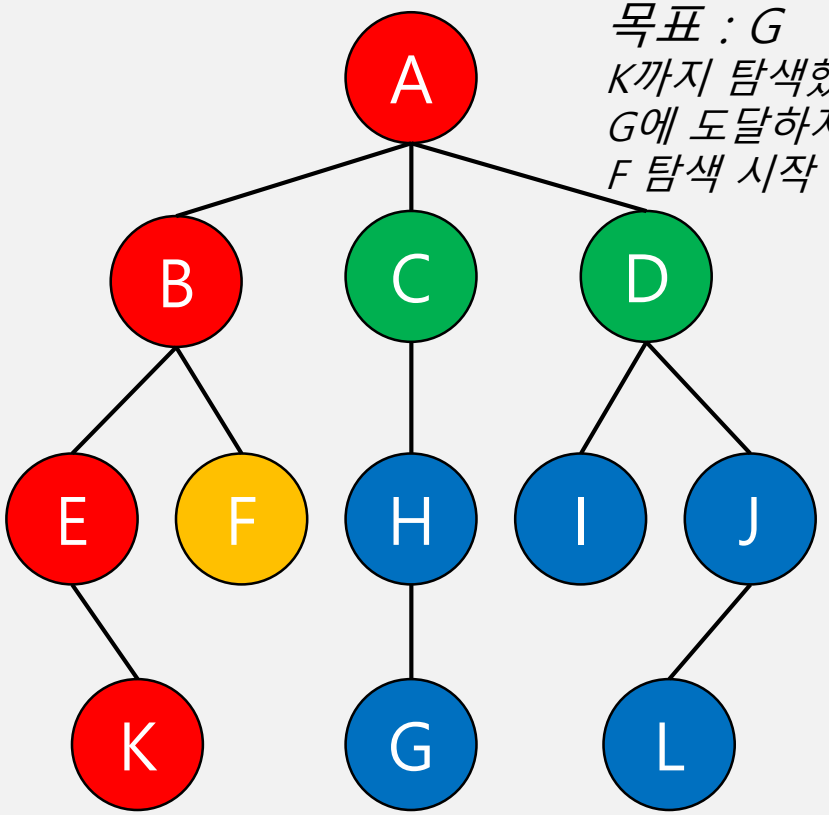
그래프

깊이 우선 탐색 (DFS)

시작점 : E
목표 : G
E 탐색이 끝났으므로
다음 후보 K 탐색 시작



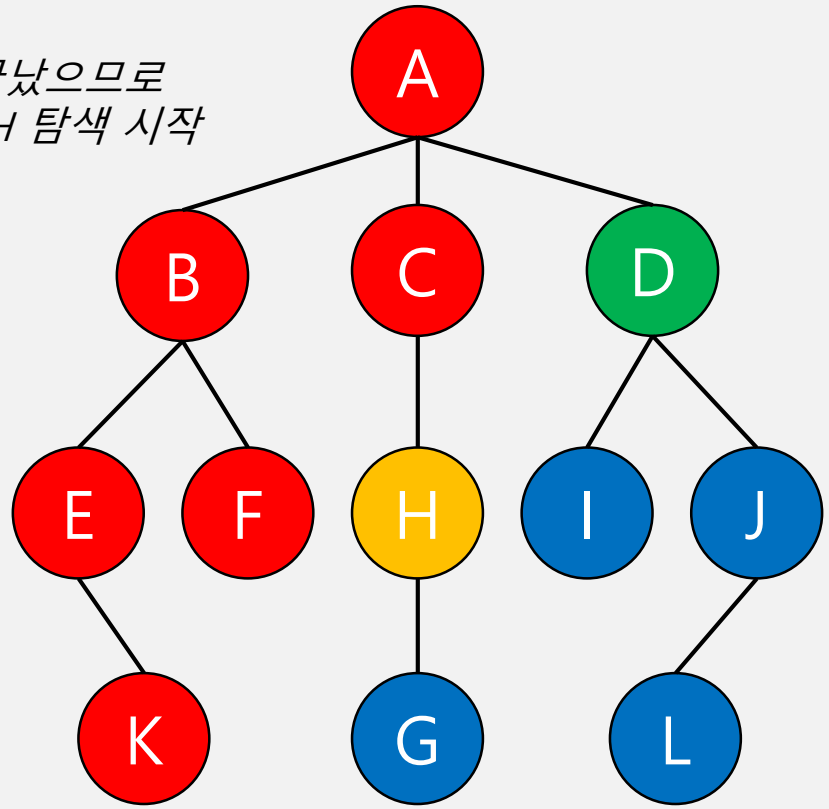
시작점 : B
목표 : G
K까지 탐색했으나
G에 도달하지 못하였으므로
F 탐색 시작



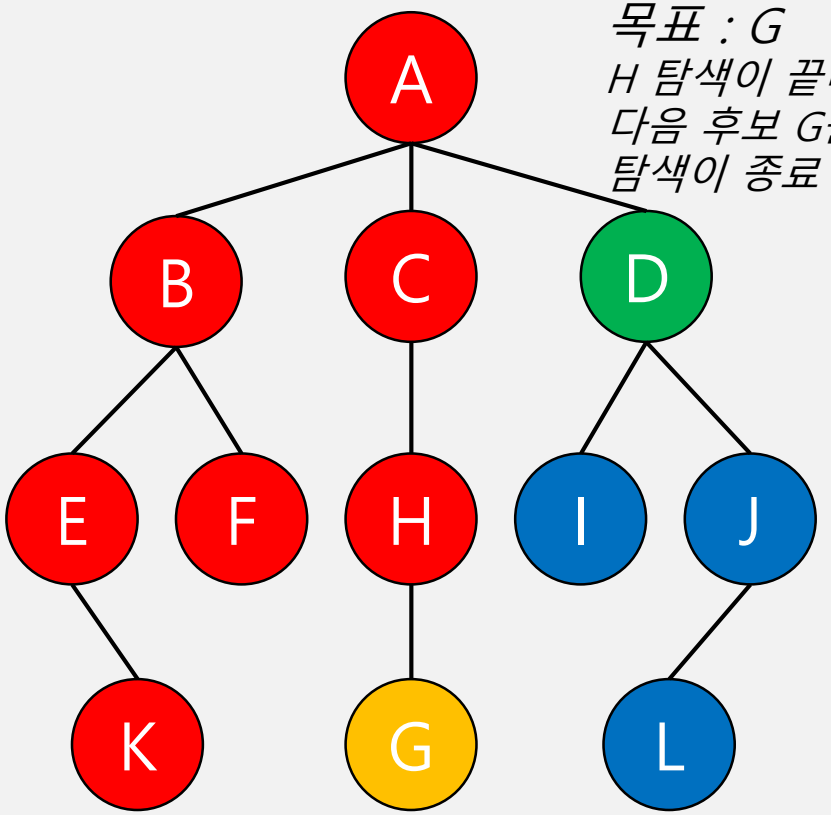
그래프

깊이 우선 탐색 (DFS)

시작점 : E
목표 : G
F 탐색이 끝났으므로
다음 후보 H 탐색 시작



시작점 : B
목표 : G
H 탐색이 끝나고
다음 후보 G를 탐색해
탐색이 종료



그래프

깊이 우선 탐색 (DFS)

Stack을 사용

```
public static void main(String[] args) {
    int[][] graph = {
            {1, 2, 3}, //A노드
            {4, 5}, //B노드
            {6}, //C노드
            {7, 8}, //D노드
            {9}, //E노드
            {}, //F노드
            {10}, //H노드
            {}, //I노드
            {11}, // J노드
            {}, //K노드
            {}, //G노드
            {} //L노드
    };

    DFS(graph, 0);
}
```

```
static void DFS(int[][] graph, int start_node) {
    ArrayList<Integer> visit = new ArrayList<>();
    Stack<Integer> stack = new Stack<>();

    stack.push(start_node);

    while(!stack.isEmpty()) {
        int node = stack.pop();

        if(!visit.contains(node)) {
            visit.add(node);
            for (int i = 0; i < graph[node].length; i++) {
                stack.push(graph[node][i]);
            }
        }
    }
    for (int i = 0; i < visit.size(); i++) {
        System.out.println(i + 1 +"번째 방문 : " + visit.get(i));
    }
}
```

그래프

깊이 우선 탐색 (DFS)

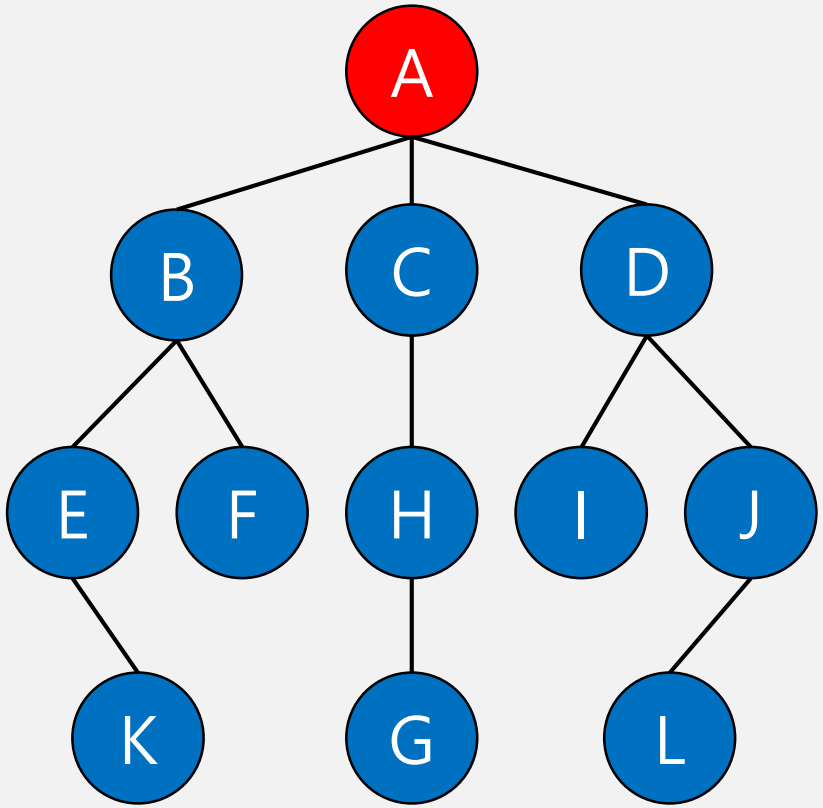
재귀함수를 사용

```
static ArrayList<Integer> recursive_DFS(ArrayList<Integer> visited, int[][] graph, int start_node) {
    visited.add(start_node);

    for (int i = 0; i < graph[start_node].length; i++) {
        if(!visited.contains(graph[start_node][i])) {
            recursive_DFS(visited, graph, graph[start_node][i]);
        }
    }

    return visited;
}
```

그래프

시간복잡도 – (BFS / DFS)

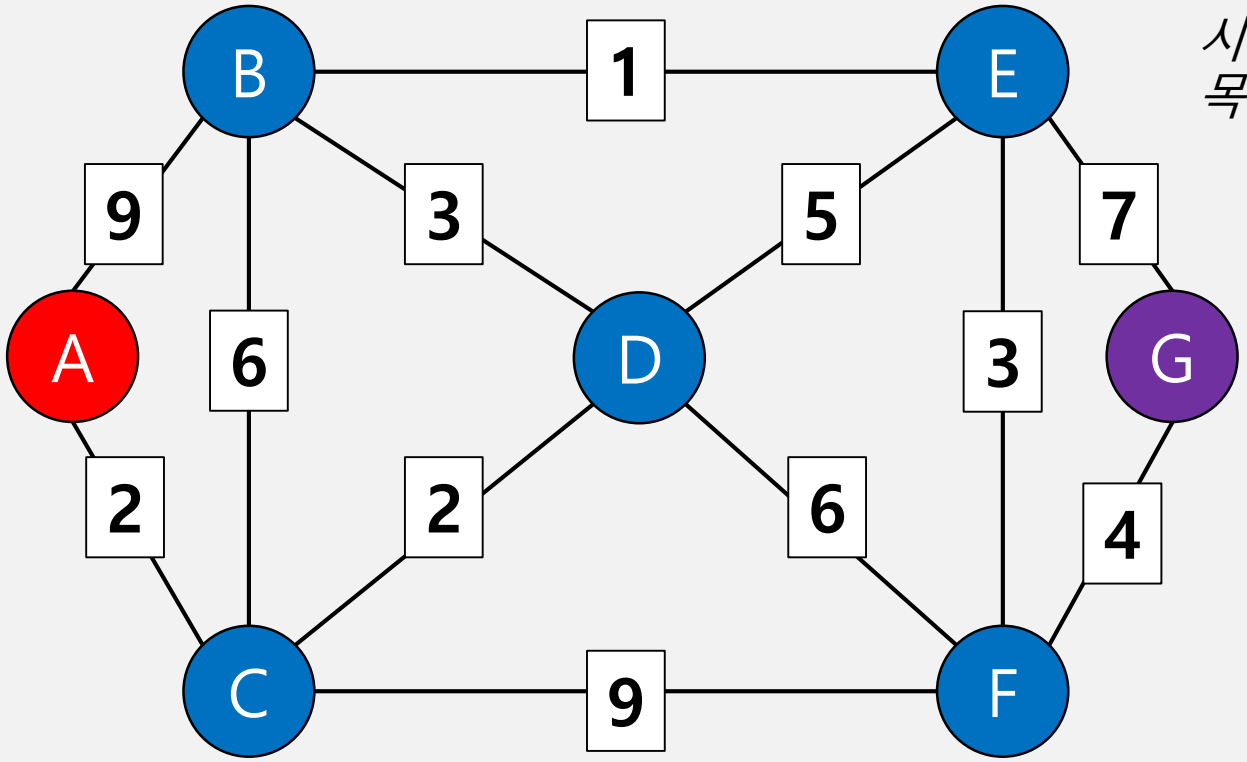
- 정점의 수가 n, 간선의 수가 m일 때,
- 모든 정점을 따라 움직이기 때문에 실행 시간은 최소 O(m)
- 큐에 정점을 추가하는 것은 O(1), 모든 정점을 추가하는 것은 O(n)
- 따라서 시간복잡도 = O(n + m)



그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)

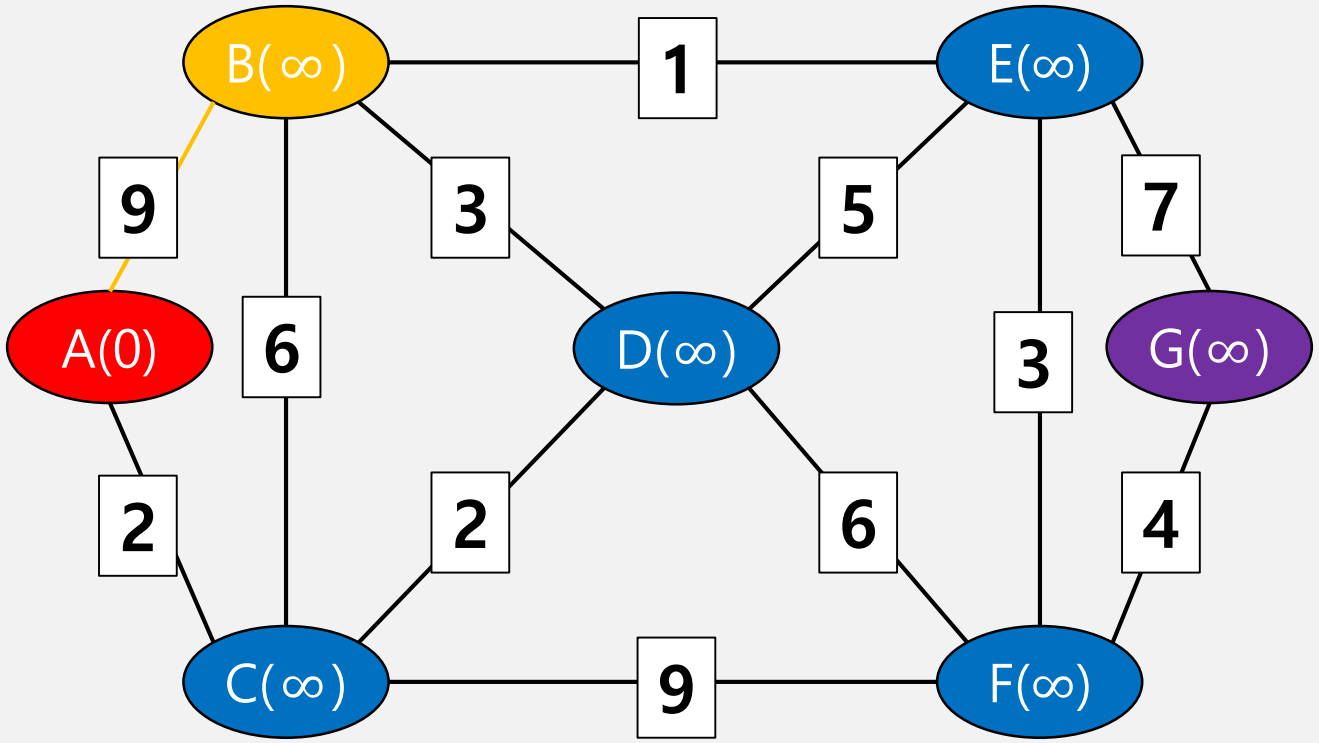
- 그래프의 최단 경로 문제를 해결하기 위한 알고리즘
- 간선에 가중치가 부여된 '가중 그래프'가 주어지며, '시작점'과 '종점'이 지정됨
- 시작점부터 종점까지의 경로 중 간선의 가중치의 합이 가장 작은 것을 구함



시작점 : A
목표 : G

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)

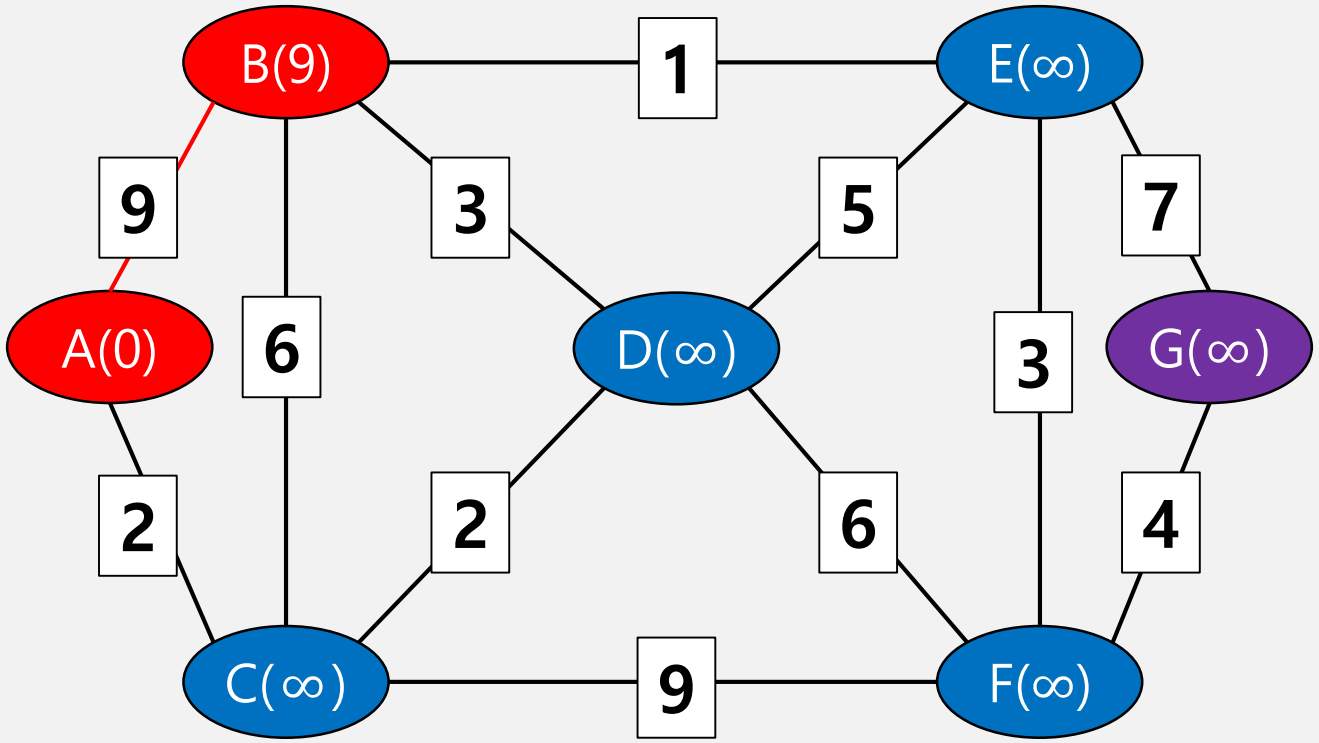


시작점 : A
목표 : G

초기 가중치 설정 :
시작점은 0,
그 외는 무한대(∞)
(도착할 수 있을지 모름)

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)



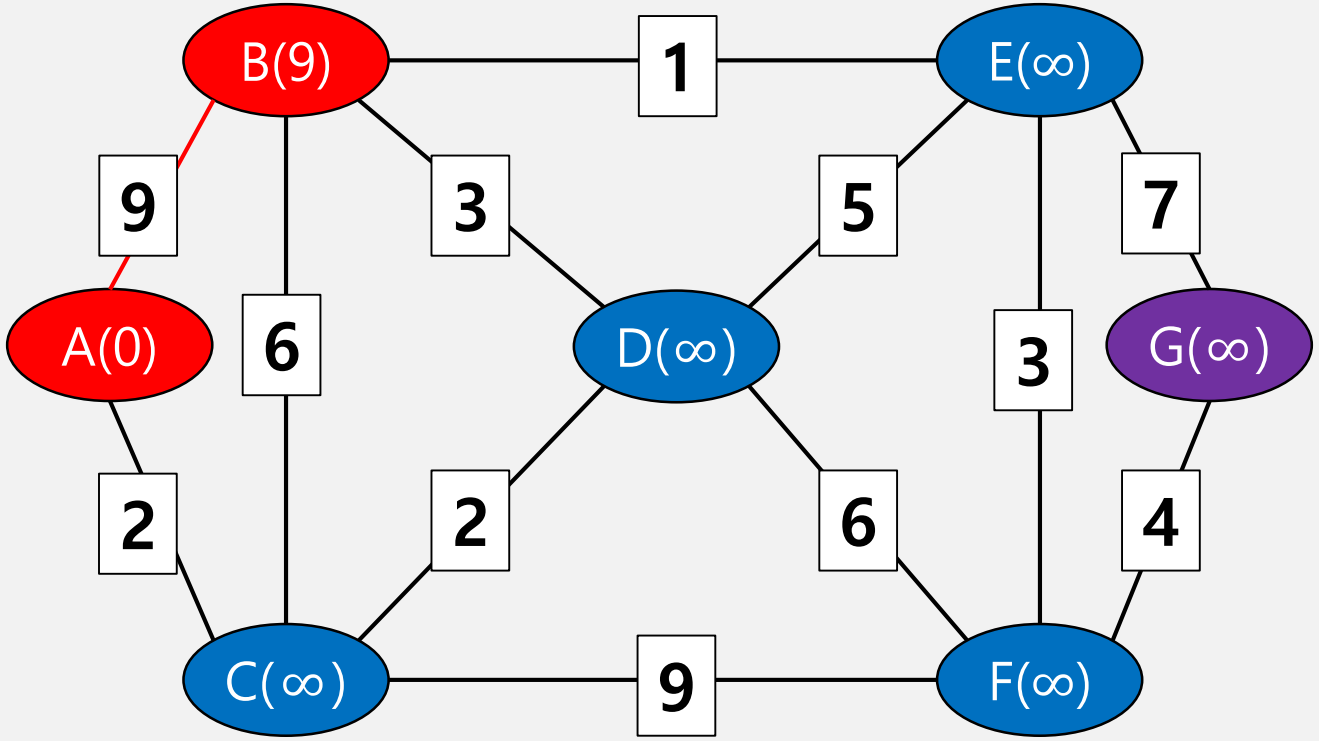
시작점 : A
목표 : G

A와 연결된 임의의 정점(B) 선택
A -> B로 가는 가중치 계산
(0 + 9) = 9

현재 정점의 값과 비교하여
더 낮은 값으로 변경 => 9

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)



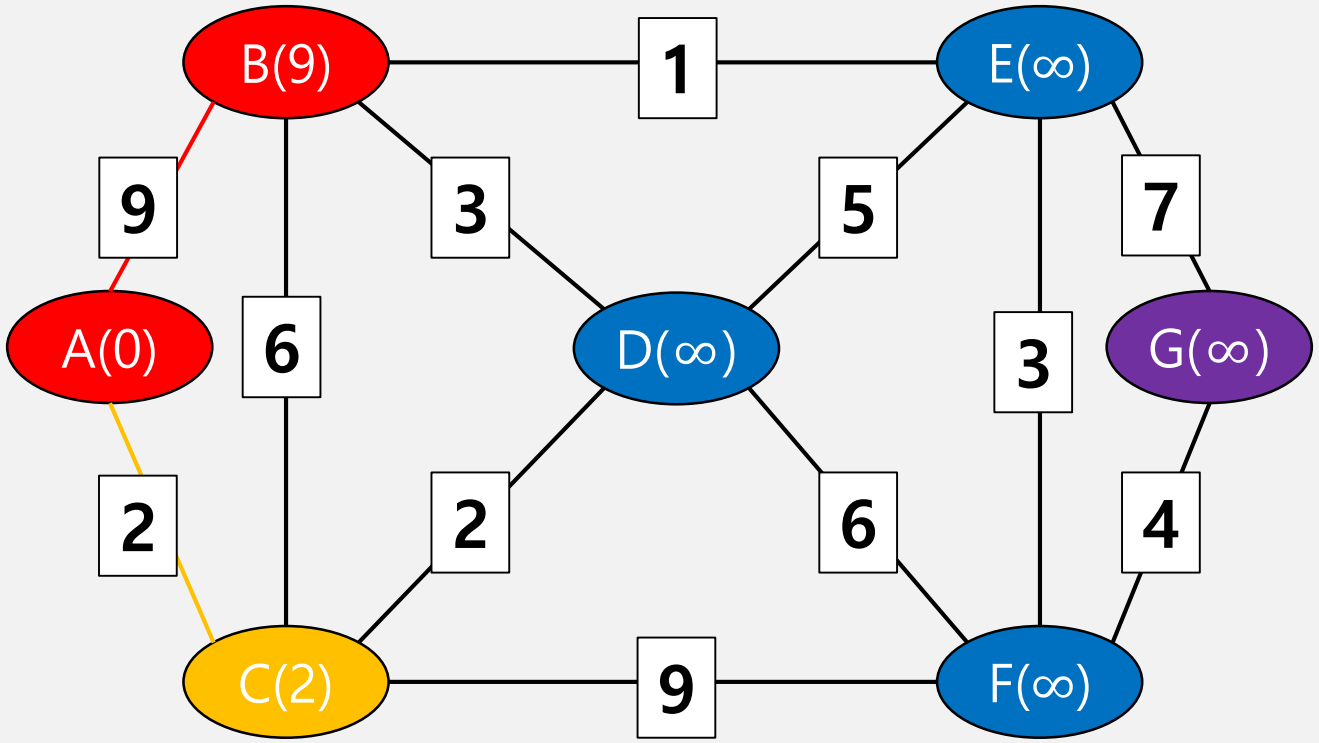
시작점 : A
목표 : G

B의 값을 변경 후 반대인
B -> A로 가는 가중치 계산
(9 + 9) = 18

현재 정점의 값과 비교하여
더 낮은 값으로 변경 => 0

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)

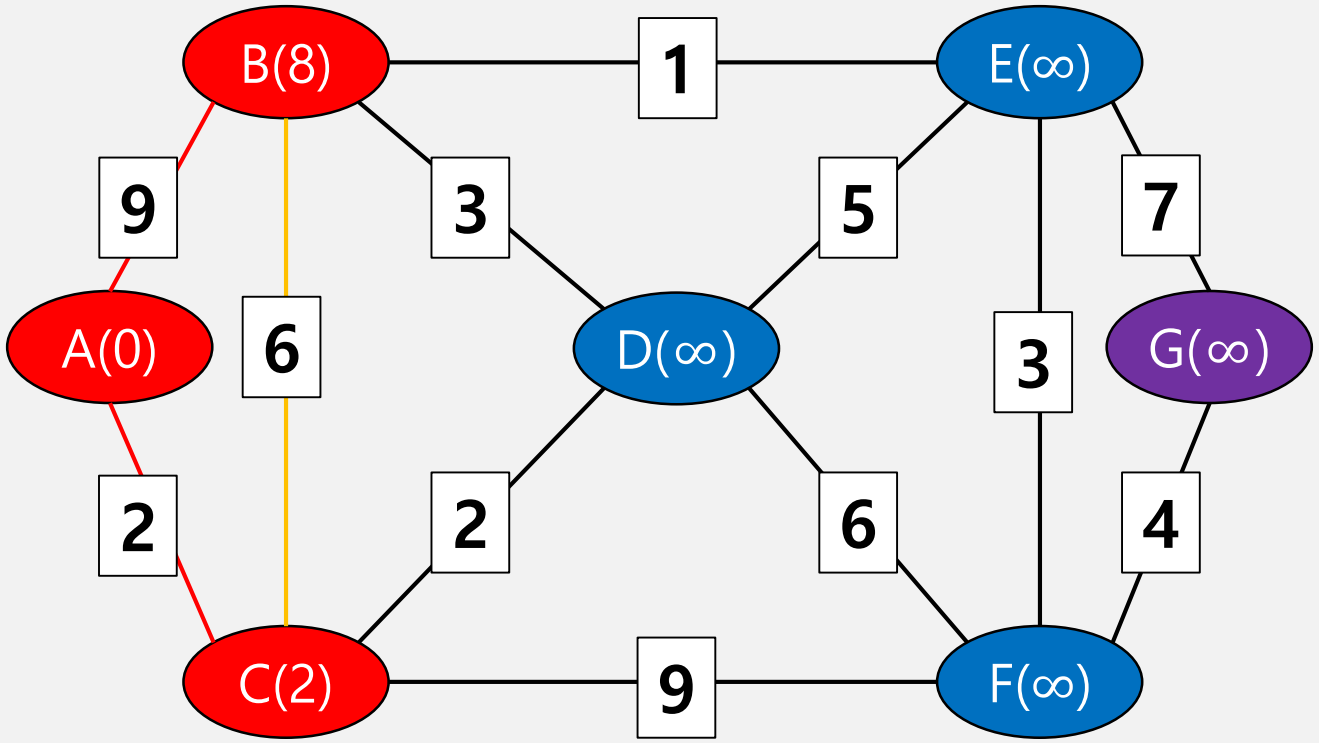


A와 연결된 임의의 정점(C) 선택
A -> C로 가는 가중치 계산
(0 + 2) = 2

현재 정점의 값과 비교하여
더 낮은 값으로 변경 => 2

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)



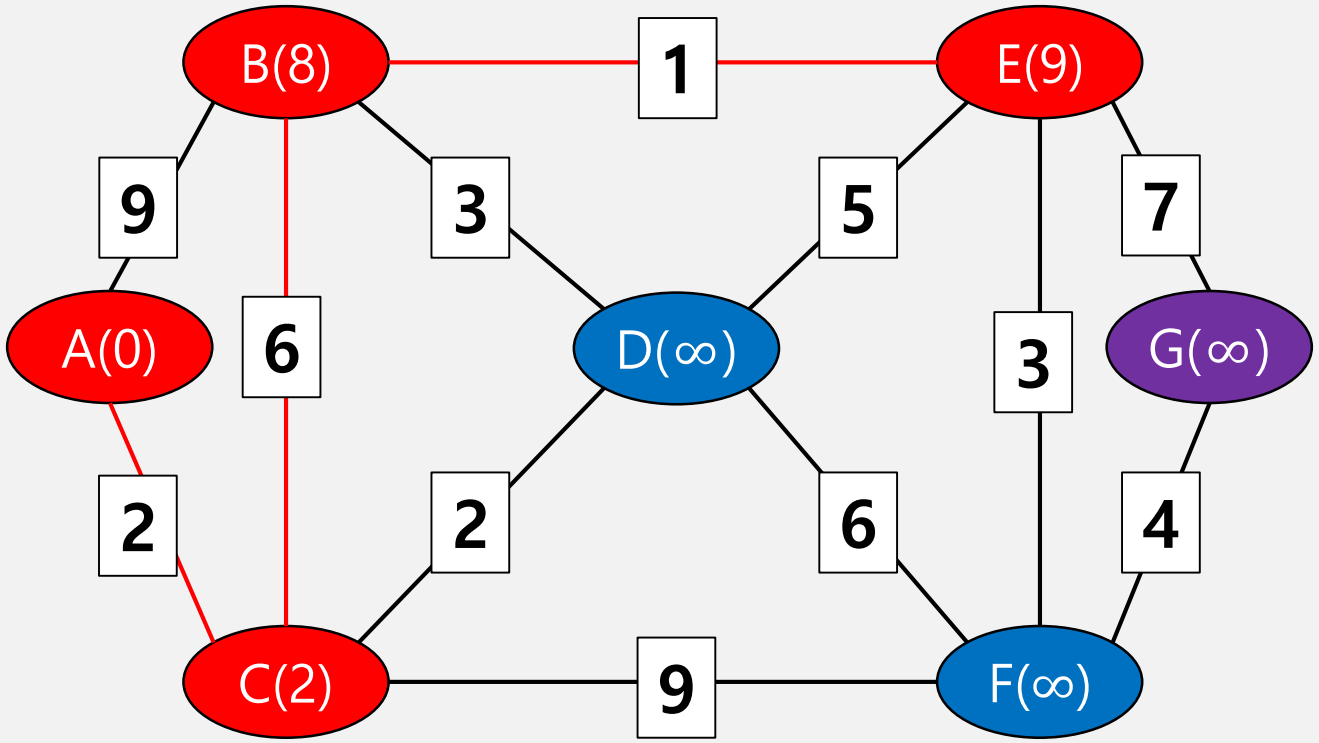
B -> C로 가는 가중치 계산
(9+ 6) = 15

C -> B로 가는 가중치 계산
(2+ 6) = 8

현재 정점의 값과 비교하여
더 낮은 값으로 변경 =>
(A -> B) < (A -> C -> B)
8로 변경

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)



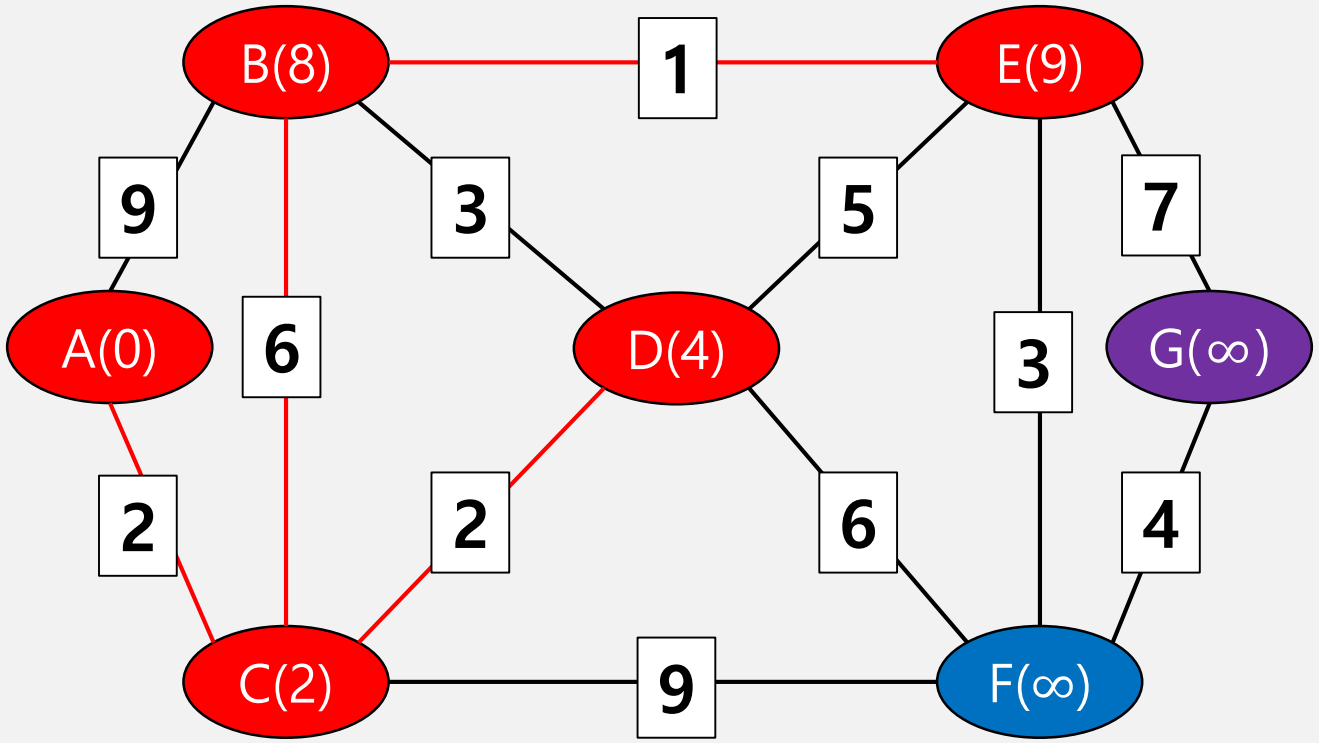
B -> C로 가는 가중치 계산
(9+ 6) = 15

C -> B로 가는 가중치 계산
(2+ 6) = 8

현재 정점의 값과 비교하여
더 낮은 값으로 변경 =>
(A -> B) < (A -> C -> B)
8로 변경

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)



E -> D로 가는 가중치 계산
(9+2) = 11

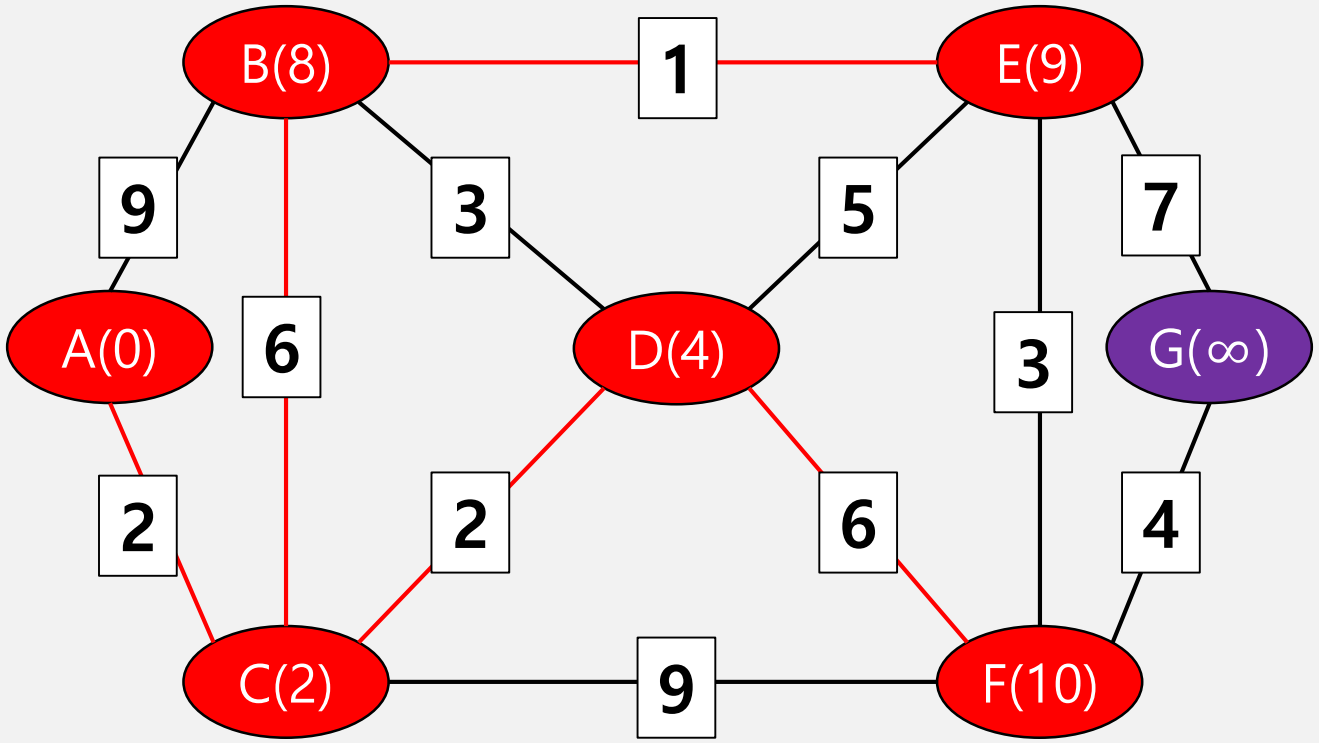
B -> D로 가는 가중치 계산
(8+2) = 10

C -> D로 가는 가중치 계산
(2+2) = 4

현재 정점의 값과 비교하여 더 낮은 값으로 변경 => 4

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)



E -> F로 가는 가중치 계산
(9+3) = 12

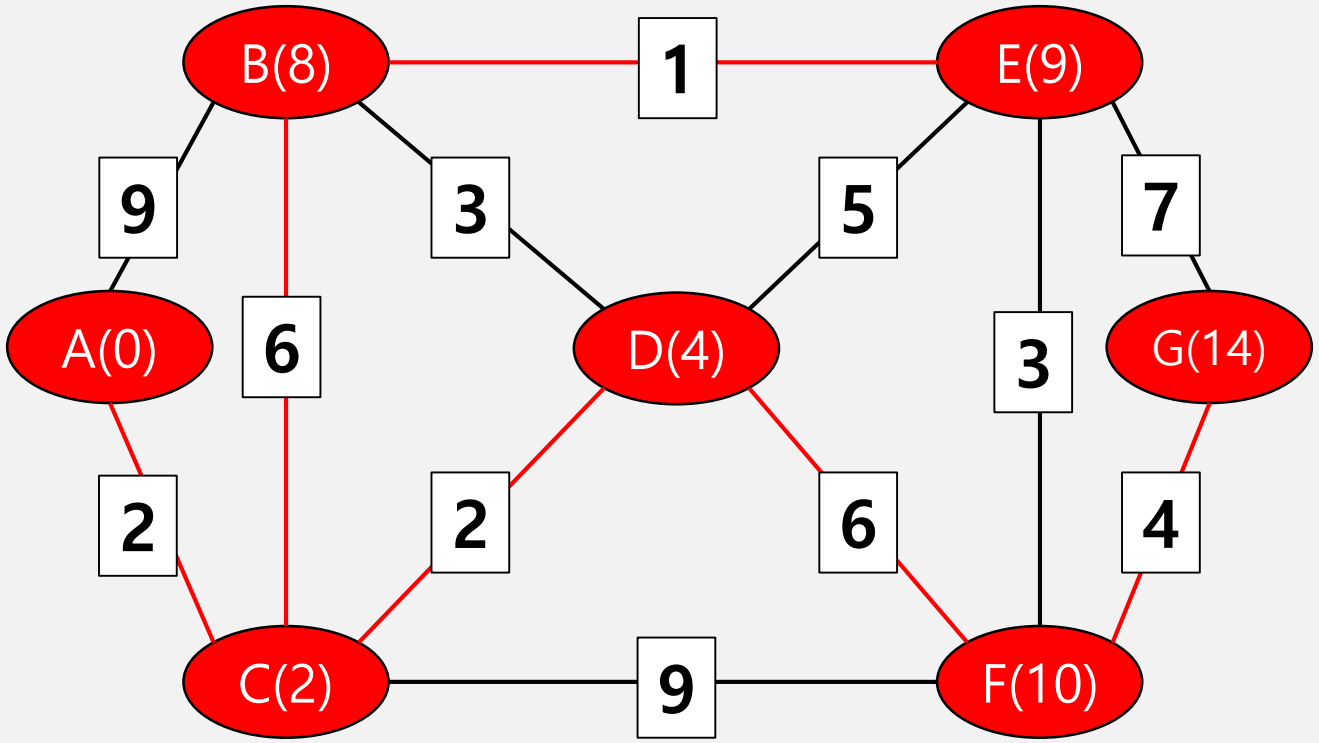
D -> F로 가는 가중치 계산
(4+6) = 10

C -> F로 가는 가중치 계산
(2+9) = 11

현재 정점의 값과 비교하여 더 낮은 값으로 변경 => 10

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)



F -> G로 가는 가중치 계산
(9+7) = 16

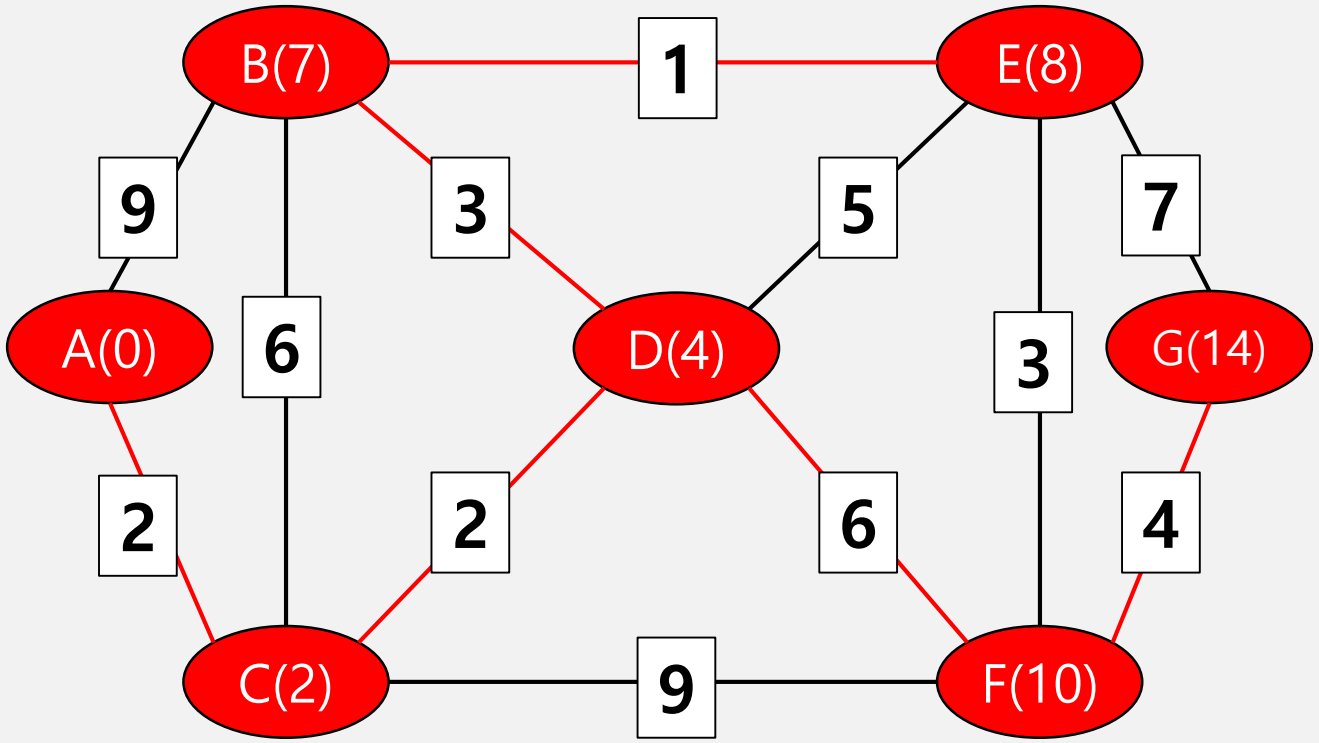
E -> G로 가는 가중치 계산
(10+4) = 14

현재 정점의 값과 비교하여 더 낮은 값으로 변경 => 14

1라운드 종료!

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)

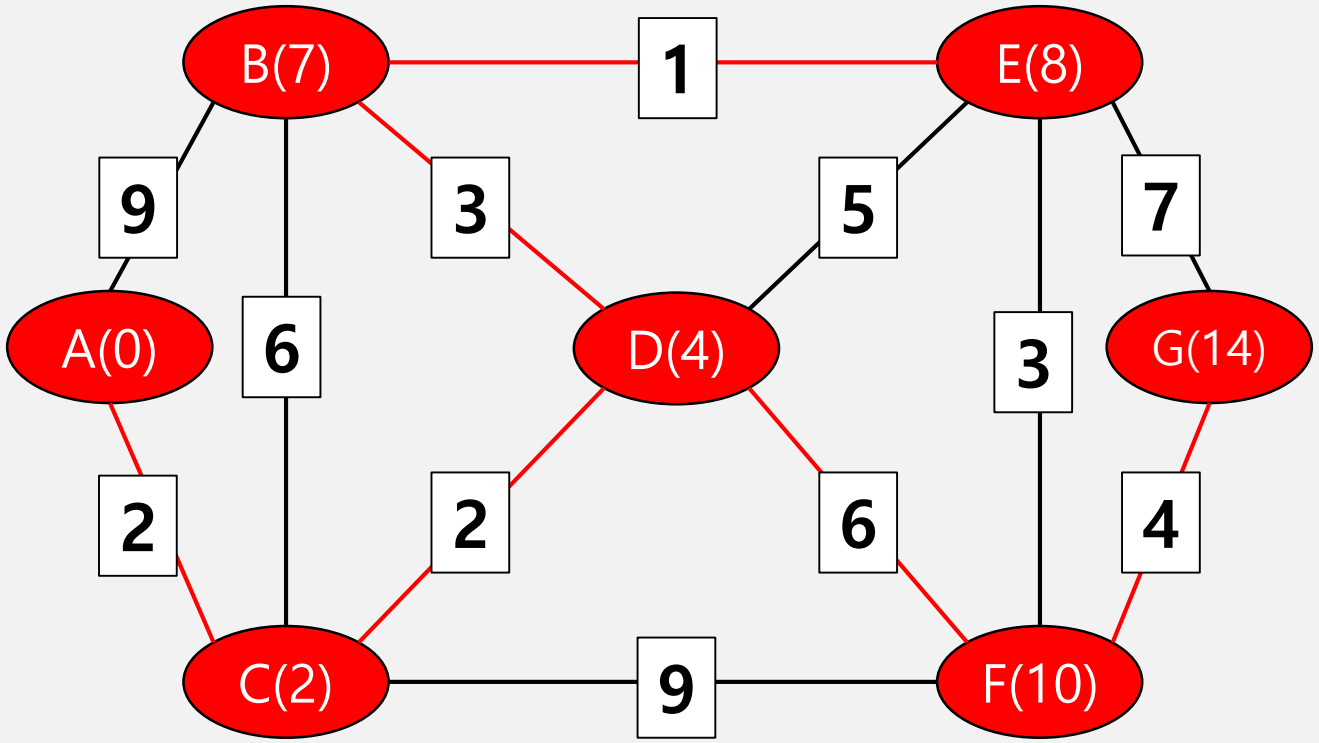


같은 방법으로
가중치가 변경되지 않을 때까지
반복 작업을 진행

2라운드 종료!

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)

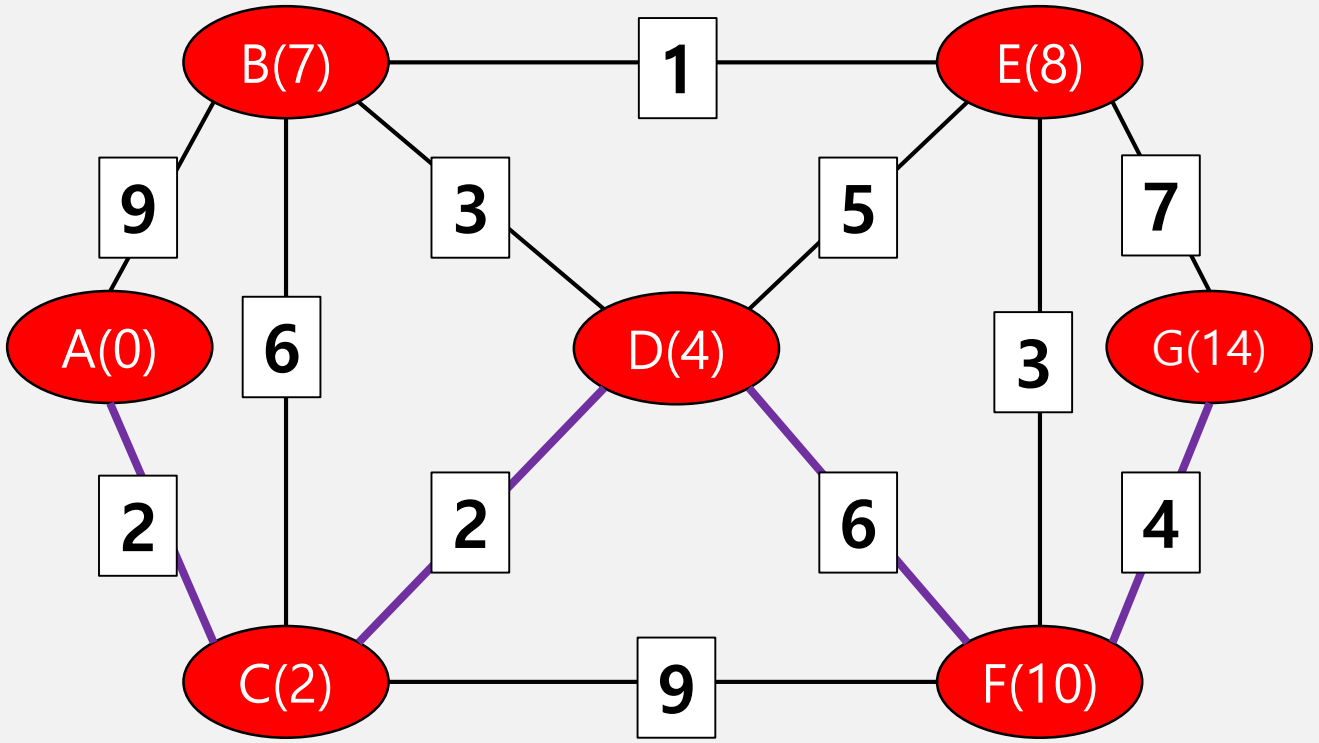


같은 방법으로
가중치가 변경되지 않을 때까지
반복 작업을 진행

3라운드 종료!

그래프

벨먼-포드 알고리즘 (Bellman-Ford algorithm)



같은 방법으로
가중치가 변경되지 않을 때까지
반복 작업을 진행

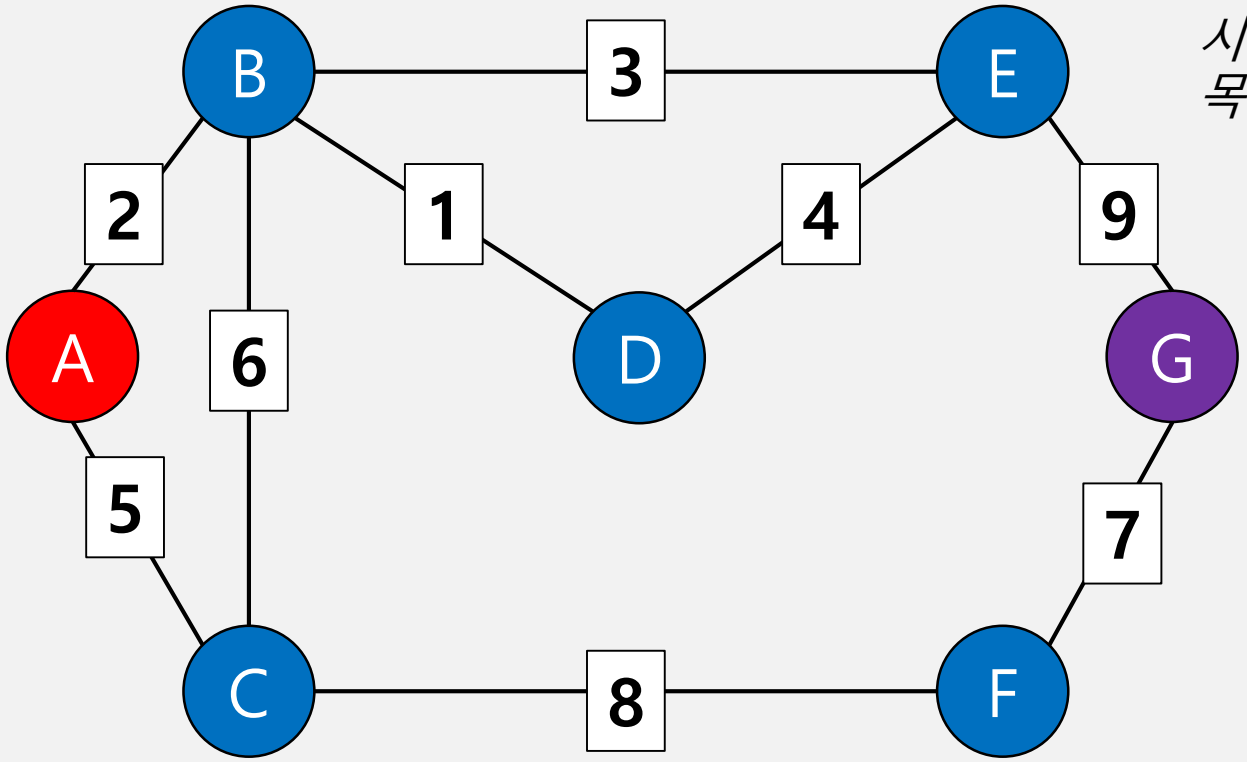
진행했지만 가중치가 변경되지
않았음으로 최단거리 탐색이
완료되었다고 판단

최단거리 탐색 완료

그래프

다익스트라 알고리즘 (Dijkstra algorithm)

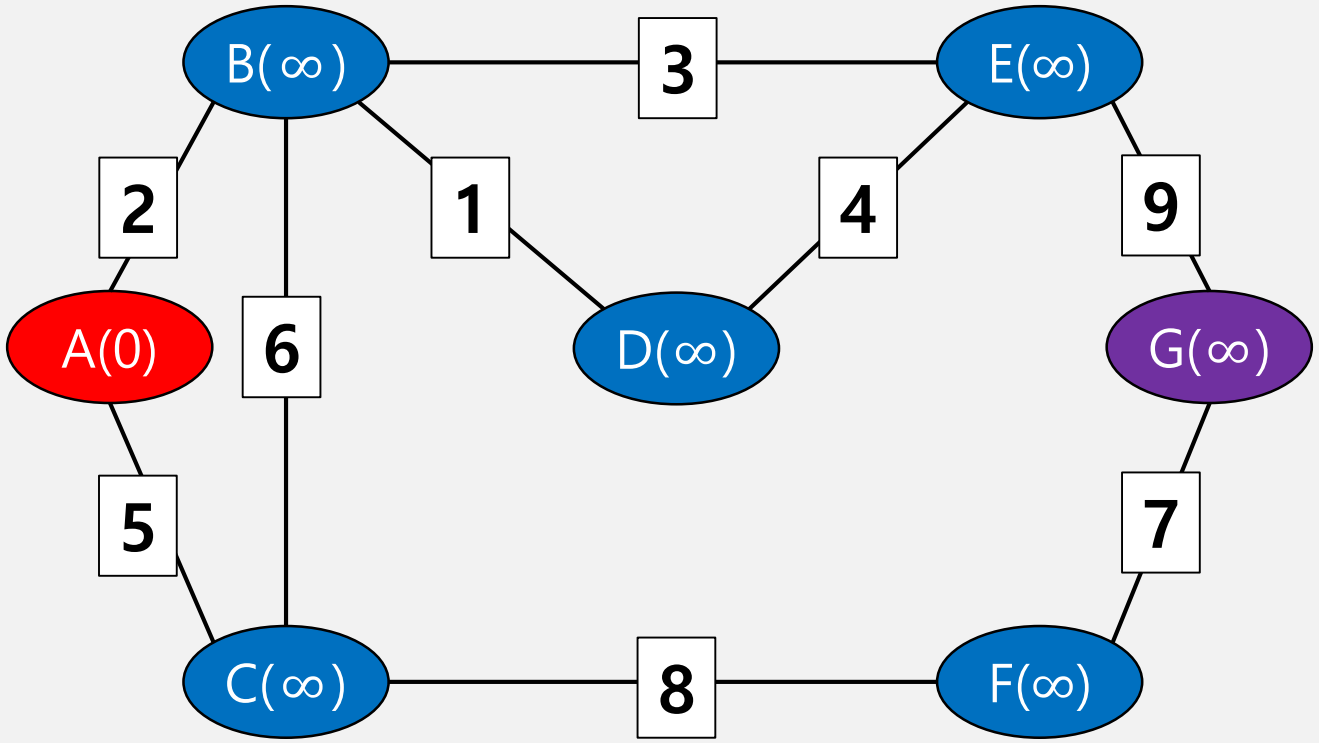
- 그래프의 최단 경로 문제를 해결하기 위한 알고리즘
- 간선에 가중치가 부여된 '가중 그래프'가 주어지며, '시작점'과 '종점'이 지정됨
- 시작점부터 종점까지의 경로 중 간선의 가중치의 합이 가장 작은 것을 구함



시작점 : A
목표 : G

그래프

다익스트라 알고리즘 (Dijkstra algorithm)

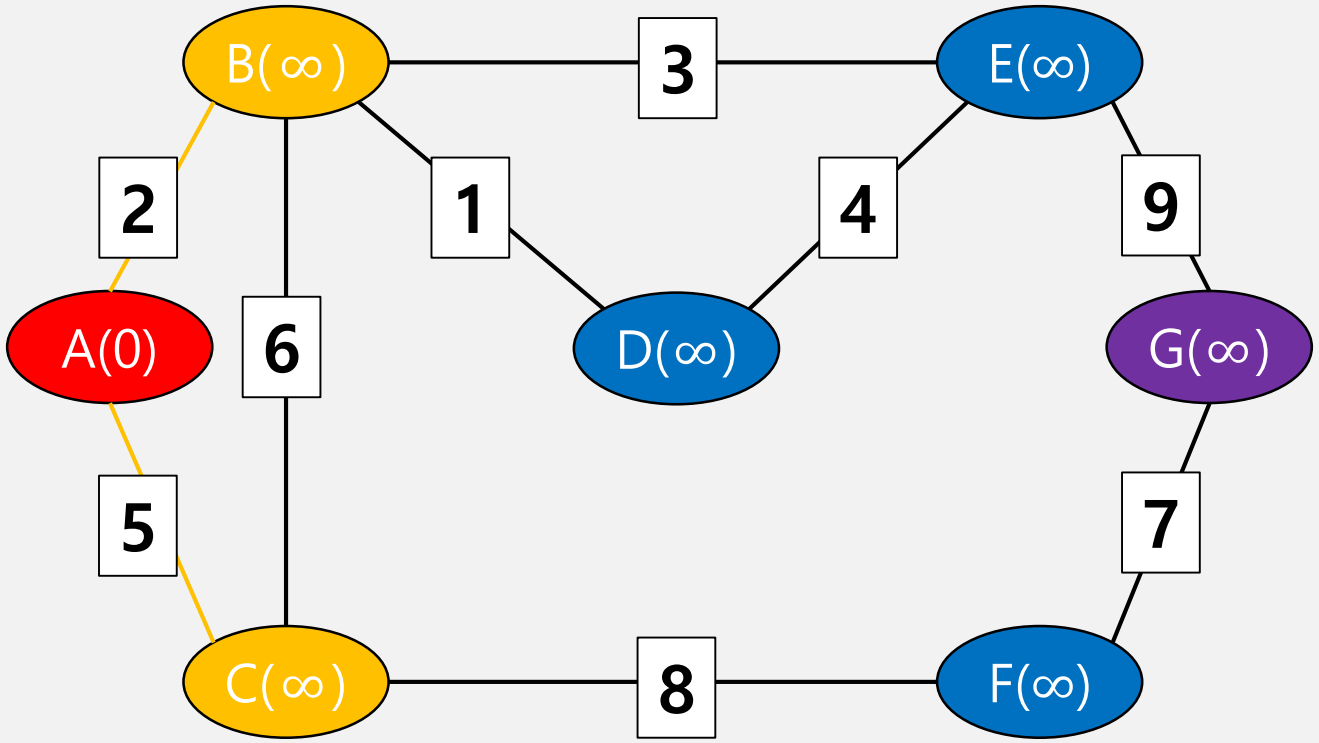


시작점 : A
목표 : G

초기 가중치 설정 :
시작점은 0,
그 외는 무한대(∞)
(도착할 수 있을지 모름)

그래프

다익스트라 알고리즘 (Dijkstra algorithm)

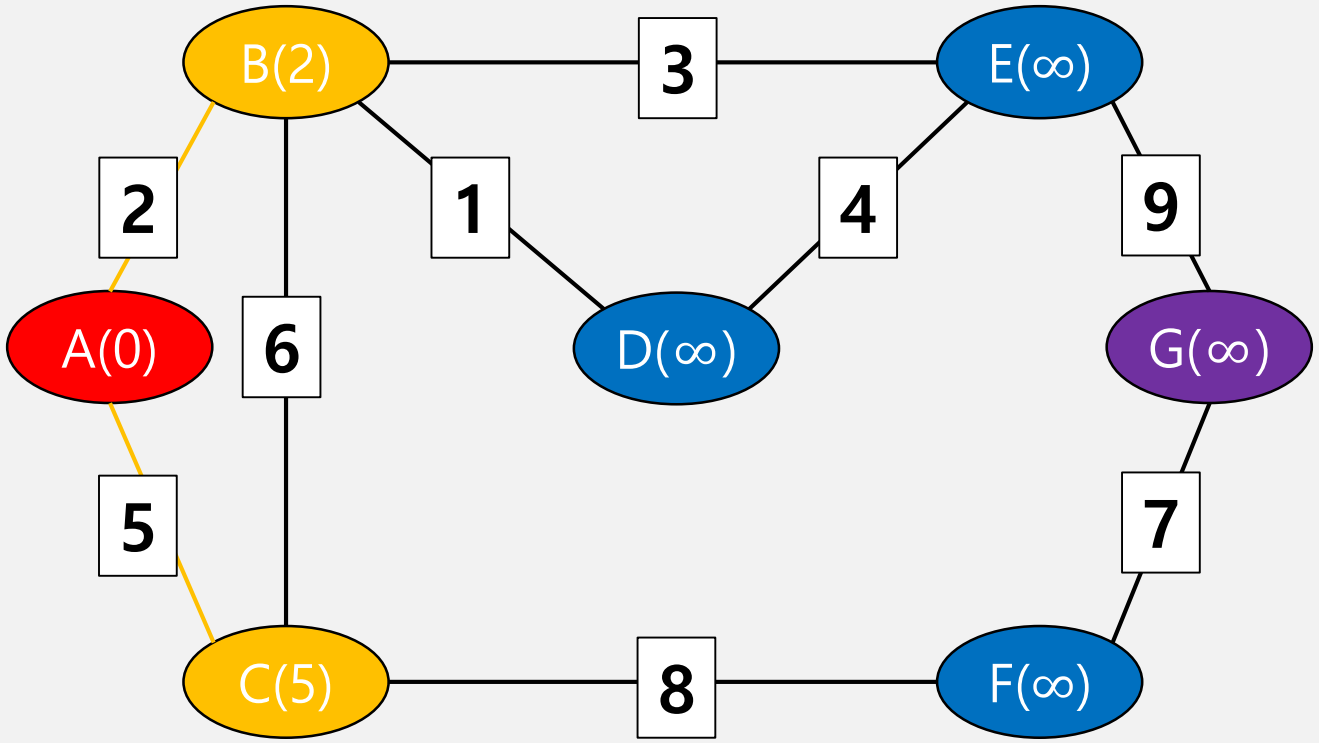


시작점 : A
목표 : G

현재 정점에서 갈 수 있고
탐색하지 않은 정점을
후보로 결정(B, C)

그래프

다익스트라 알고리즘 (Dijkstra algorithm)

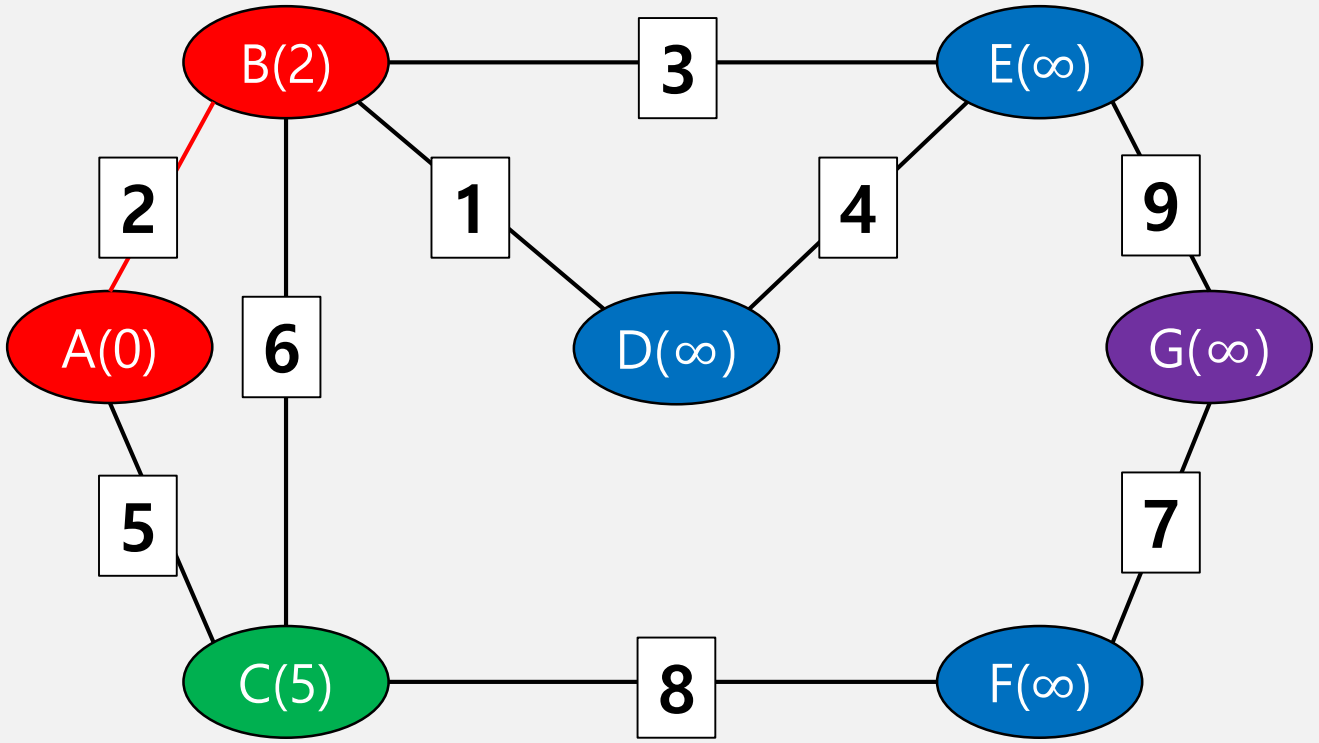


시작점 : A
목표 : G

후보 정점의 가중치를 계산
현재 정점 + 가중치 = 후보정점

그래프

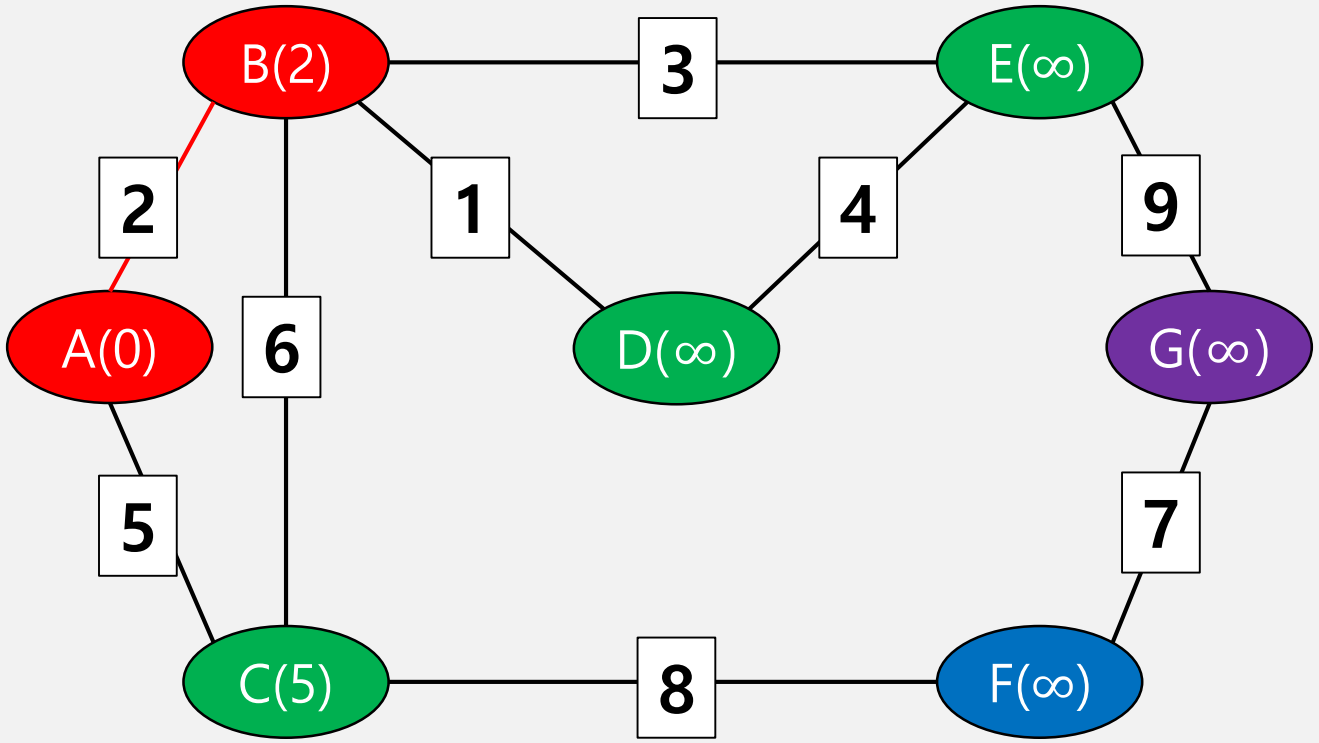
다익스트라 알고리즘 (Dijkstra algorithm)



후보 정점 중
가중치가 가장 작은 정점 선택
⇒ B

그래프

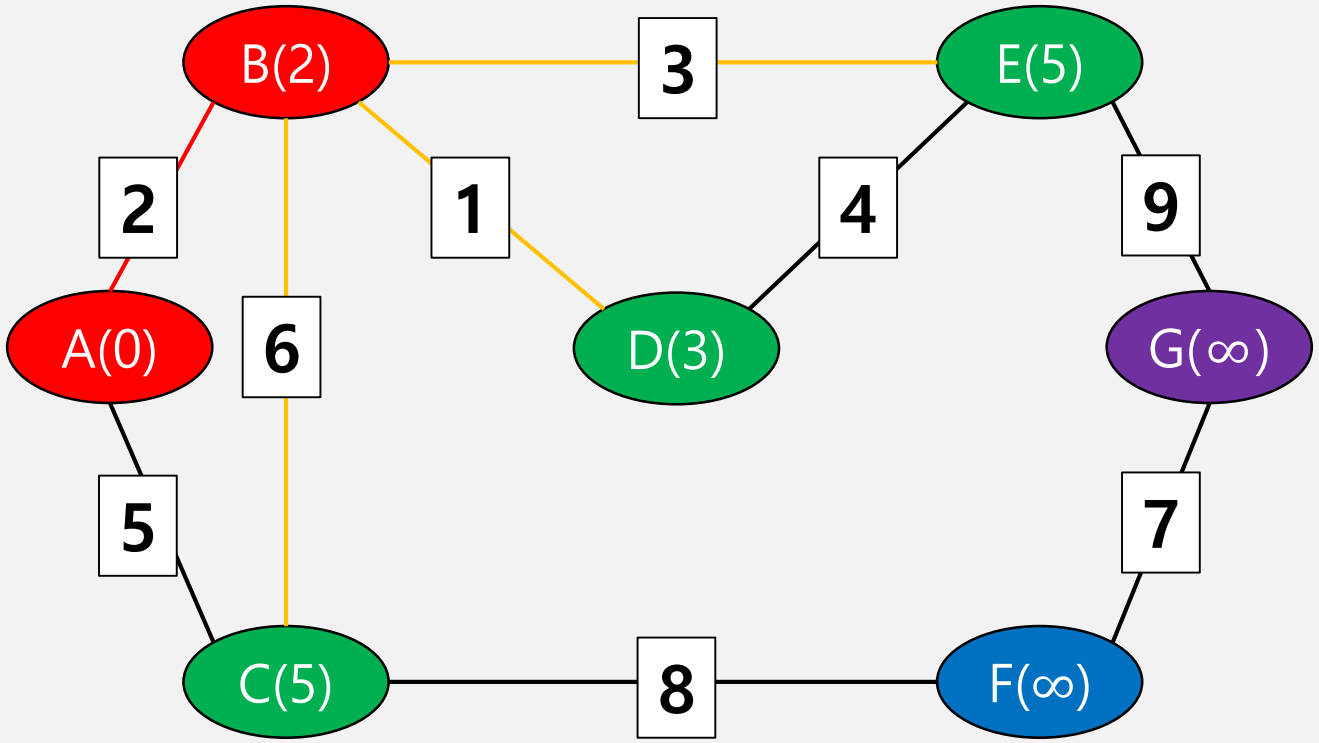
다익스트라 알고리즘 (Dijkstra algorithm)



현재 정점에서 갈 수 있고
탐색하지 않은 정점을 추가하여
후보로 결정(C, D, E)

그래프

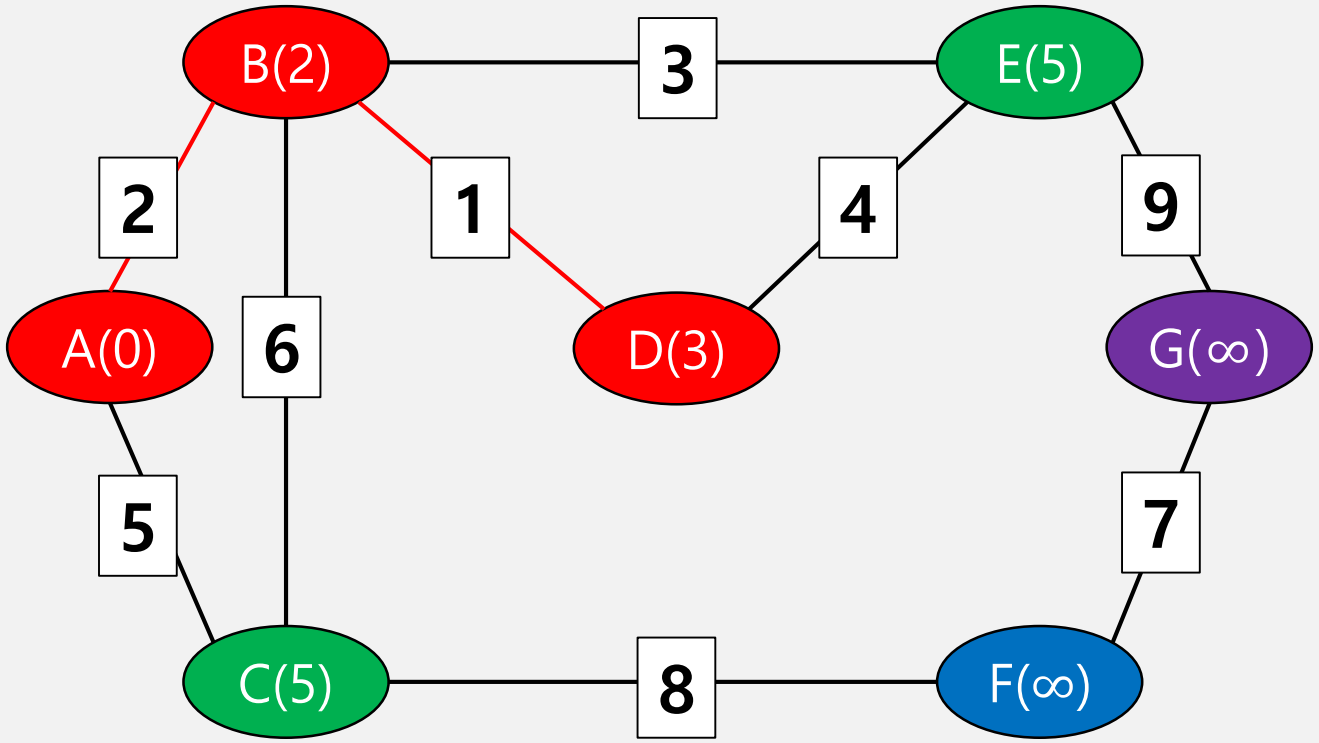
다익스트라 알고리즘 (Dijkstra algorithm)



후보 정점의 가중치를 계산
현재 정점 + 가중치 = 후보정점

그래프

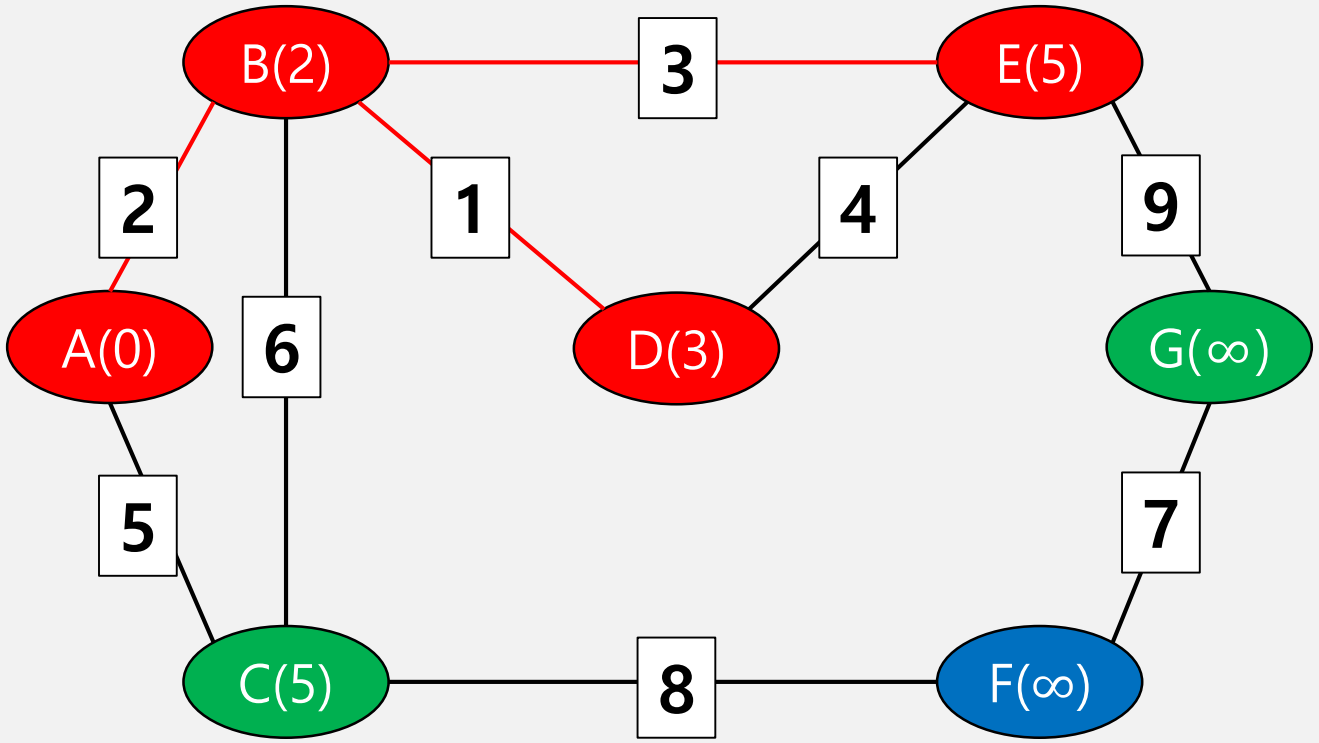
다익스트라 알고리즘 (Dijkstra algorithm)



후보 정점 중
가중치가 가장 작은 정점 선택
⇒ D

그래프

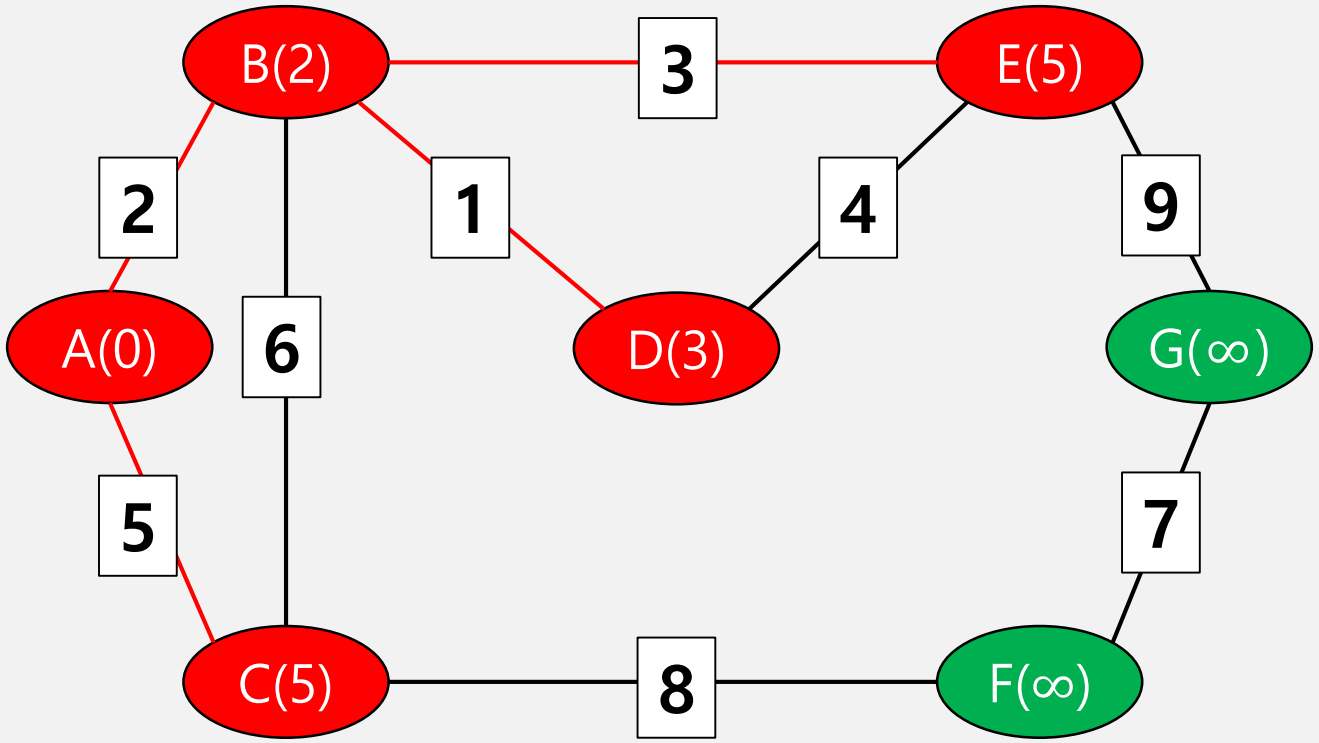
다익스트라 알고리즘 (Dijkstra algorithm)



같은 작업을 반복

그래프

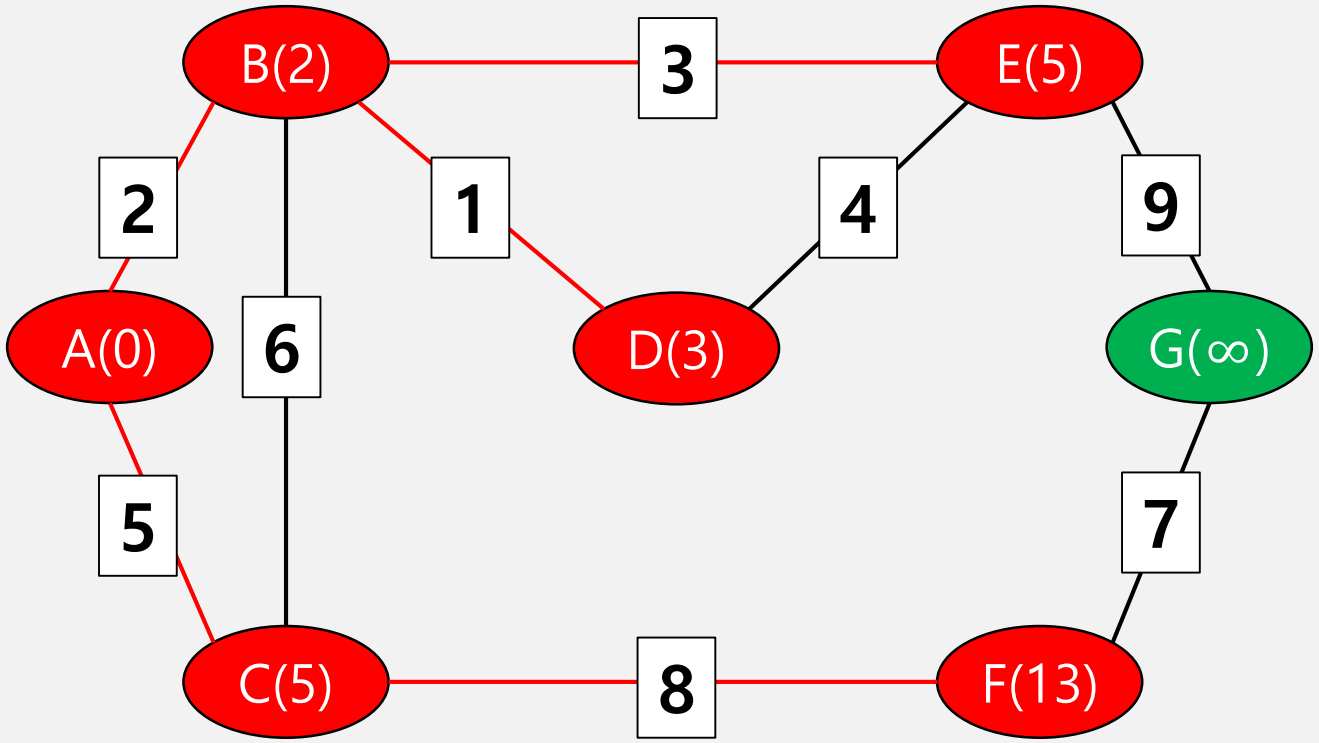
다익스트라 알고리즘 (Dijkstra algorithm)



같은 작업을 반복

그래프

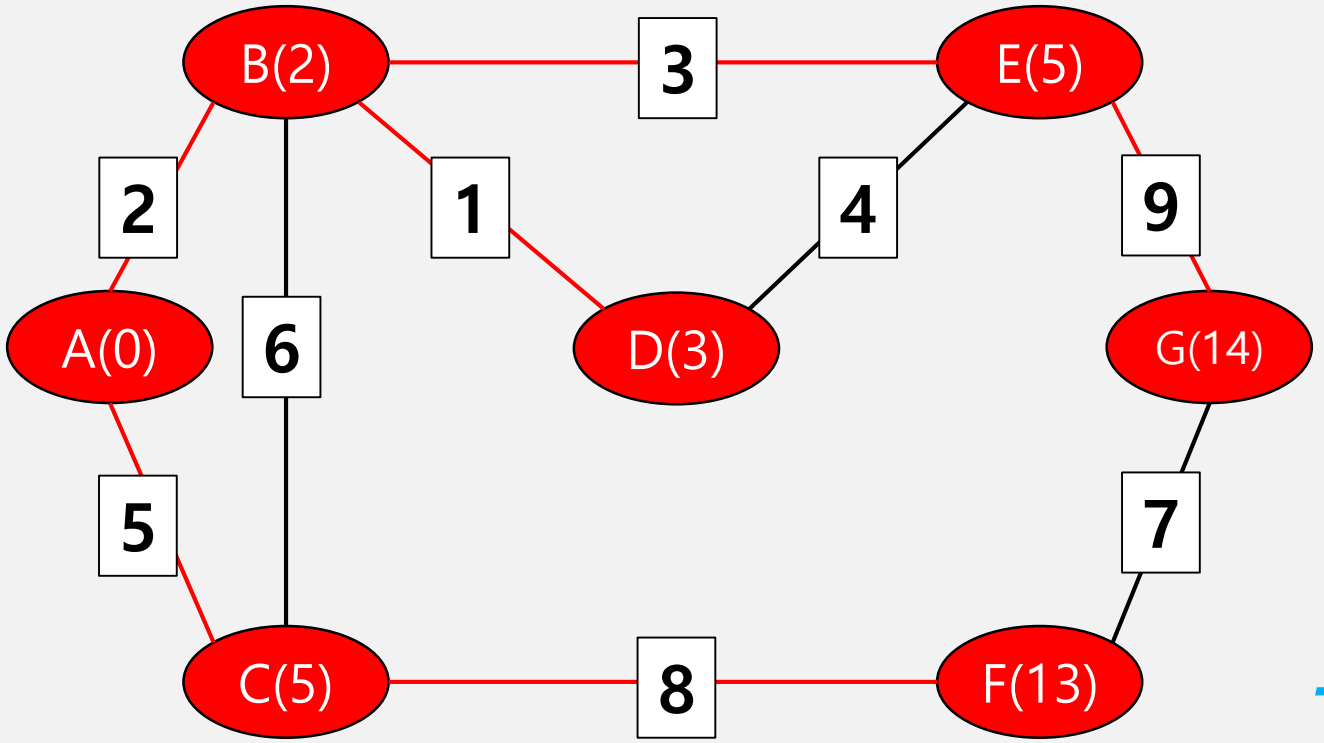
다익스트라 알고리즘 (Dijkstra algorithm)



같은 작업을 반복

그래프

다익스트라 알고리즘 (Dijkstra algorithm)



같은 작업을 반복

최단거리 탐색 완료

그래프

다익스트라 알고리즘 (Dijkstra algorithm)

```
public static void main(String[] args) {
    int[][] nodes =
    {
        {1, 2, 2},
        {1, 3, 5},
        {2, 3, 6},
        {2, 4, 1},
        {2, 5, 3},
        {4, 5, 4},
        {5, 7, 9},
        {3, 6, 8},
        {6, 7, 7}
    };
```

```
    int startNode = 1;
    int N = 7; //node의 수
    int E = 9; //간선의 수

    int INF = 100000000;

    boolean[] visit = new boolean[N];
    int[] nodeCost = new int[N];

    for (int i = 0; i < N; i++) {
        visit[i] = false;
        nodeCost[i] = INF;
    }

    visit[startNode] = true;
    nodeCost[startNode] = 0;
```

그래프

다익스트라 알고리즘 (Dijkstra algorithm)

```
while (true) {
    boolean out = true;
    for (int j = 0; j < visit.length; j++) {
        if(!visit[j]) {
            out = false;
        }
    }
    if(out) {
        break;
    }

    int lowCost = INF;
    int lowNode = -1;
    startNode = -1;
```

```
    //현재 출발지를 도착지로 보고 연결 할 수 있는 출발지를 찾는다
    for (int j = 0; j < nodes.length; j++) {
        int nowNode = nodes[j][0] - 1;
        int arriveNode = nodes[j][1] - 1;
        int cost = nodes[j][2];

        //만약 연결되어있고 출발지가 아직 방문되지 않았다면
        if (visit[nowNode] && !visit[arriveNode] && nodeCost[nowNode] + cost < lowCost) {
            //가중치를 계산한 후 현재상태에서 가장 작은 가중치와 비교
            lowCost = nodeCost[nowNode] + cost;
            lowNode = arriveNode;
            startNode = nowNode;
        }
        else if (visit[arriveNode] && !visit[nowNode] && nodeCost[arriveNode] + cost < lowCost) {
            lowCost = nodeCost[arriveNode] + cost;
            lowNode = nowNode;
            startNode = arriveNode;
        }
    }

    visit[lowNode] = true;
    nodeCost[lowNode] = lowCost;
}
```

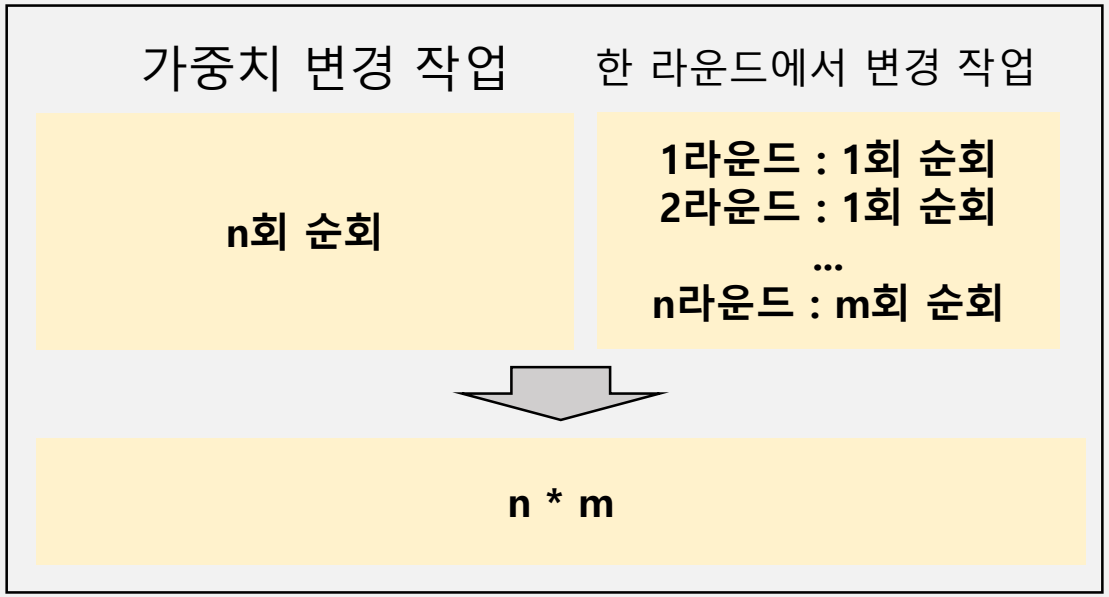
정렬 알고리즘

시간복잡도 – 벨만 포드

- 가중치 계산과 변경을 모든 간선에 대해 반복
- 정점 수를 n, 간선 수를 m이라고 하면 O(nm)
- 음의 가중치가 존재할 경우에도 최단거리를 찾을 수 있음

시간복잡도 – 다익스트라

- 정점 선택 방법을 고민해서 효율이 좋은 최단 경로를 구함
- 정점 수를 n, 간선 수를 m
- 최악의 시간 복잡도 : O(n^2) – 정점 선택이 좋지 않았을 경우
- 평균 시간 복잡도 : O(m + $n\log_2 n$) – 데이터를 고민하여 정점을 선택하였을 경우
- 음의 가중치가 존재할 경우에는 최단거리를 찾을 수 없음



연습 문제

아래 지시를 읽고 탑 오르기의 결과를 출력하세요.

지시사항

탑을 오르려고 한다. 탑에는 손잡이가 있는 층이 존재한다.

탑을 오를 때는 손잡이가 있는 층에만 머무를 수 있으며

- 이전 층에서 올라온 층 수
- 혹은 이전 층에서 올라온 층 수+1,
- 혹은 이전 층에서 올라온 층 수 -1 만큼만 점프할 수 있다.

탑에서 손잡이가 있는 층이 저장되어 있는 배열이 주어졌을 때,

마지막 층에 도달할 수 있으면 TRUE 없으면 FALSE를 출력하라.

- 첫 층은 무조건 0층부터 시작하며 다음으로 가는 다음 층은 무조건 1층이다.
- 탑은 위로만 올라갈 수 있다.
- 탑의 손잡이가 있어도 머무르지 않아도 된다면 머무르지 않을 수 있다.

(1) 탑의 배열 = (0,1,2,4,7,9)

0층에서 1층으로 간다.

1층에서 1만큼 오른 2층으로 간다.

2층에서 2만큼 점프한 4층으로 간다.

4층에서 3만큼 점프한 7층으로 간다.

7층에서 2만큼 점프한 9층으로 간다.

정답 : TRUE

(2) 탑의 배열 = (0,1,3,5,9)

0층에서 1층으로 간다.

1층에서 2만큼 오른 3층으로 간다.

3층에서 2만큼 오른 5층으로 간다.

5층에서는 갈 방법이 없다.

정답 : FALSE

스택과 큐

추천 문제

DFS/BFS

1. https://programmers.co.kr/learn/courses/30/lessons/12980 (점프와 순간 이동)
2. https://programmers.co.kr/learn/courses/30/lessons/43162 (네트워크)
3. https://programmers.co.kr/learn/courses/30/lessons/1844 (게임 맵 최단거리)
4. https://programmers.co.kr/learn/courses/30/lessons/43165 (타겟 넘버)

수고하셨습니다.