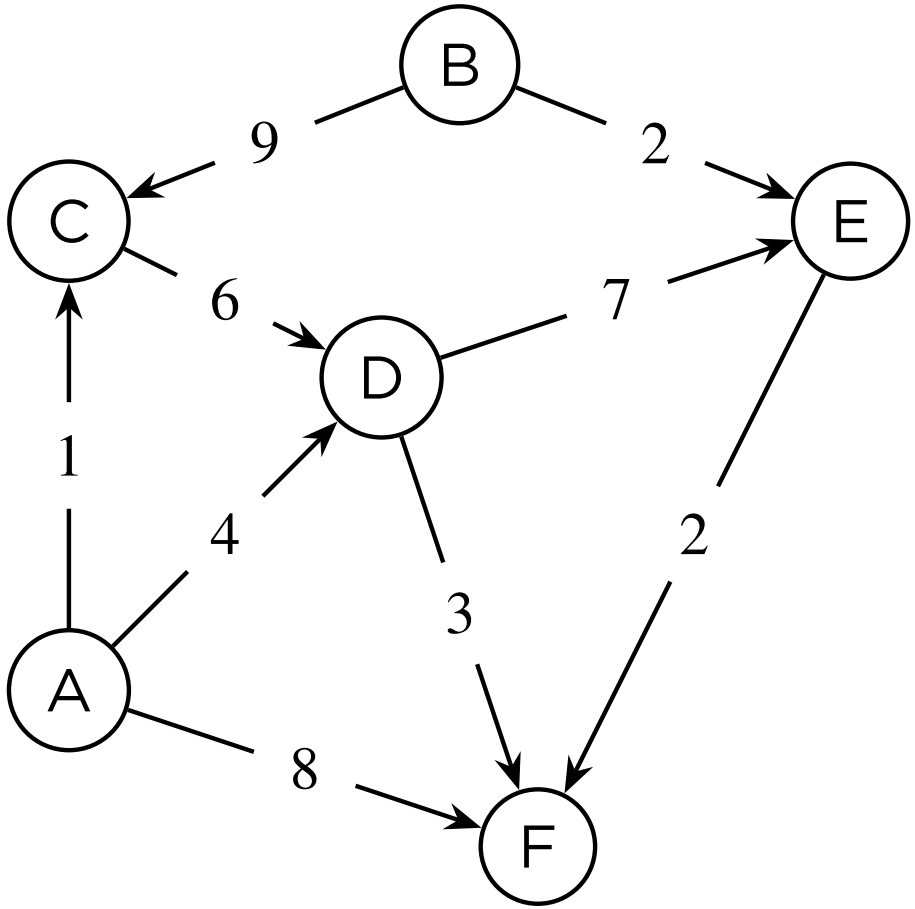
#2 그래프의 저장과 탐색

2019 SCSC Summer Coding Workshop

서강대학교 컴퓨터공학과 박수현

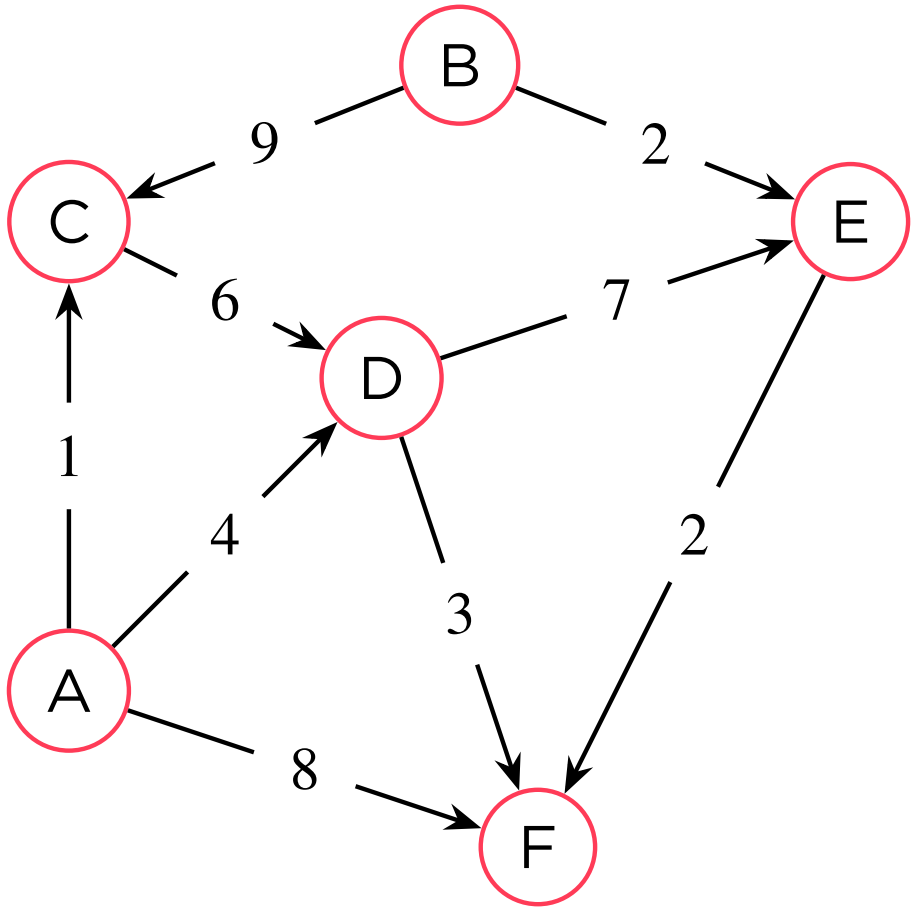
me@shiftpsh.com

그래프



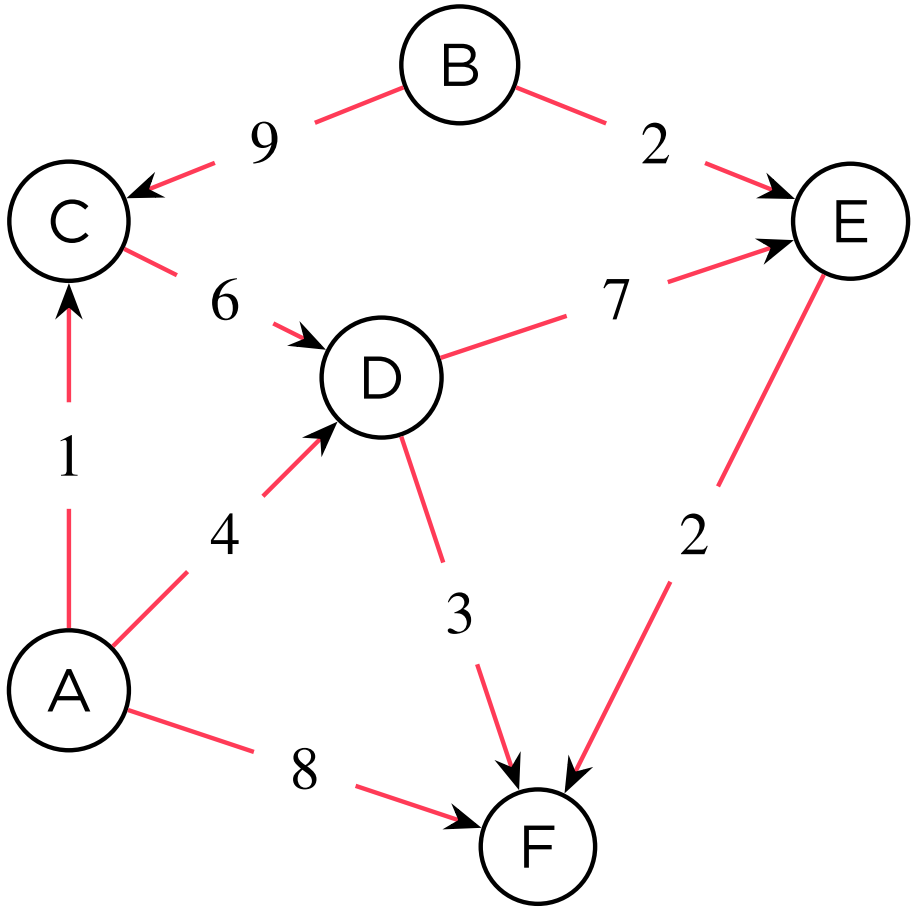
$G = (V, E)$

그래프



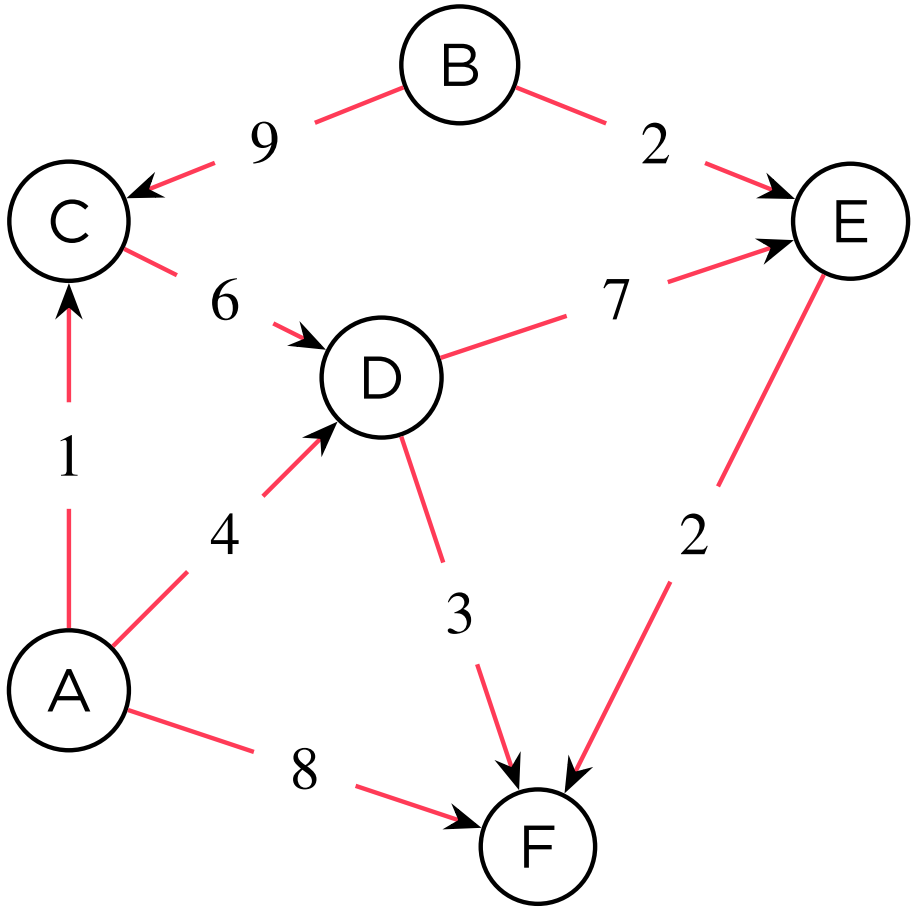
정점, 또는 **노드**node, **버텍스**vertex

그래프



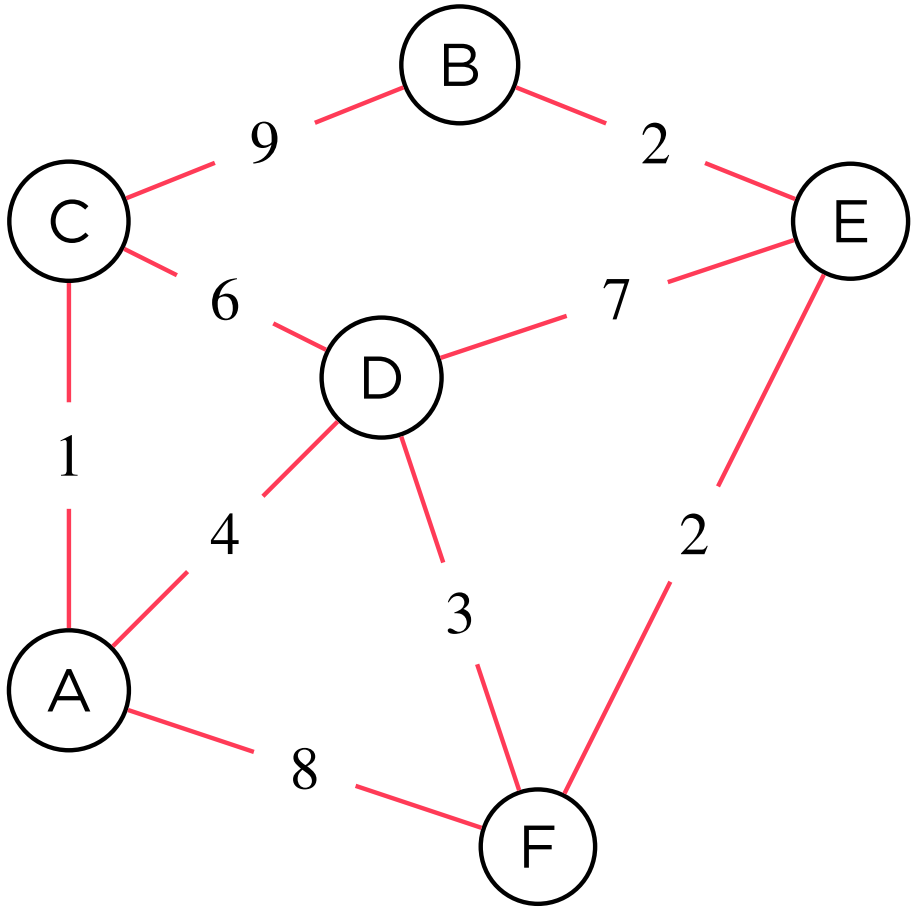
간선, 또는 **엣지**edge

그래프



단방향, 또는 **유향** 그래프

그래프



양방향, 또는 **무향** 그래프

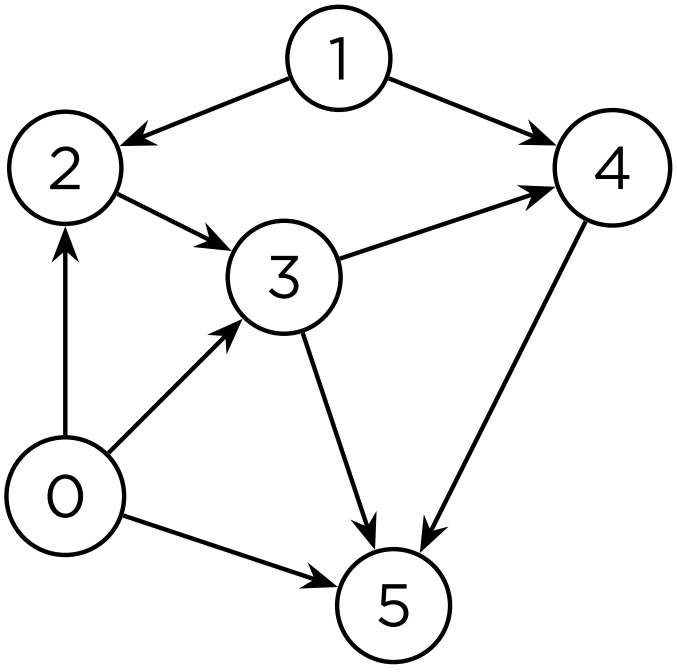
그래프

활용 사례

- 노드를 방으로, 간선을 통로로 $\rightarrow$ 미로 찾기
- 노드를 지역으로, 간선을 도로로 $\rightarrow$ 경로를 지나가는 데 걸리는 시간 구하기
- 노드를 사람으로, 간선을 친구 관계로 $\rightarrow$ 함께 아는 친구가 많은 사람 구하기

등 무궁무진...

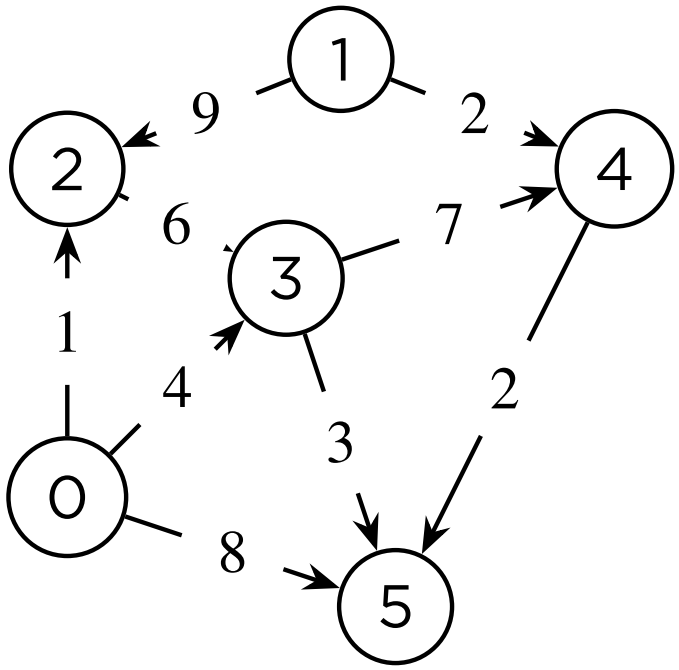
그래프의 저장 – 인접 행렬



$$A = \begin{bmatrix} 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} \\ 0 & 0 & \mathbf{1} & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

A_{ij}: $i \to j$의 간선 존재 여부

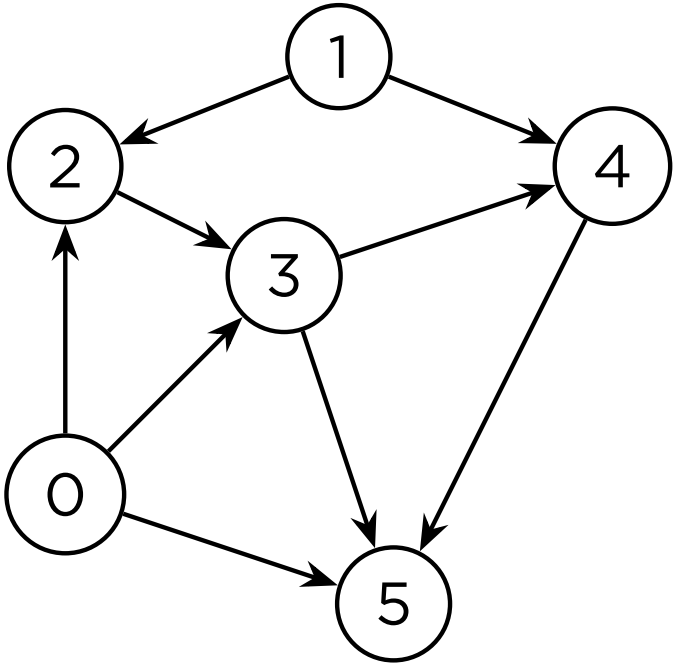
그래프의 저장 – 인접 행렬



$$A = \begin{bmatrix} \infty & \infty & \mathbf{1} & \mathbf{4} & \infty & \mathbf{8} \\ \infty & \infty & \mathbf{9} & \infty & \mathbf{4} & \infty \\ \infty & \infty & \infty & \mathbf{6} & \infty & \infty \\ \infty & \infty & \infty & \infty & \mathbf{7} & \mathbf{3} \\ \infty & \infty & \infty & \infty & \infty & \mathbf{2} \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

A_{ij}: $i \rightarrow j$의 간선의 가중치

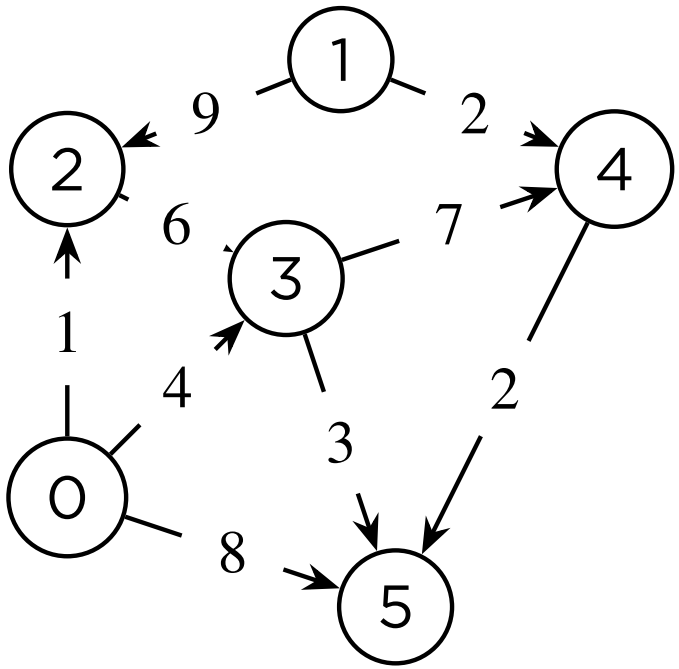
그래프의 저장 – 인접 리스트



$$A = \begin{bmatrix} \{2, 3, 5\} \\ \{2, 4\} \\ \{3\} \\ \{4, 5\} \\ \{5\} \\ \emptyset \end{bmatrix}$$

A_i: i에 인접한 노드의 집합

그래프의 저장 – 인접 리스트



$$A = \begin{bmatrix} \{\langle \mathbf{2}, 1\rangle, \langle \mathbf{3}, 4\rangle, \langle \mathbf{5}, 8\rangle\} \\ \{\langle \mathbf{2}, 9\rangle, \langle \mathbf{4}, 2\rangle\} \\ \{\langle \mathbf{3}, 6\rangle\} \\ \{\langle \mathbf{4}, 7\rangle, \langle \mathbf{5}, 3\rangle\} \\ \{\langle \mathbf{5}, 2\rangle\} \\ \emptyset \end{bmatrix}$$

A_i: i에 인접한 노드의 집합 ($\langle$노드, 거리$\rangle$의 쌍)

그래프의 저장

인접 행렬의 경우

- **2차원** (정적) **배열**을 이용해 저장
- 예를 들어 `int[][]`

인접 리스트의 경우

- **동적 배열들의 배열**을 이용해 저장
- 예를 들어 `vector<pair<int, int>>[]`

물론 특수한 그래프의 경우 다른 방식으로도 저장할 수 있음! (후술)

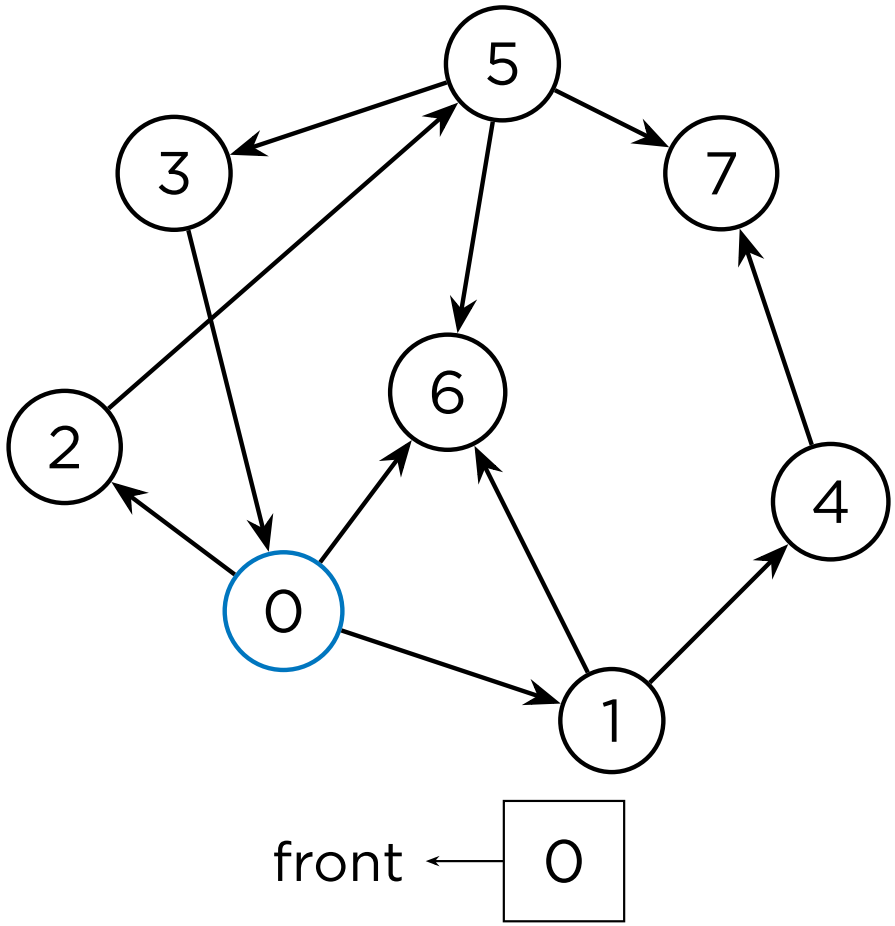
그래프의 탐색

- 너비 우선 탐색Breadth First Search
- 깊이 우선 탐색Depth First Search

그래프의 탐색 - BFS

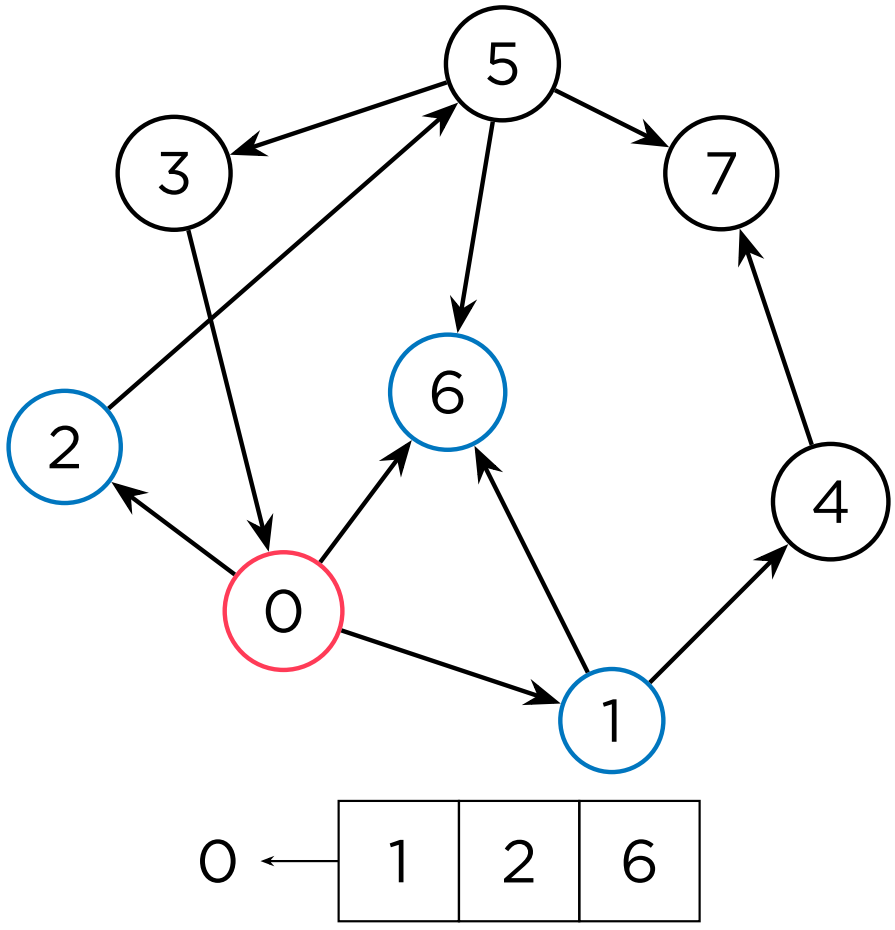
- 현재 정점에 **인접한 정점들을 우선적으로** 모두 방문하는 탐색 기법
- **큐**queue 를 사용

그래프의 탐색 – BFS



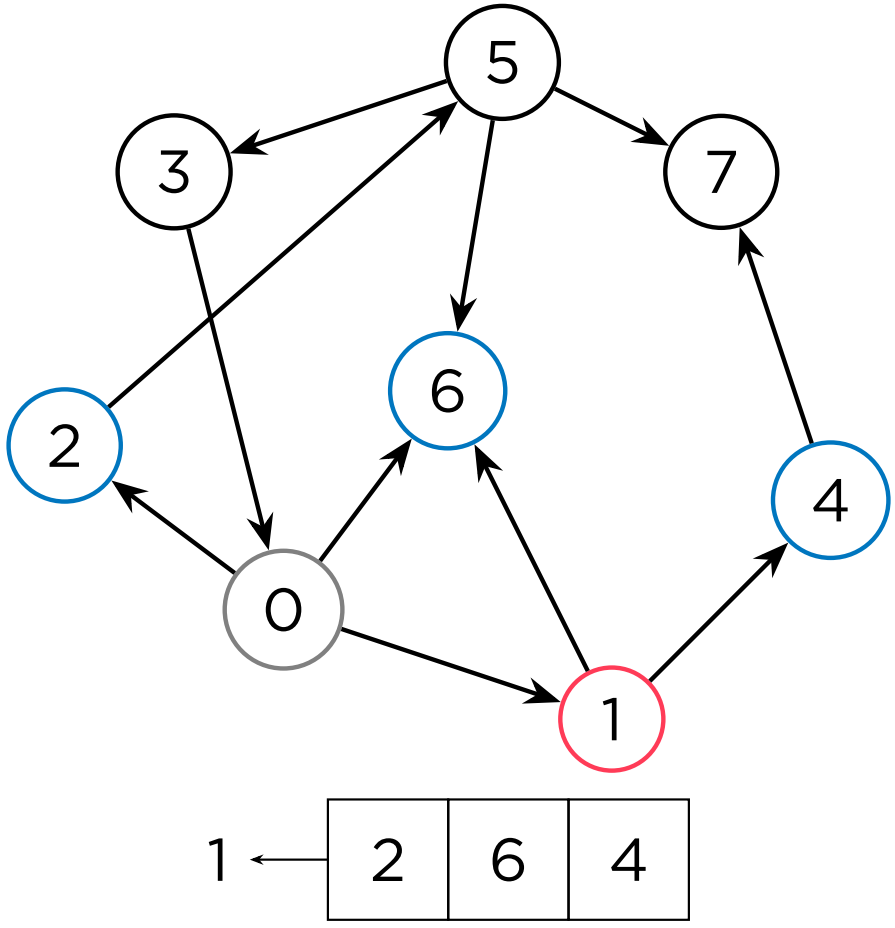
초기 상태

그래프의 탐색 - BFS



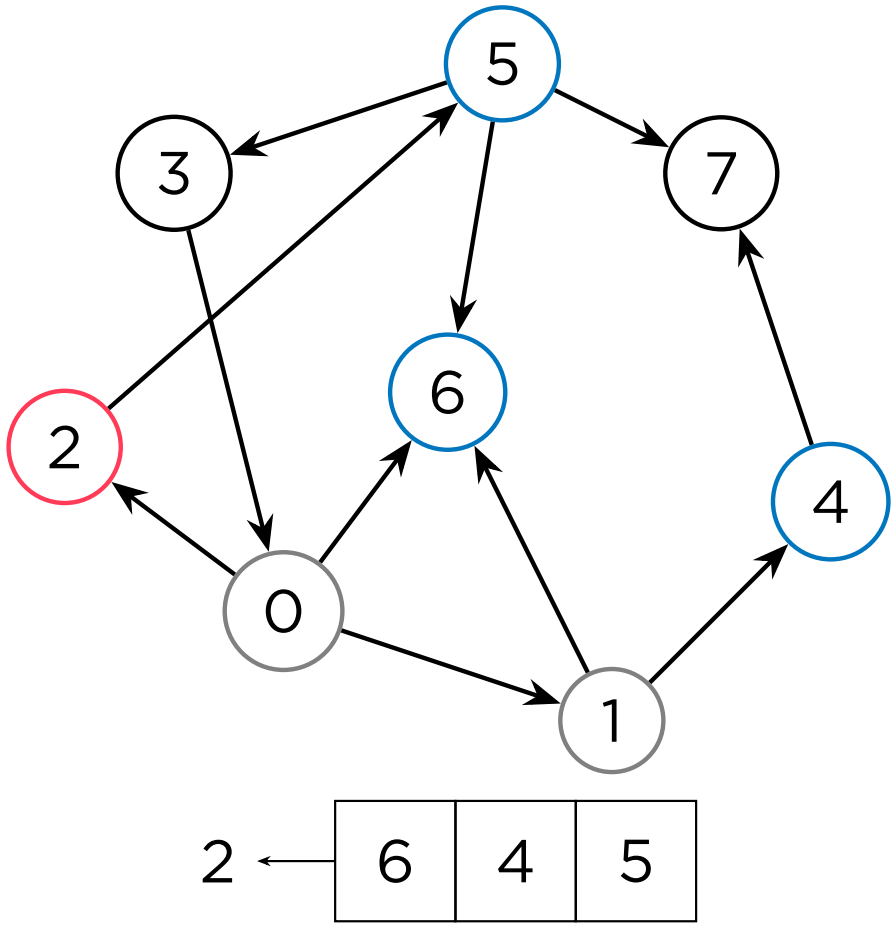
0번 노드를 큐에서 꺼내고 인접한 노드들을 큐에 넣음

그래프의 탐색 - BFS



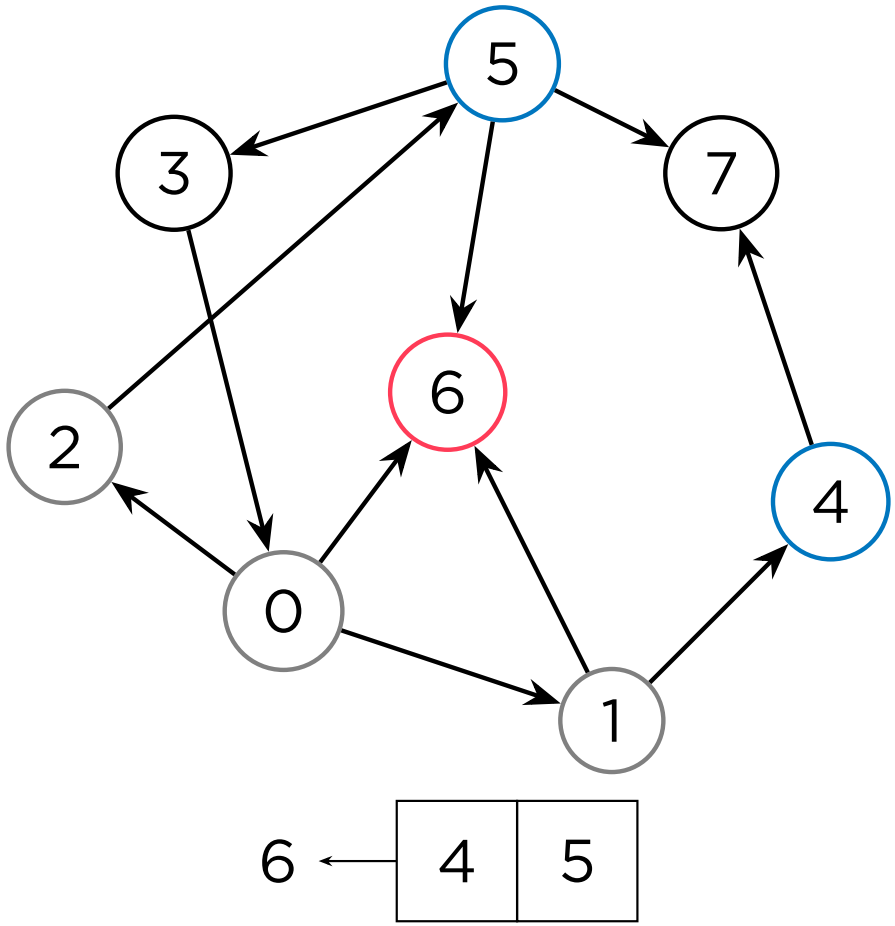
1번 노드를 큐에서 꺼내고 인접한 노드들을 큐에 넣음 (6번 노드는 이미 큐에 들어가 있으므로 다시 넣지 않는다)

그래프의 탐색 - BFS



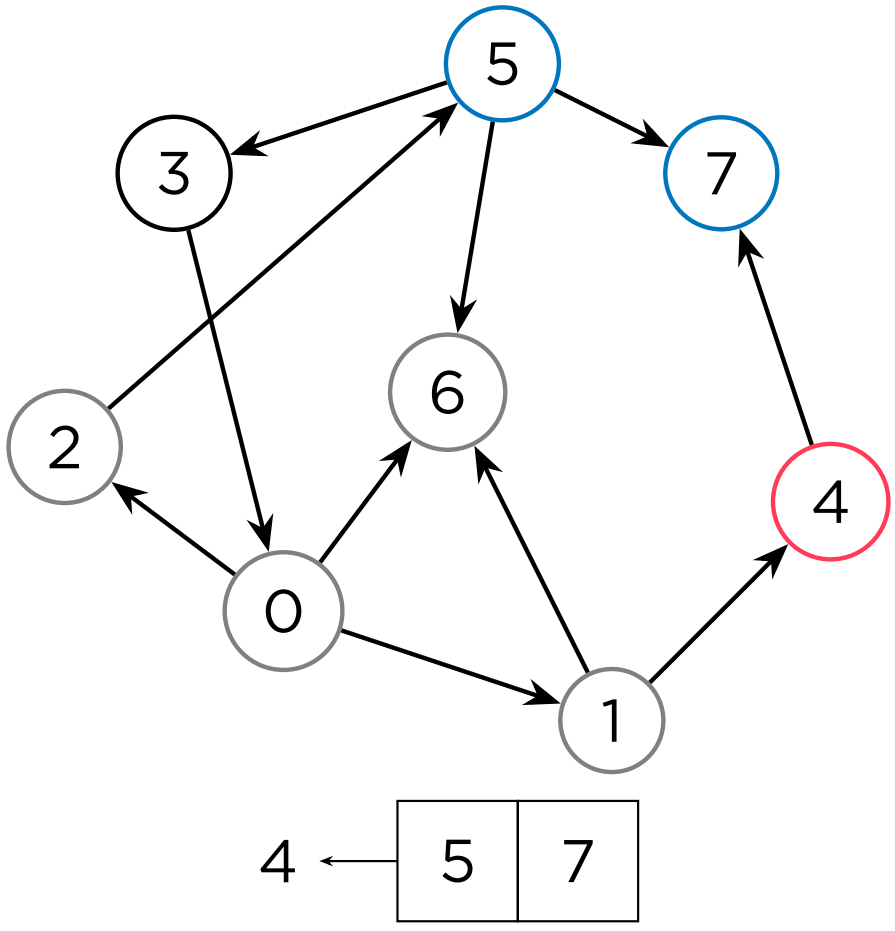
2번 노드를 큐에서 꺼내고 인접한 노드들을 큐에 넣음

그래프의 탐색 - BFS



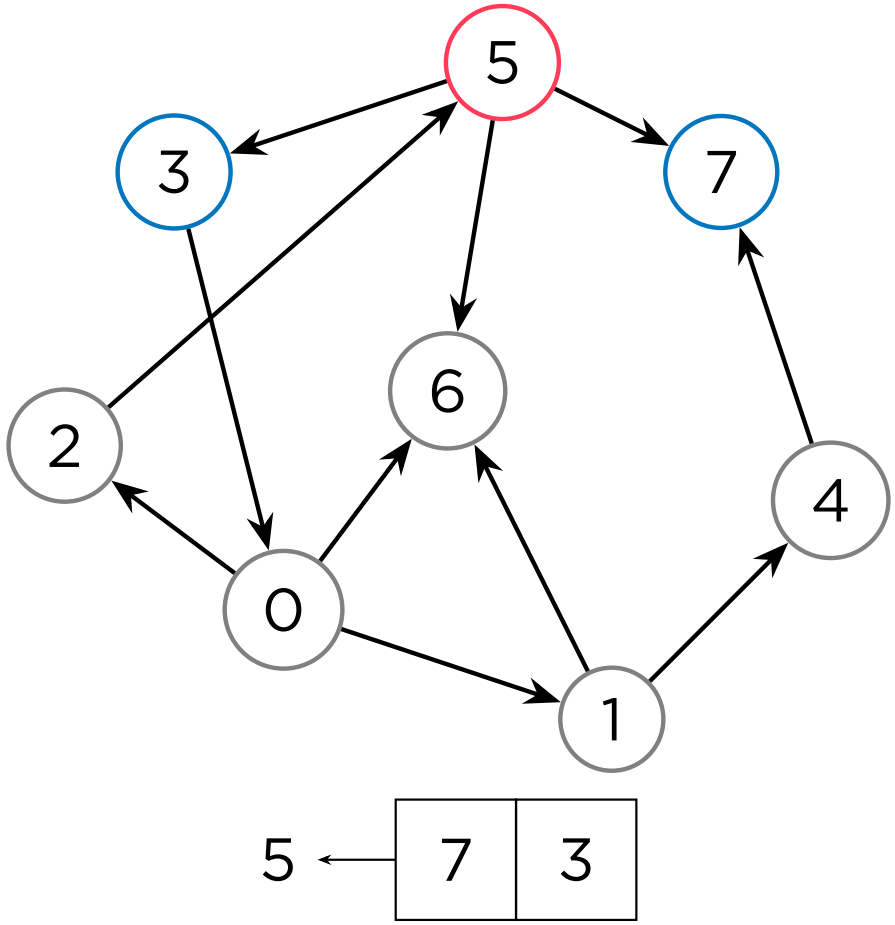
6번 노드엔 인접한 정점들이 없다

그래프의 탐색 - BFS



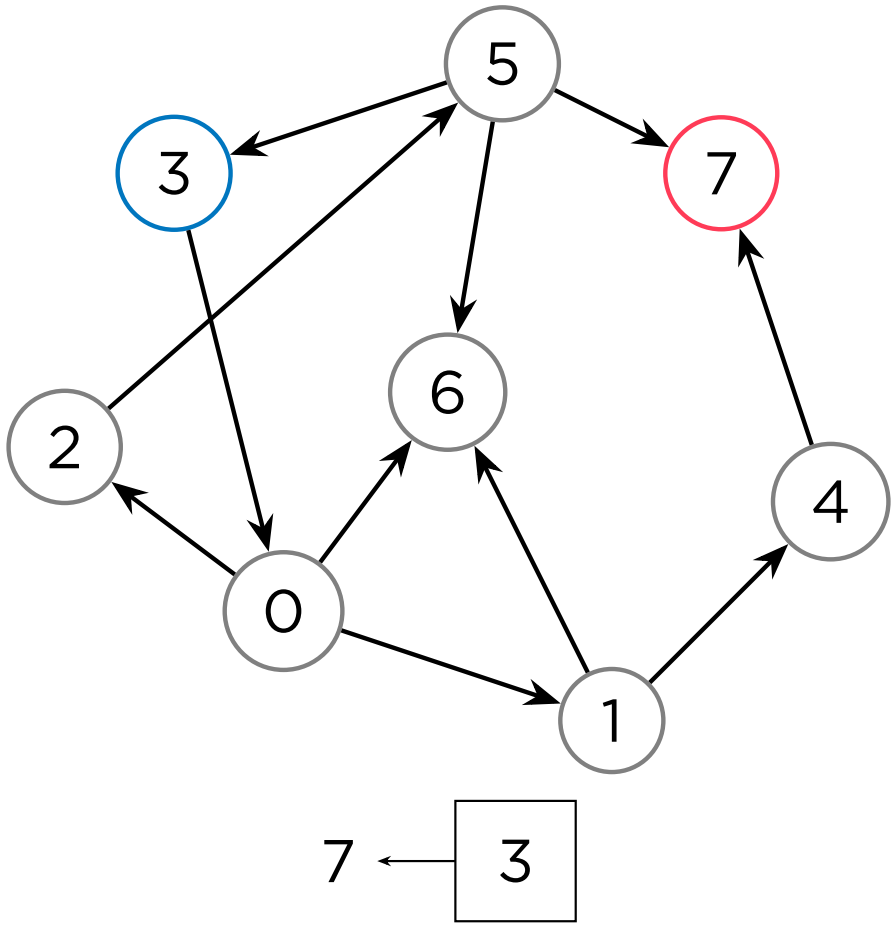
4번 노드를 큐에서 꺼내고 인접한 노드들을 큐에 넣음

그래프의 탐색 - BFS



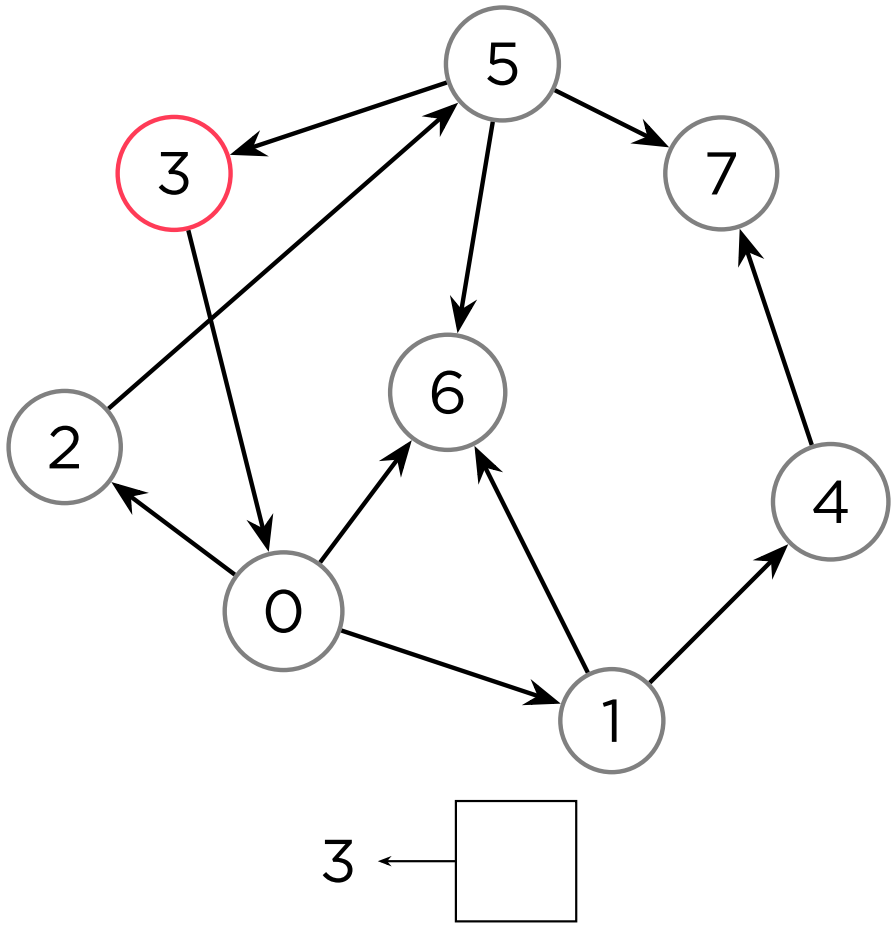
5번 노드를 큐에서 꺼내고 인접한 노드들을 큐에 넣음 (7번 노드는 이미 큐에 들어가 있고, 6번 노드는 이미 방문했으므로 다시 넣지 않는다)

그래프의 탐색 - BFS



7번 노드엔 인접한 정점들이 없다

그래프의 탐색 – BFS



3번 노드를 큐에서 꺼내는데, 0번 노드는 이미 방문했으므로 다시 넣지 않는다

그래프의 탐색 - BFS

- 큐를 하나 준비한다
- `visit` 배열을 하나 만들어서 노드가 큐에 들어간 적이 있는지의 여부를 관리한다

그래프의 탐색 - BFS

```
1   const int N;
2   vector<int> graph[N];
3   bool visit[N];
4
5   queue<int> q;
6   visit[0] = true;
7   q.emplace(0);
8
9   while (!q.empty()) {
10      int u = q.front();
11      q.pop();
12      cout << u << ' ';
13      for (int v : graph[u]) {
14          if (visit[v]) continue;
15          visit[v] = true;
16          q.emplace(v);
17      }
18  }
```

예제 코드

그래프의 탐색 - BFS

```
1  const int N;
2  vector<int> graph[N];
3  bool visit[N];
4
5  queue<int> q;
6  visit[0] = true;
7  q.emplace(0);
8
9  while (!q.empty()) {
10     int u = q.front();
11     q.pop();
12     cout << u << ' ';
13     for (int v : graph[u]) {
14         if (visit[v]) continue;
15         visit[v] = true;
16         q.emplace(v);
17     }
18 }
```

초기 설정. 0번 노드부터 방문함을 전제로 한다

그래프의 탐색 - BFS

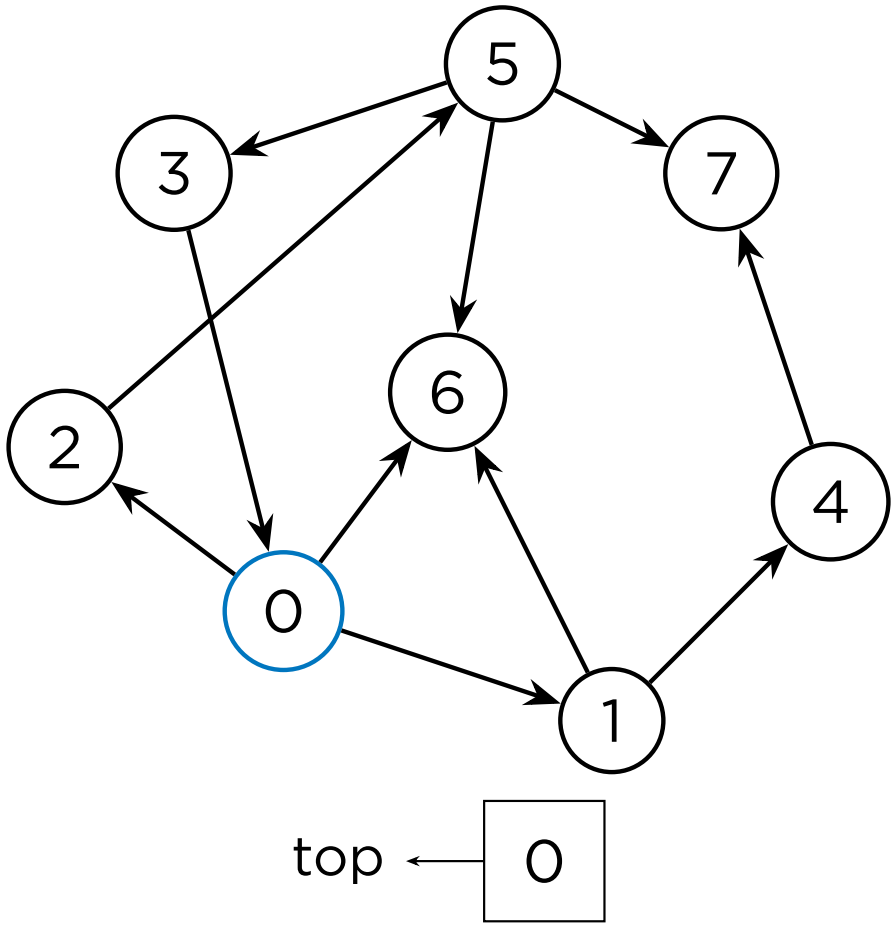
```cpp
const int N;
vector<int> graph[N];
bool visit[N];

queue<int> q;
visit[0] = true;
q.emplace(0);

while (!q.empty()) {
    int u = q.front();
    q.pop();
    cout << u << ' ';
    for (int v : graph[u]) {
        if (visit[v]) continue;
        visit[v] = true;
        q.emplace(v);
    }
}
```

큐 맨 앞의 노드를 u라 하면, 모든 인접한 노드 v에 대해 v를 미방문한 경우 큐에 넣어준다

그래프의 탐색 - DFS

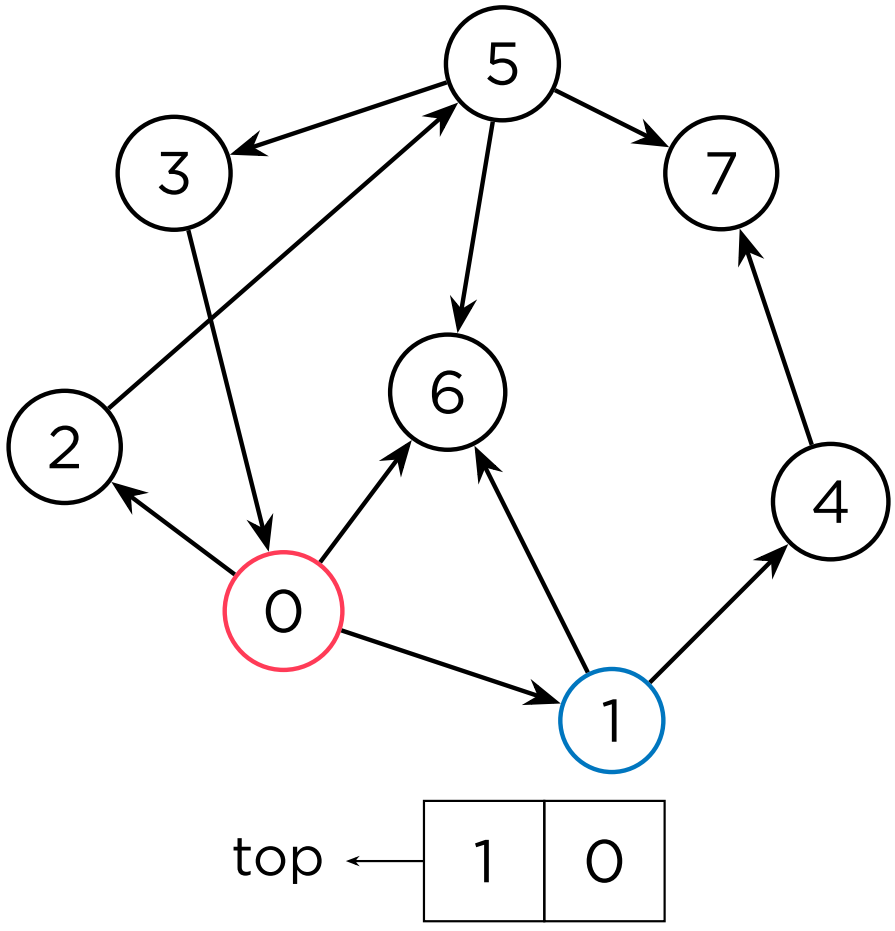
- 현재 정점에서 **한 방향으로 끝까지 진행한 뒤 다른 방향으로 진행하면서** 모두 방문하는 탐색 기법
- **스택**stack 또는 재귀 함수를 사용

그래프의 탐색 – DFS



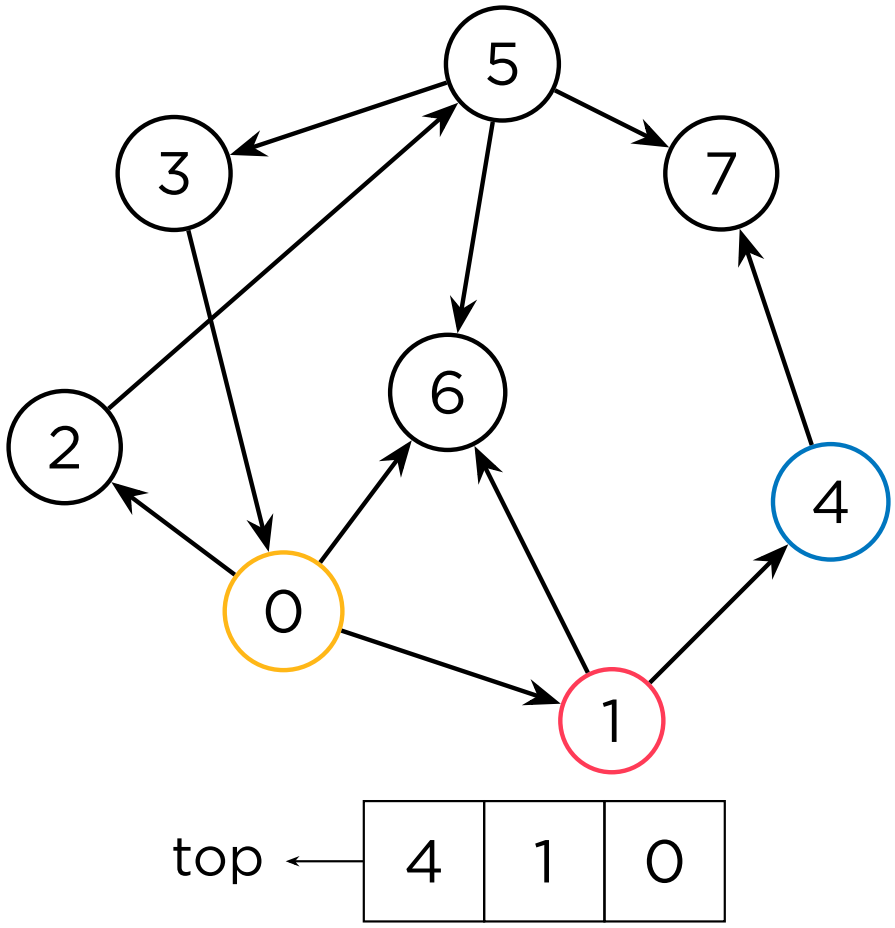
초기 상태

그래프의 탐색 - DFS



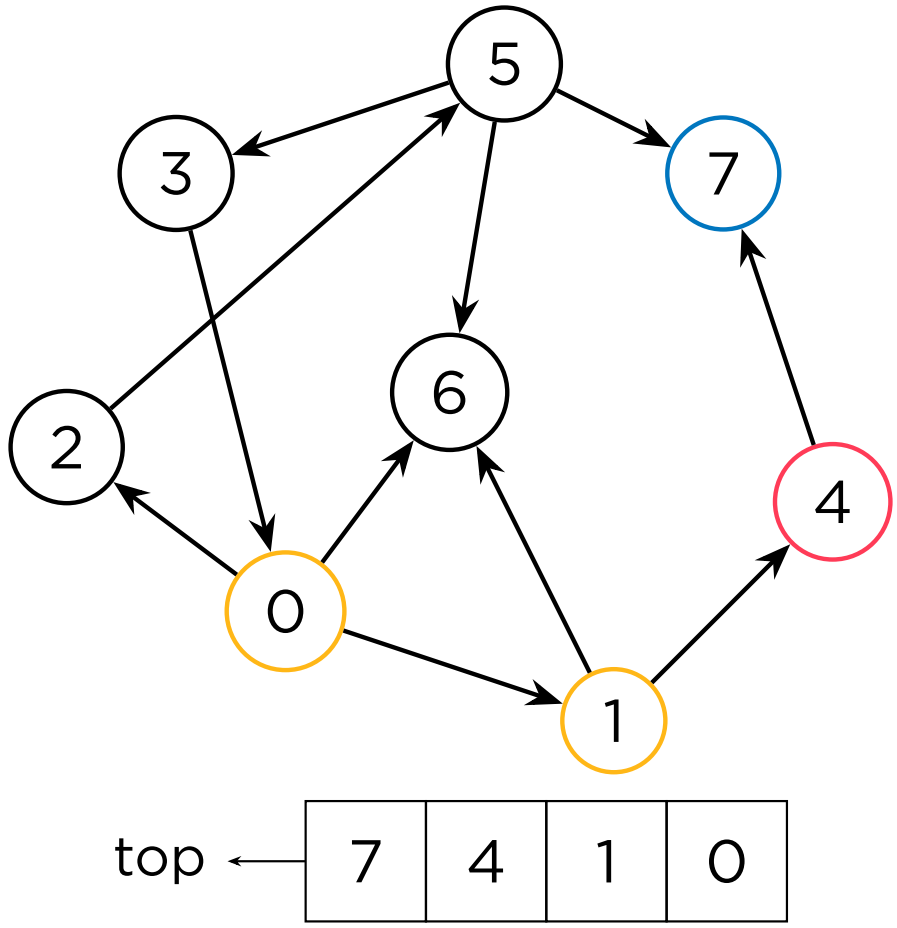
0번 노드에 인접한 1번 노드는 아직 미방문이므로 스택에 넣음

그래프의 탐색 - DFS



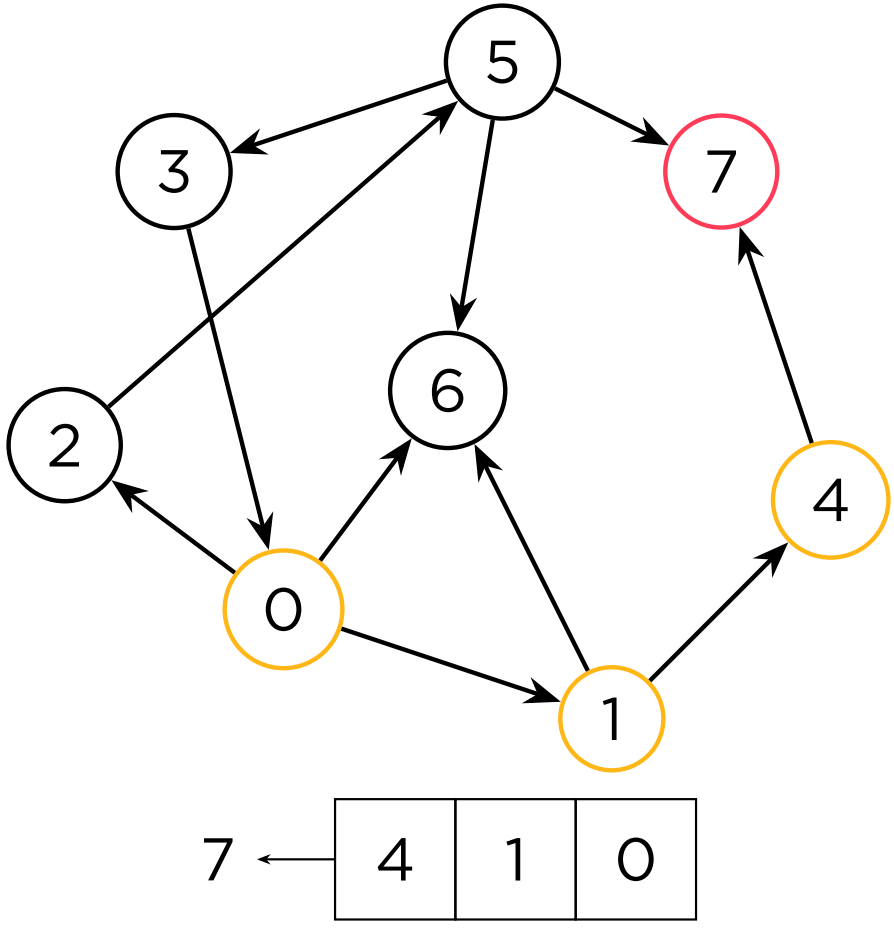
스택 맨 위의 1번 노드에 인접한 4번 노드는 아직 미방문이므로 스택에 넣음

그래프의 탐색 - DFS



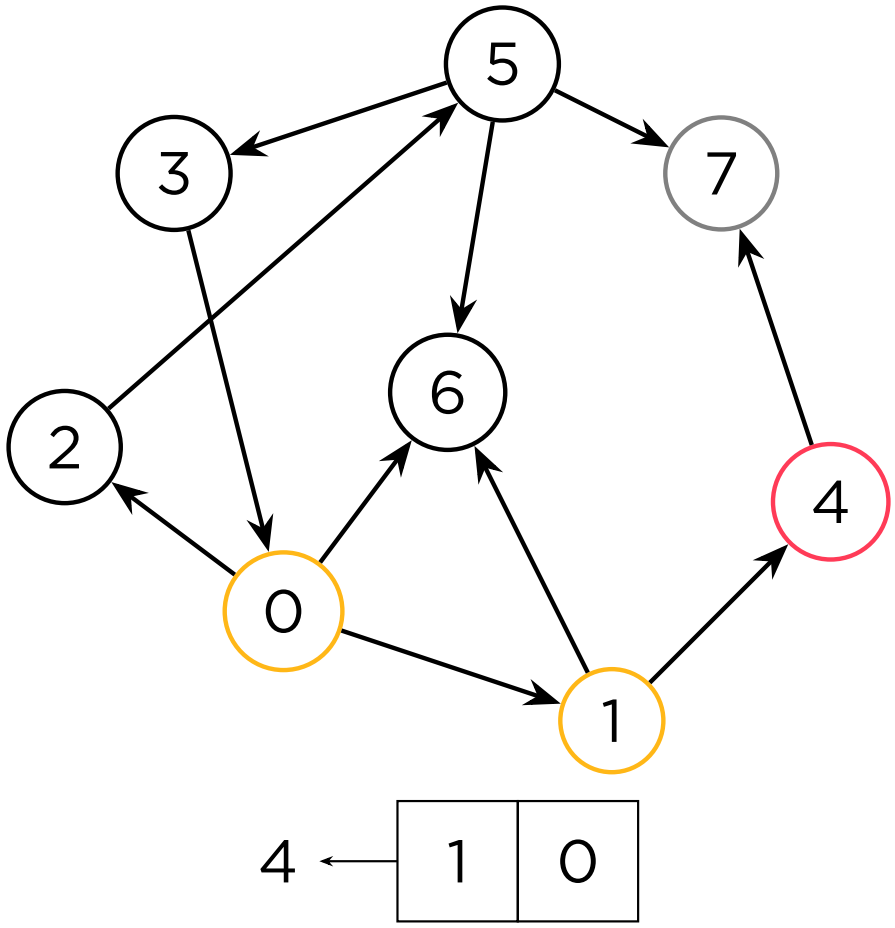
스택 맨 위의 4번 노드에 인접한 7번 노드는 아직 미방문이므로 스택에 넣음

그래프의 탐색 - DFS



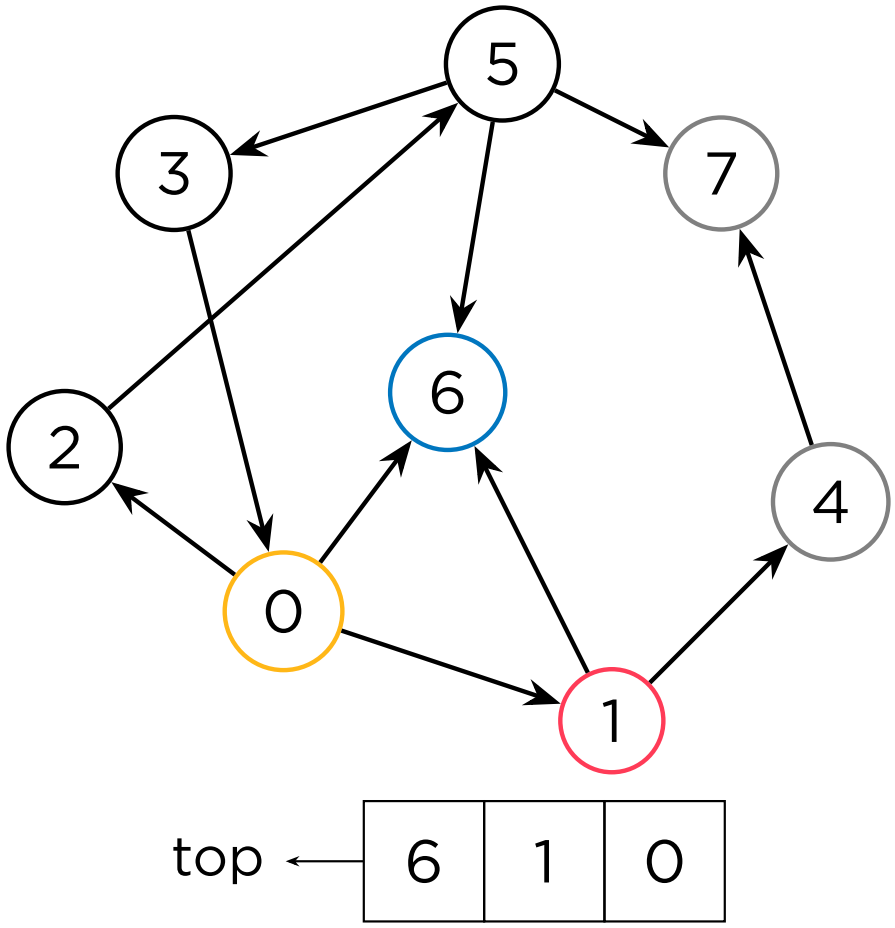
스택 맨 위의 7번 노드에는 인접한 정점이 없으므로, 스택에서 제거

그래프의 탐색 - DFS



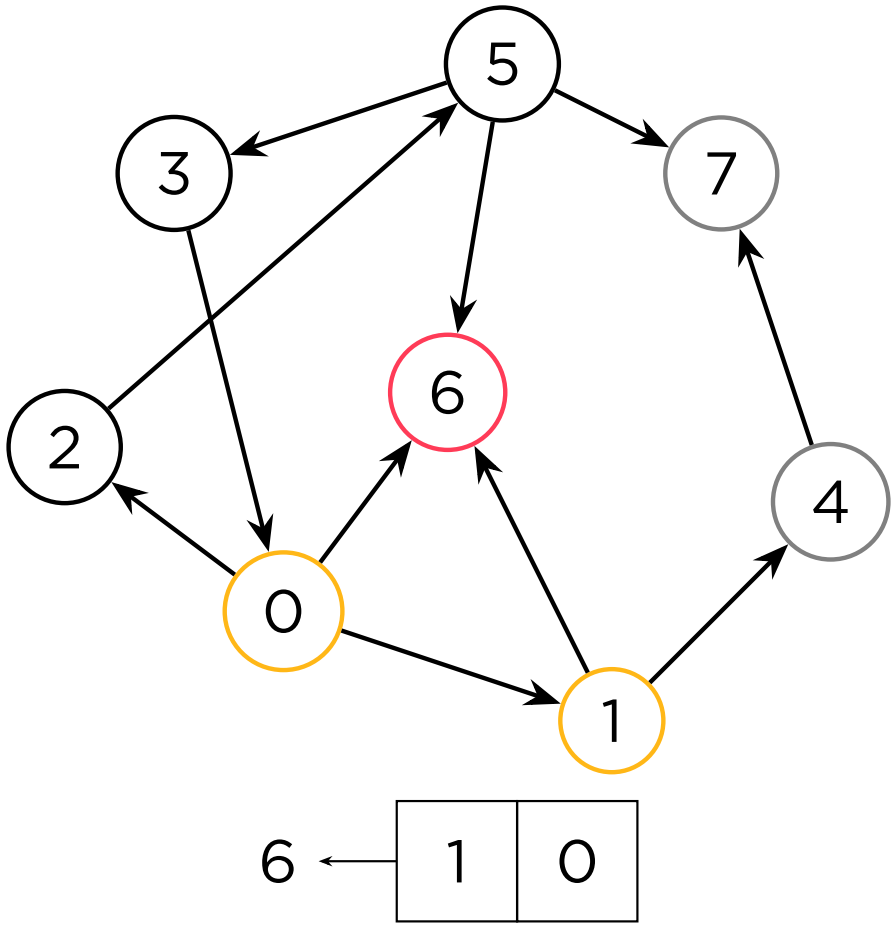
스택 맨 위의 4번 노드에는 미방문한 인접한 정점이 없으므로, 역시 스택에서 제거

그래프의 탐색 - DFS



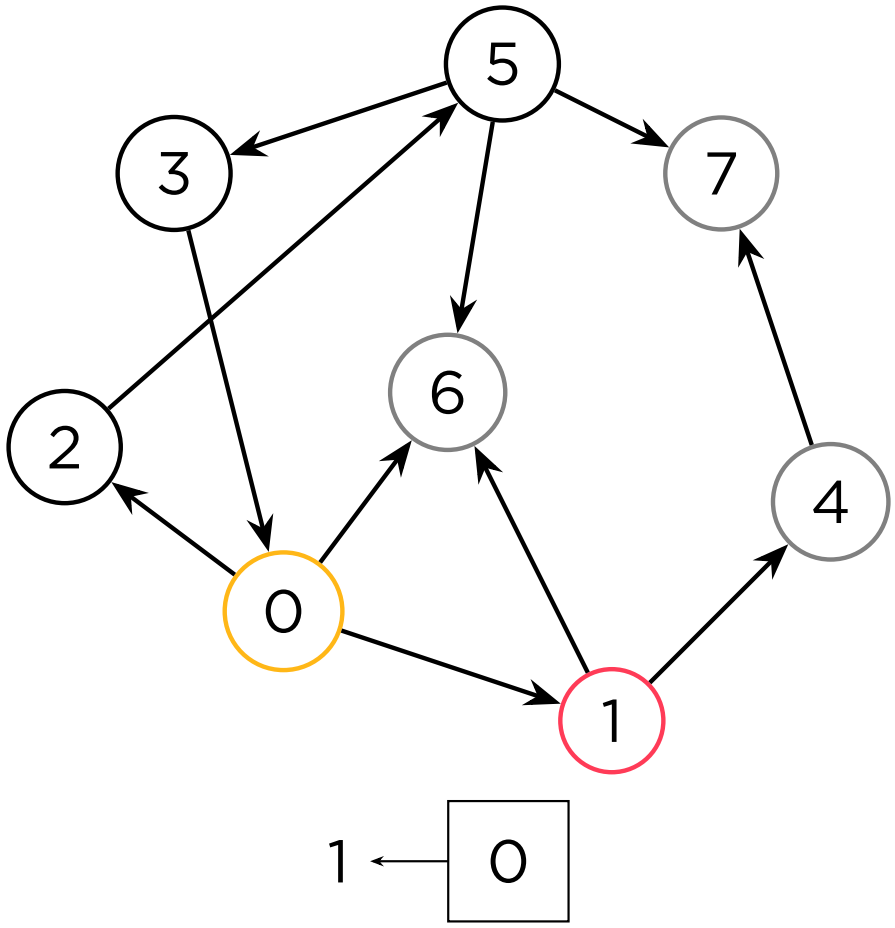
스택 맨 위의 1번 노드에 인접한 6번 노드는 아직 미방문이므로 스택에 넣음

그래프의 탐색 - DFS



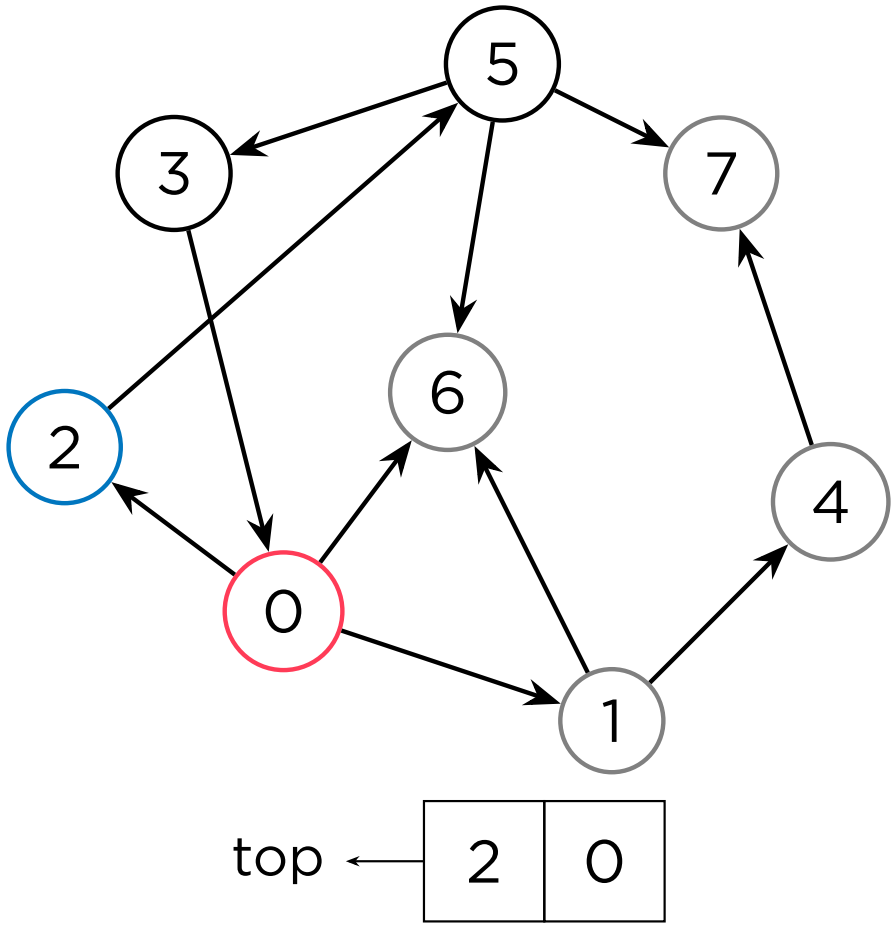
스택 맨 위의 6번 노드에는 미방문한 인접한 정점이 없으므로,
스택에서 제거

그래프의 탐색 - DFS



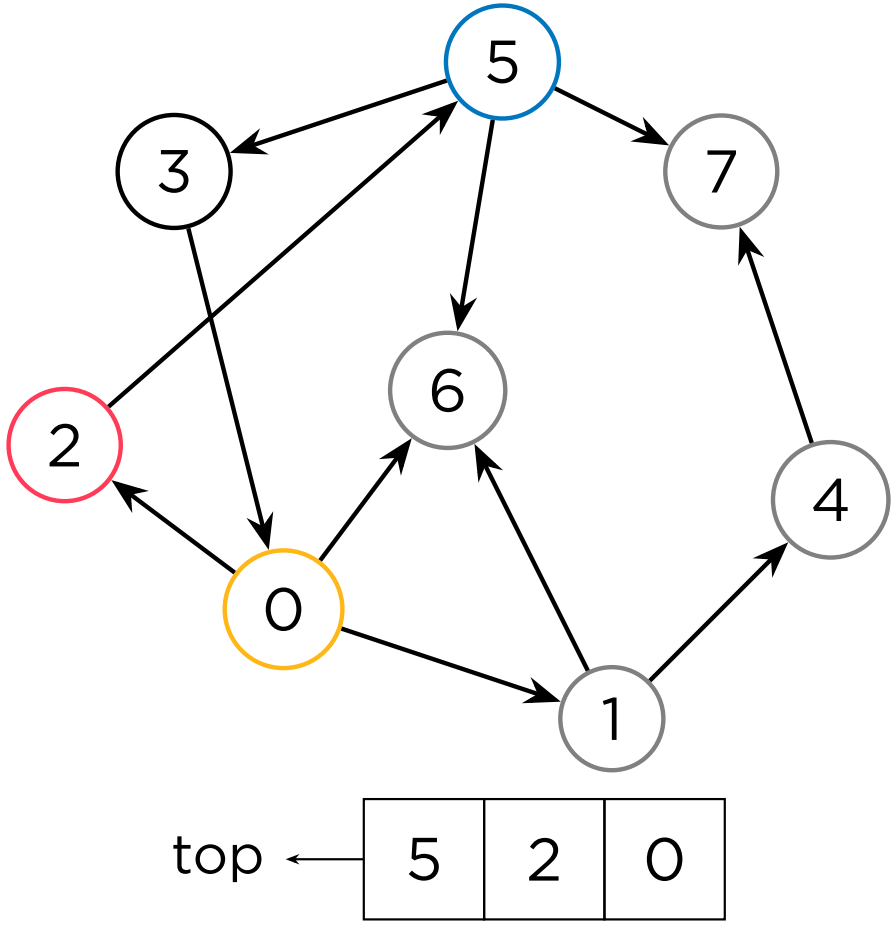
스택 맨 위의 1번 노드에 인접한 정점은 모두 방문했으므로, 스택에서 제거

그래프의 탐색 - DFS



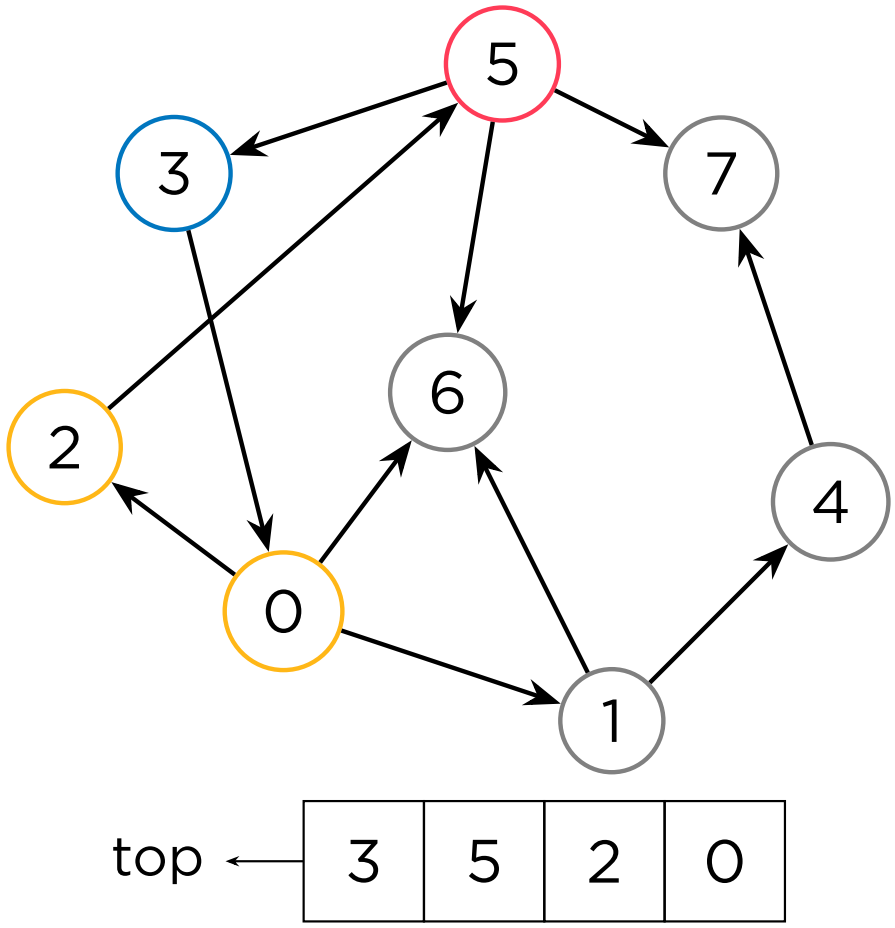
스택 맨 위의 0번 노드에 인접한 2번 노드는 아직 미방문이므로 스택에 넣음

그래프의 탐색 - DFS



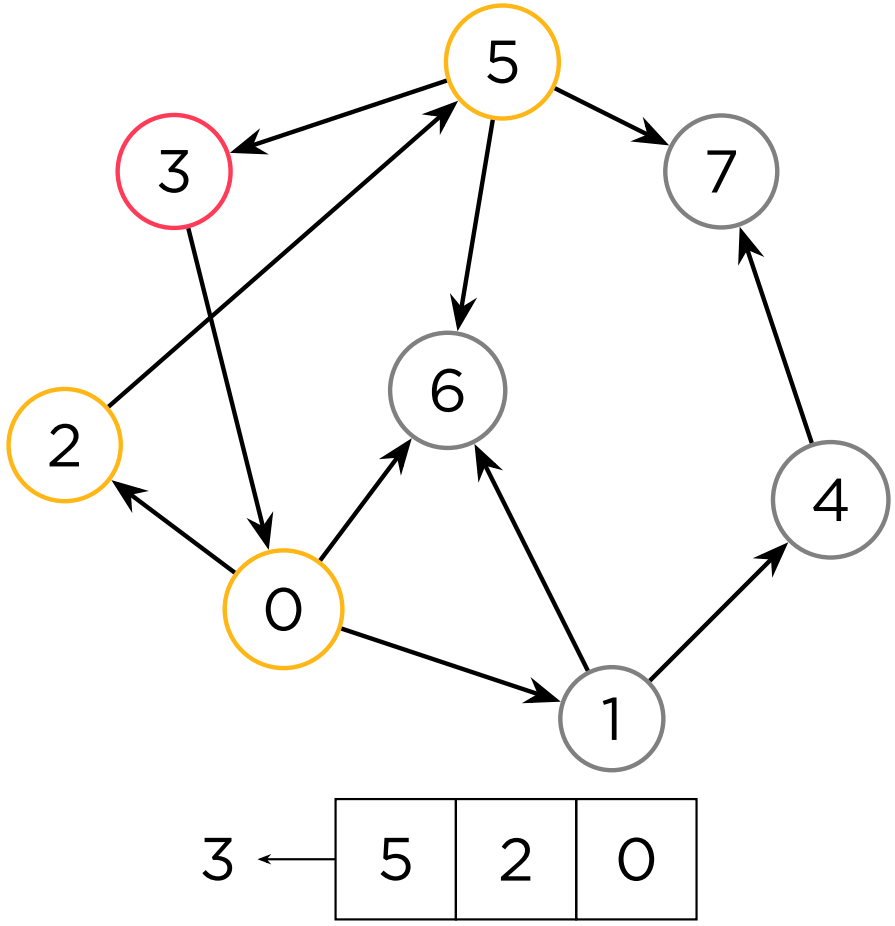
스택 맨 위의 2번 노드에 인접한 5번 노드는 아직 미방문이므로 스택에 넣음

그래프의 탐색 – DFS



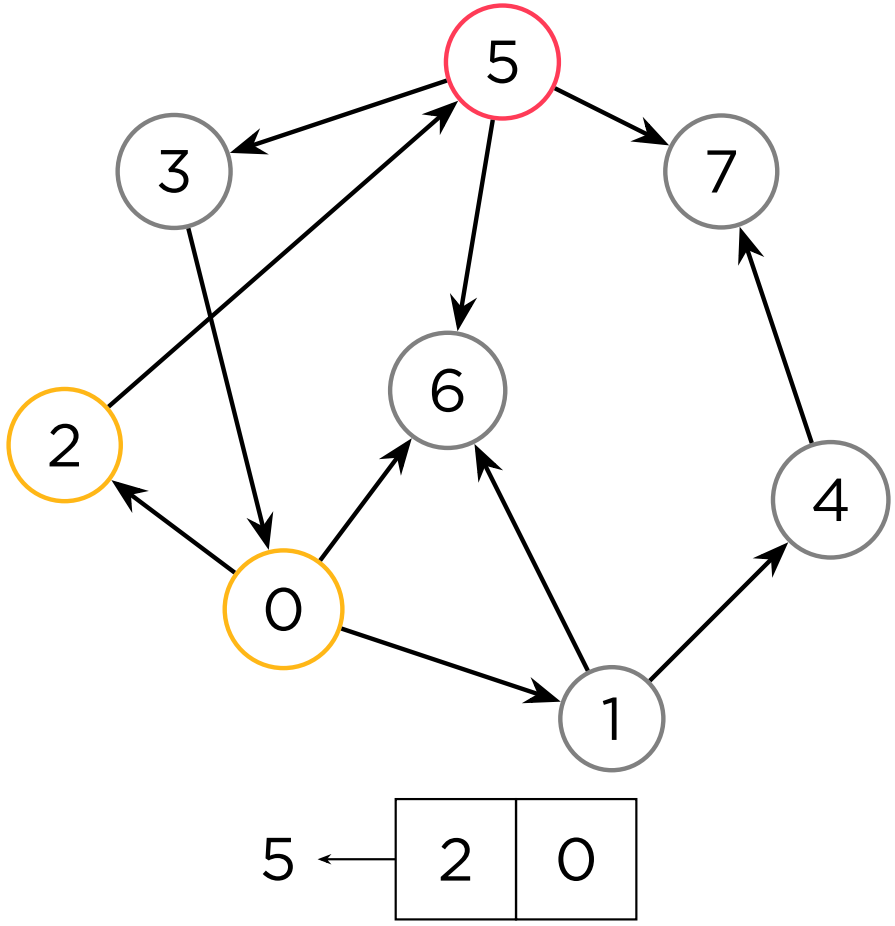
스택 맨 위의 5번 노드에 인접한 3번 노드는 아직 미방문이므로 스택에 넣음

그래프의 탐색 - DFS



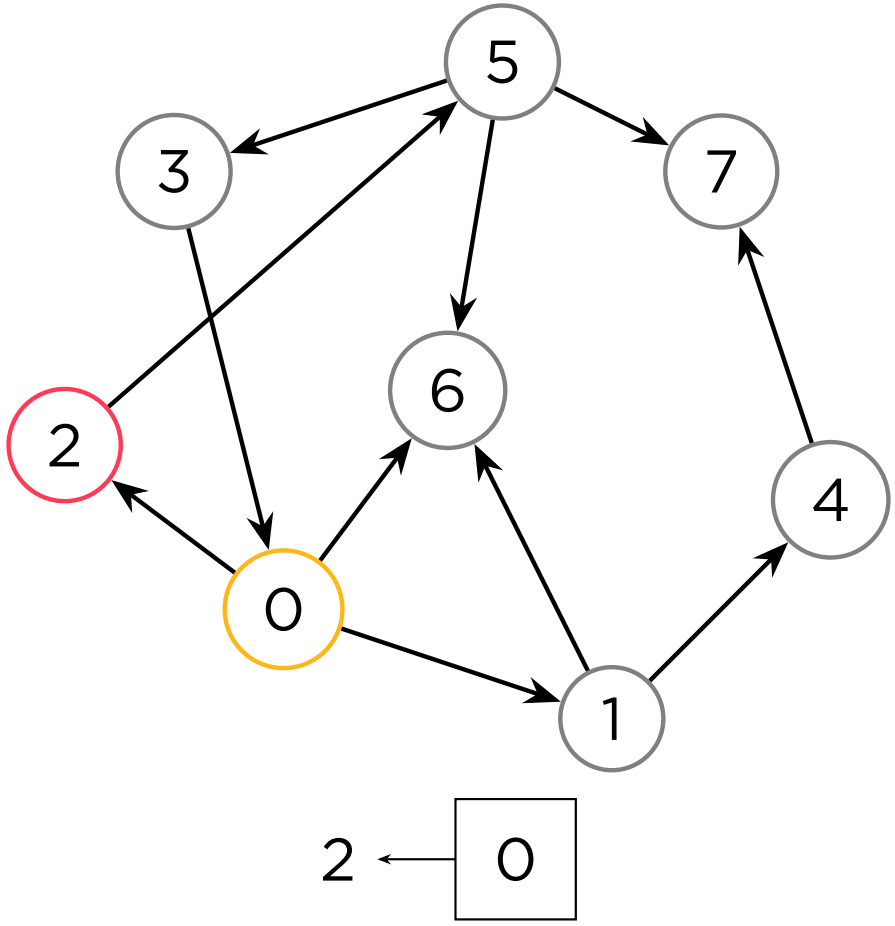
스택 맨 위의 3번 노드에 인접한 정점은 모두 방문했으므로, 스택에서 제거

그래프의 탐색 - DFS



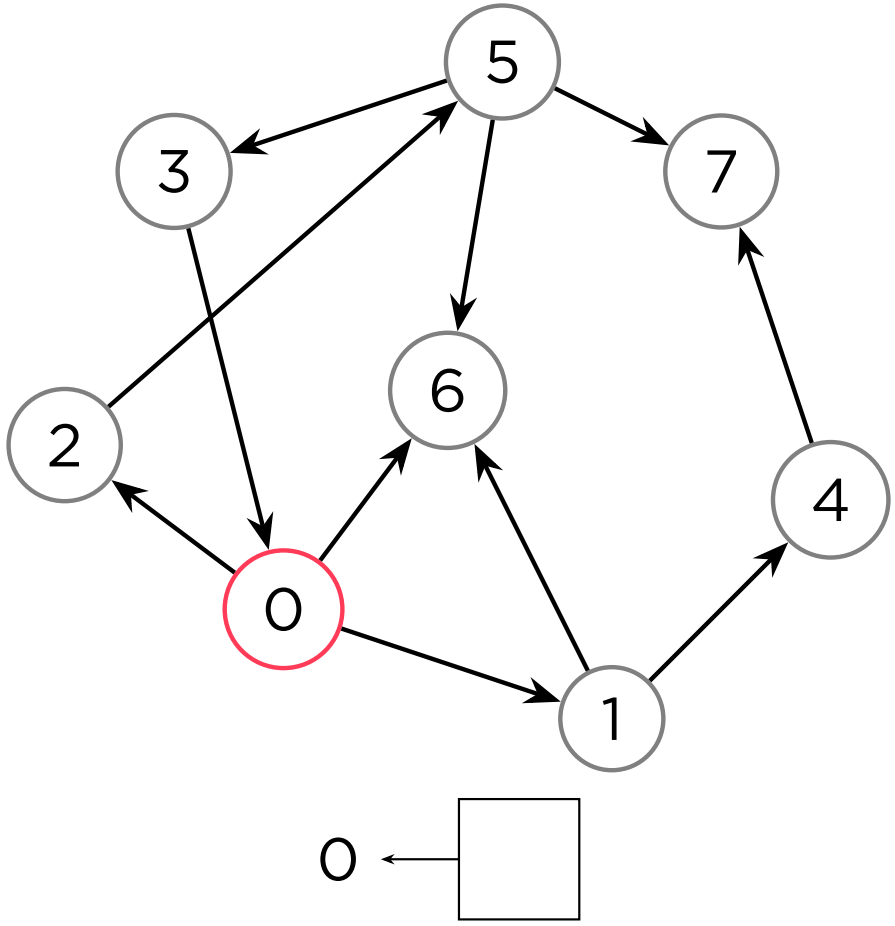
스택 맨 위의 5번 노드에 인접한 정점은 모두 방문했으므로, 스택에서 제거

그래프의 탐색 - DFS



스택 맨 위의 2번 노드에 인접한 정점은 모두 방문했으므로, 스택에서 제거

그래프의 탐색 - DFS



스택 맨 위의 0번 노드에 인접한 정점은 모두 방문했으므로, 스택에서 제거

그래프의 탐색 - DFS

```
1  const int N;
2  vector<int> graph[N];
3  bool visit[N];
4
5  void dfs(int u) {
6      if (visit[u]) return;
7      cout << u << ' ';
8      for (int v : graph[u]) {
9          if (visit[v]) continue;
10         dfs(v);
11     }
12 }
13
14 int main() {
15     // ...
16     dfs(0);
17     // ...
18 }
```

예제 코드

그래프의 탐색 - DFS

```
1  const int N;
2  vector<int> graph[N];
3  bool visit[N];
4
5  void dfs(int u) {
6      if (visit[u]) return;
7      cout << u << ' ';
8      for (int v : graph[u]) {
9          if (visit[v]) continue;
10         dfs(v);
11     }
12 }
13
14 int main() {
15     // ...
16     dfs(0);
17     // ...
18 }
```

초기 설정(이랄 것도 없음 사실). 0번 노드부터 방문함을 전제로 한다

그래프의 탐색 - DFS

```cpp
const int N;
vector<int> graph[N];
bool visit[N];

void dfs(int u) {
    if (visit[u]) return;
    cout << u << ' ';
    for (int v : graph[u]) {
        if (visit[v]) continue;
        dfs(v);
    }
}

int main() {
    // ...
    dfs(0);
    // ...
}
```

u에 인접한 노드 v를 방문하는 재귀 함수

시간 복잡도

DFS, BFS 모두..

- 인접 리스트를 사용했을 경우 $\mathcal{O}(\|V\| + \|E\|)$
- 인접 행렬을 사용했을 경우 $\mathcal{O}\left(\|V\|^2\right)$

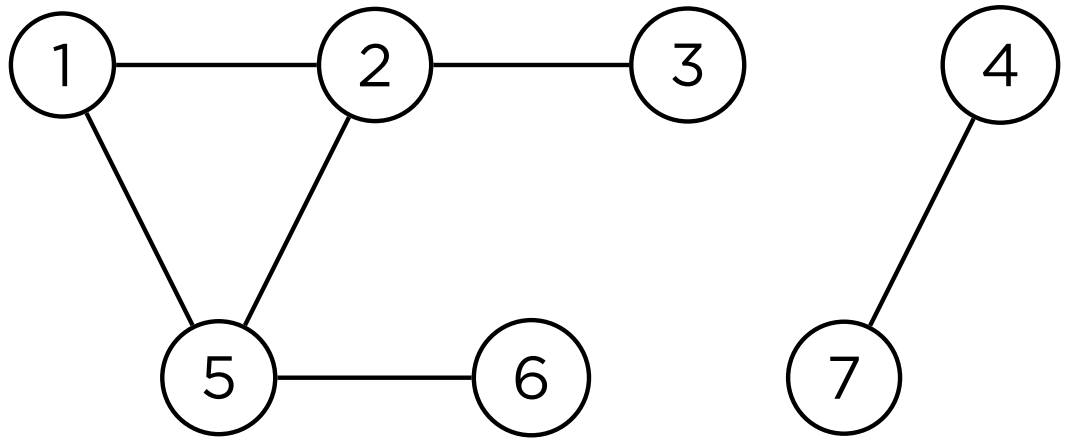
로 같으므로 상황에 따라 유리한 탐색 알고리즘을 사용

바이러스 BOJ #2606

한 컴퓨터가 웜 바이러스에 걸리면 그 컴퓨터와 네트워크 상에서 연결되어 있는 모든 컴퓨터는 웜 바이러스에 걸리게 된다.
어느 날 1번 컴퓨터가 웜 바이러스에 걸렸다. 1번 컴퓨터를 통해 웜 바이러스에 걸리게 되는 컴퓨터의 수를 출력하는 프로그램을 작성하시오.

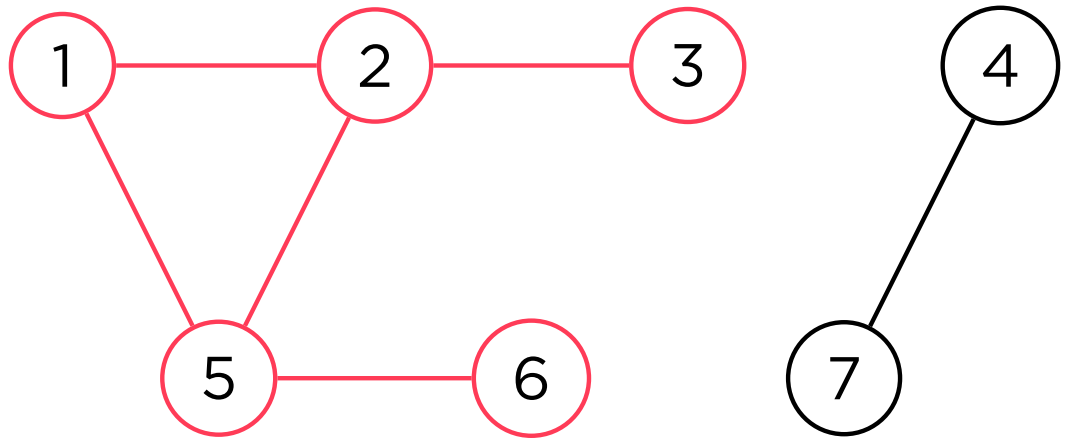
- 컴퓨터의 수 ≤ 100

바이러스 BOJ #2606



이런 네트워크가 있을 때

바이러스 BOJ #2606



1번이 감염되면 2, 3, 5, 6이 같이 감염된다

이렇게 연결되어 있는 덩어리를 **연결 요소**Connected Component 라고 하며, BFS/DFS 한 번으로 구할 수 있다

바이러스 BOJ #2606

```
7
6
1 2
2 3
1 5
5 2
5 6
4 7
```

첫째 줄에는 컴퓨터의 수가 주어진다. 둘째 줄에는 네트워크 상에서 직접 연결되어 있는 컴퓨터 쌍의 수가 주어진다. 이어서 그 수만큼 한 줄에 한 쌍씩 네트워크 상에서 직접 연결되어 있는 컴퓨터의 번호 쌍이 주어진다.

- 예제 입력을 보고 생각했을 때 그래프를 어떤 형식으로 저장하는 것이 좋을까?

바이러스 BOJ #2606

입력이 원래부터 인접 행렬이었다

- 인접 행렬!

이외 그냥 일반적인 케이스

- 인접 리스트!

(일반적이나 절대적인 것은 아니므로 참고만)

바이러스 BOJ #2606

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

vector<int> graph[101];
bool visit[101];
int main() {
    int n, m;
    cin >> n >> m;
    while (m--) {
        int u, v;
        cin >> u >> v;
        graph[u].emplace_back(v);
        graph[v].emplace_back(u);
    }
```

양방향 간선 $u \leftrightarrow v$는 $u \rightarrow v, v \rightarrow u$ 두 개의 단방향 간선으로 생각한다

바이러스 BOJ #2606

```
18      int s = 0;
19      queue<int> q;
20      visit[1] = true;
21      q.emplace(1);
22      while (!q.empty()) {
23          int u = q.front();
24          q.pop();
25          s++;
26          for (int v : graph[u]) {
27              if (visit[v]) continue;
28              visit[v] = true;
29              q.emplace(v);
30          }
31      }
32      cout << s - 1;
```

1번 노드부터 방문할 것이므로 1번 노드를 큐에 넣어 준다 (DFS를 할 경우 dfs(1))

바이러스 BOJ #2606

```
18      int s = 0;
19      queue<int> q;
20      visit[1] = true;
21      q.emplace(1);
22      while (!q.empty()) {
23          int u = q.front();
24          q.pop();
25          s++;
26          for (int v : graph[u]) {
27              if (visit[v]) continue;
28              visit[v] = true;
29              q.emplace(v);
30          }
31      }
32      cout << s - 1;
```

DFS/BFS를 하면서 방문하는 정점마다 s를 1 증가시켜 준다

바이러스 BOJ #2606

```
32      cout << s - 1;
33
34      return 0;
35  }
```

그러면 s는 1번 컴퓨터를 포함해 감염된 컴퓨터의 대수가 되므로, 1번 컴퓨터를 제외한 $s-1$이 답이다

유기농 배추 BOJ #1012

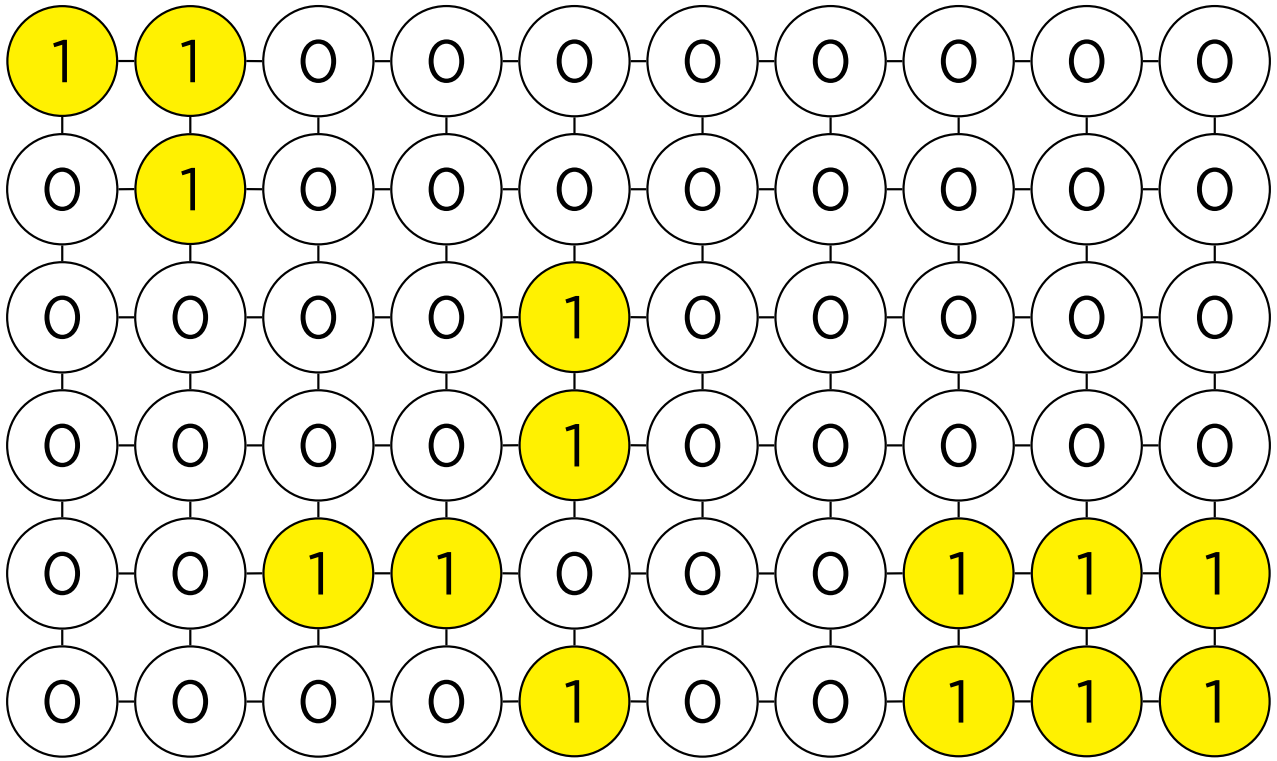
배추흰지렁이는 배추근처에 서식하며 해충을 잡아 먹음으로써 배추를 보호한다.
어떤 배추에 배추흰지렁이가 한 마리라도 살고 있으면 이 지렁이는 인접한 다른 배추로 이동할 수 있어, 그 배추들 역시 해충으로부터 보호받을 수 있다.
따라서 서로 인접해 있는 배추들이 몇 군데에 퍼져 있는지 조사하면 총 몇 마리의 지렁이가 필요한지 알 수 있다.

유기농 배추 BOJ #1012

1	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	1	1	1
0	0	0	0	1	0	0	1	1	1

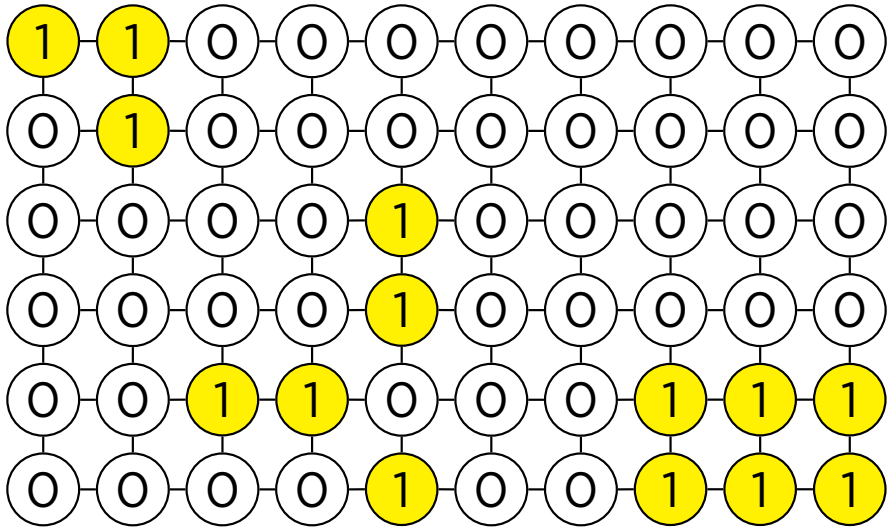
이게 뭐지···?!

유기농 배추 BOJ #1012



이런 그래프의 연결 요소의 개수를 세라는 뜻이다

유기농 배추 BOJ #1012



이걸 인접 행렬이라고 하지는 않지만 정점과 간선이 되게 규칙적이기 때문에 **그대로 저장하고** 탐색할 때 인접한 정점을 체크하는 대신 **인접한 칸을 체크**하면 된다

유기농 배추 BOJ #1012

```cpp
int board[N][M];
bool visit[N][M];
int dx[4] = {-1, 1, 0, 0};
int dy[4] = {0, 0, -1, 1};

// ...

while (!q.empty()) {
    int x = q.front().first;
    int y = q.front().second;
    q.pop();

    for (int d = 0; d < 4; d++) {
        int nx = x + dx[d], ny = y + dy[d];
        if (0 > nx || nx ⩾ n) continue;
        if (0 > ny || ny ⩾ m) continue;
        if (visit[nx][ny]) continue;
        visit[nx][ny] = true;
        q.emplace(make_pair(nx, ny));
    }
}
```

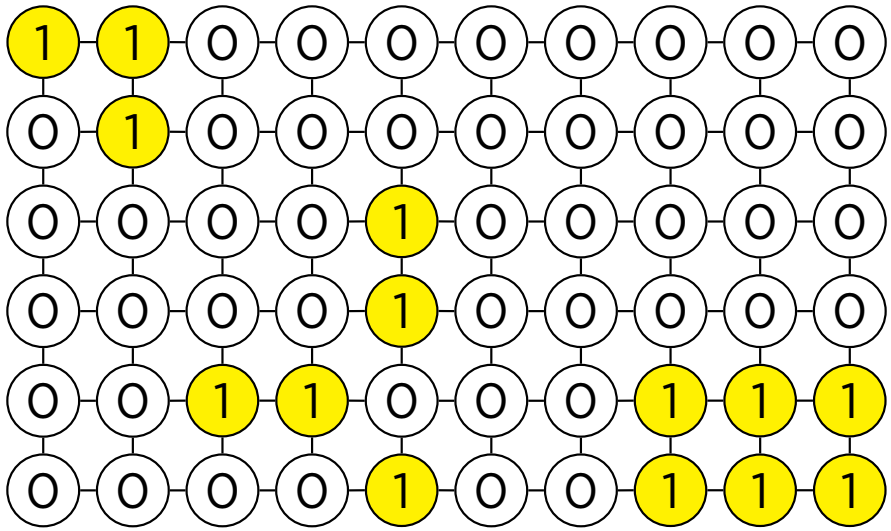
BFS라면 이런 식! DFS도 똑같다 (※ 이거 문제 정답 코드 아님)

유기농 배추 BOJ #1012

```
1  int board[N][M];
2  bool visit[N][M];
3  int dx[4] = {-1, 1, 0, 0};
4  int dy[4] = {0, 0, -1, 1};
5
6  // ...
7
8  while (!q.empty()) {
9      int x = q.front().first;
10     int y = q.front().second;
11     q.pop();
12
13     for (int d = 0; d < 4; d++) {
14         int nx = x + dx[d], ny = y + dy[d];
15         if (0 > nx || nx ⩾ n) continue;
16         if (0 > ny || ny ⩾ m) continue;
17         if (visit[nx][ny]) continue;
18         visit[nx][ny] = true;
19         q.emplace(make_pair(nx, ny));
20     }
21 }
```

좌표 범위를 신경써주자. 세그폴트 날 수 있다

유기농 배추 BOJ #1012



그러면 연결 요소의 개수는 어떻게 셀까?

유기농 배추 BOJ #1012

`visit` 배열을 참고해, 모든 칸을 확인하면서

- 확인하고 있는 칸이 0이라면 스킵
- 방문하지 않은 칸이라면 BFS/DFS를 돌려서 인접한 칸들을 모두 방문한다 (연결 요소 $+1$)
- 방문한 칸이라면 스킵

유기농 배추 BOJ #1012

```cpp
#include <iostream>
#include <queue>
#include <cstring>

using namespace std;
using pii = pair<int, int>;

int board[50][50];
bool visit[50][50];
int dx[4] = {-1, 1, 0, 0};
int dy[4] = {0, 0, -1, 1};
```

이런 그래프를 다룰 땐 pair<int, int>를 엄청 많이 사용하는데... 길기 때문에 pii 등으로 줄인다

유기농 배추 BOJ #1012

```
13  int main() {
14      int t;
15      cin >> t;
16
17      while (t--) {
18          memset(board, 0, sizeof(board));
19          memset(visit, 0, sizeof(visit));
20
21          int n, m, k;
22          cin >> n >> m >> k;
23          while (k--) {
24              int x, y;
25              cin >> x >> y;
26              board[x][y] = 1;
27          }
```

`memset(a, 0, sizeof(a))`는 배열 a를 전부 0으로 초기화한다 (0, -1 이외엔 안 된다)

유기농 배추 BOJ #1012

```
29          int cc = 0;
30          queue<pii> q;
31          for (int i = 0; i < n; i++) {
32              for (int j = 0; j < m; j++) {
33                  if (board[i][j] == 0) continue;
34                  if (visit[i][j]) continue;
35                  visit[i][j] = true;
36                  q.emplace(pii(i, j));
37                  cc++;
```

모든 칸을 보되, 0인 칸과 이미 방문한 칸은 제외.

미방문한 칸이라는 건 아직 확인하지 않은 연결 요소라는 뜻이므로

연결 요소의 개수 $+1$

유기농 배추 BOJ #1012

```
39              while (!q.empty()) {
40                  int x = q.front().first;
41                  int y = q.front().second;
42                  q.pop();
43
44                  for (int d = 0; d < 4; d++) {
45                      int nx = x + dx[d];
46                      int ny = y + dy[d];
47                      if (0 > nx || nx ≥ n) continue;
48                      if (0 > ny || ny ≥ m) continue;
49                      if (visit[nx][ny]) continue;
50                      if (board[nx][ny] == 0) continue;
51                      visit[nx][ny] = true;
52                      q.emplace(pii(nx, ny));
53                  }
54              }
```

DFS/BFS 하는 부분, 위에서 언급한 인접 칸 확인 처리에 유의.

추가적으로 인접 칸이 0인 경우에도 스킵한다

유기농 배추 BOJ #1012

```
55                  }
56              }
57
58              cout << cc << '\n';
59          }
60
61          return 0;
62      }
```

그러면 연결 요소의 개수는 cc개!

문제 풀어보고, 질문하는 시간 (-17시까지)