Algorithm

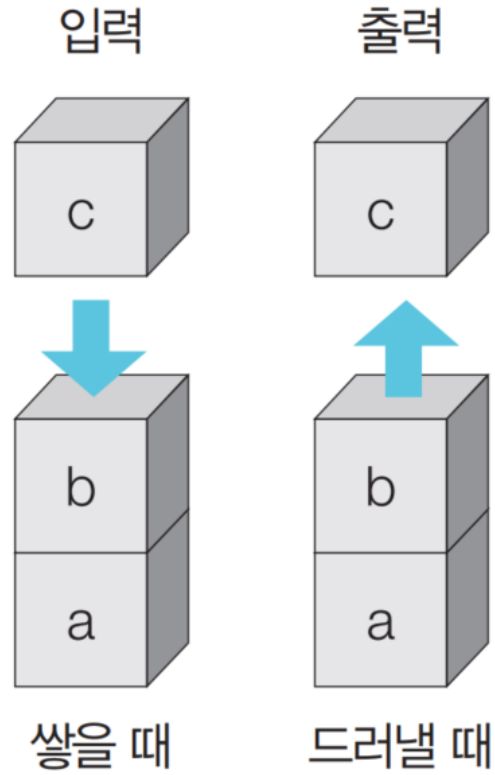
동양미래대학교 강은영

그래프 탐색 알고리즘: DFS/BFS

- 탐색(Search)이란 많은 양의 데이터 중에서 원하는 데이터를 찾는 과정을 말합니다.
- 대표적인 그래프 탐색 알고리즘으로는 DFS와 BFS가 있습니다.
- DFS/BFS는 코딩 테스트나 인터뷰에서 매우 자주 등장하는 유형입니다.

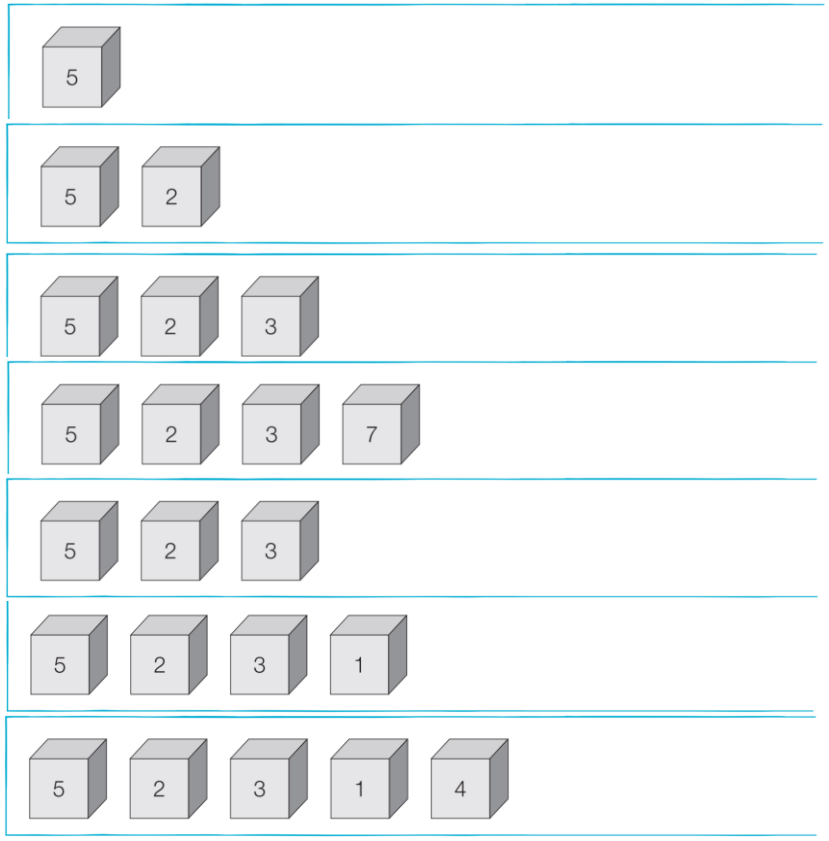
스택자료구조

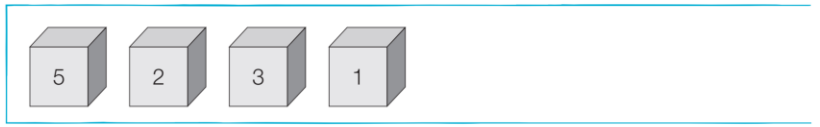
- 먼저 들어 온 데이터가 나중에 나가는 형식(선입후출)의 자료구조입니다
- 입구와 출구가 동일한 형태로 스택을 시각화할 수 있습니다.



스택 동작 예시

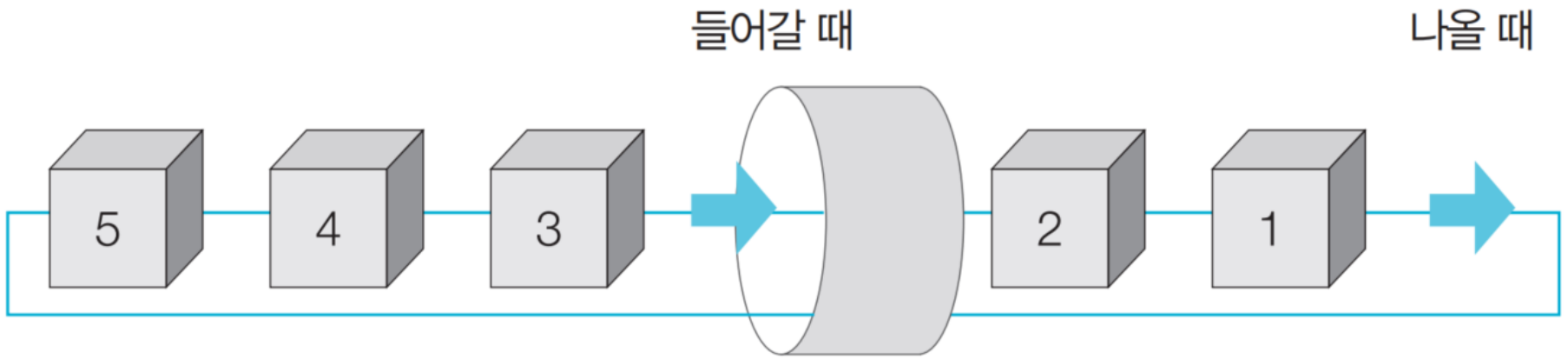
- 삽입(5) – 삽입(2) – 삽입(3) – 삽입(7) – 삭제() – 삽입(1) – 삽입(4) – 삭제()





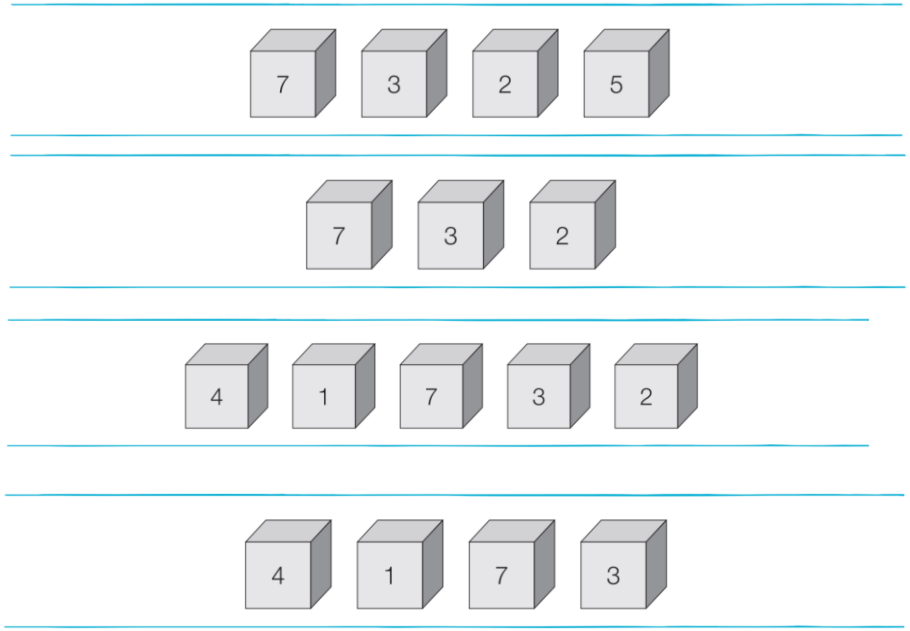
큐 자료구조

- 먼저 들어 온 데이터가 먼저 나가는 형식(선입선출)의 자료구조입니다.
- 큐는 입구와 출구가 모두 뚫려 있는 터널과 같은 형태로 시각화 할 수 있습니다.



큐 동작 예시

- 삽입(5) – 삽입(2) – 삽입(3) – 삽입(7) – 삭제() – 삽입(1) – 삽입(4) – 삭제()



재귀 함수

- 재귀 함수(Recursive Function)란 자기 자신을 다시 호출하는 함수를 의미합니다.
- 단순한 형태의 재귀 함수 예제
  - '재귀 함수를 호출합니다.'라는 문자열을 무한히 출력합니다.
  - 어느 정도 출력하다가 최대 재귀 깊이 초과 메시지가 출력됩니다.

```
def recursive_function():
    print('재귀 함수를 호출합니다.')
    recursive_function()

recursive_function()
```

재귀 함수의 종료 조건

- 재귀 함수를 문제 풀이에서 사용할 때는 재귀 함수의 종료 조건을 반드시 명시해야 합니다.
- 종료 조건을 제대로 명시하지 않으면 함수가 무한히 호출될 수 있습니다.
  - 종료 조건을 포함한 재귀 함수 예제

```
def recursive_function(i):
    # 100번째 호출을 했을 때 종료되도록 종료 조건 명시
    if i == 100:
        return
    print(i, '번째 재귀함수에서', i + 1, '번째 재귀함수를 호출합니다.')
    recursive_function(i + 1)
    print(i, '번째 재귀함수를 종료합니다.')

recursive_function(1)
```

재귀 함수 사용의 유의 사항

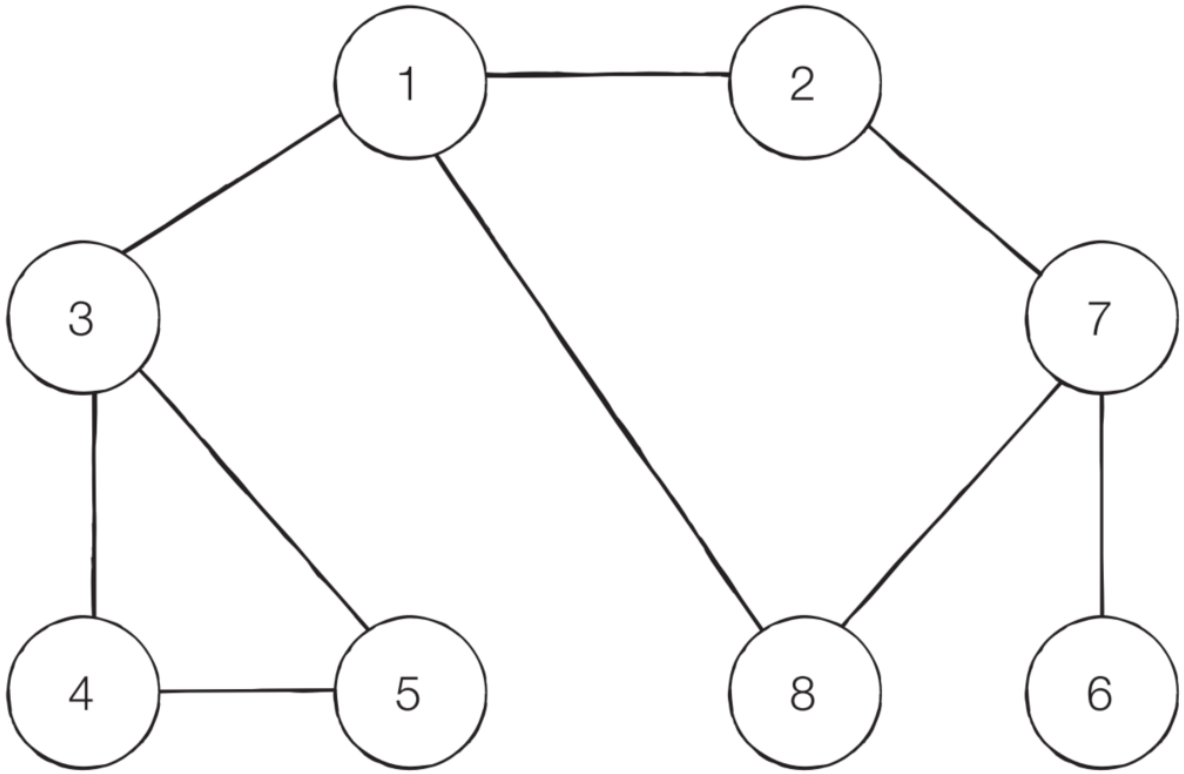
- 재귀 함수를 잘 활용하면 복잡한 알고리즘을 간결하게 작성할 수 있습니다.
  - 단, 오히려 다른 사람이 이해하기 어려운 형태의 코드가 될 수도 있으므로 신중하게 사용해야 합니다.
- 모든 재귀 함수는 반복문을 이용하여 동일한 기능을 구현할 수 있습니다.
- 재귀 함수가 반복문보다 유리한 경우도 있고 불리한 경우도 있습니다.
- 컴퓨터가 함수를 연속적으로 호출하면 컴퓨터 메모리 내부의 스택 프레임에 쌓입니다.
  - 그래서 스택을 사용해야 할 때 구현상 스택 라이브러리 대신에 재귀 함수를 이용하는 경우가 많습니다.

DFS(Depth-First Search)

- DFS는 깊이 우선 탐색이라고도 부르며 그래프에서 깊은 부분을 우선적으로 탐색하는 알고리즘입니다.
- DFS는 스택 자료구조(혹은 재귀 함수)를 이용하며, 구체적인 동작 과정은 다음과 같습니다.
  1. 탐색 시작 노드를 스택에 삽입하고 방문 처리를 합니다.
  2. 스택의 최상단 노드에 방문하지 않은 인접한 노드가 하나라도 있으면 그 노드를 스택에 넣고 방문 처리합니다. 방문하지 않은 인접 노드가 없으면 스택에서 최상단 노드를 꺼냅니다.
  3. 더 이상 2번의 과정을 수행할 수 없을 때까지 반복합니다.

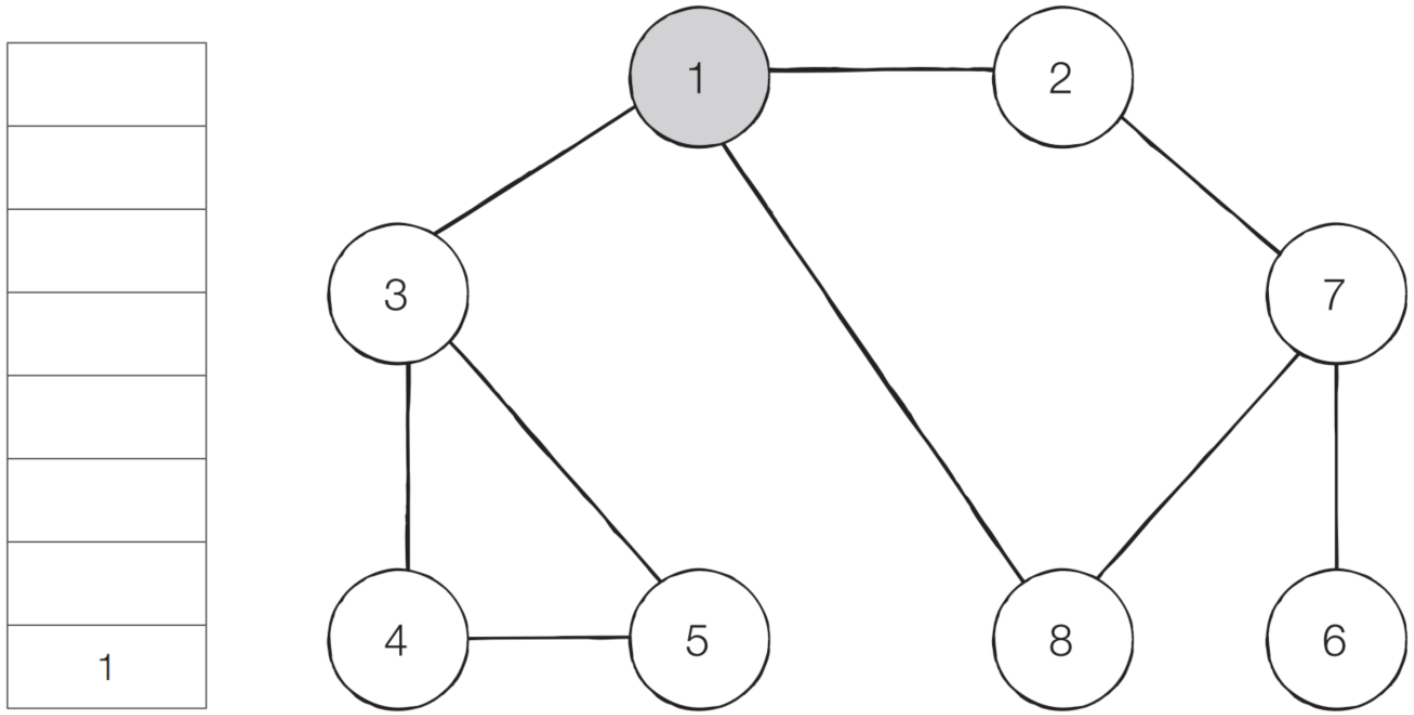
DFS 동작 예시

- [Step 0] 그래프를 준비합니다. (방문 기준: 번호가 낮은 인접 노드부터)
  - 시작 노드: 1

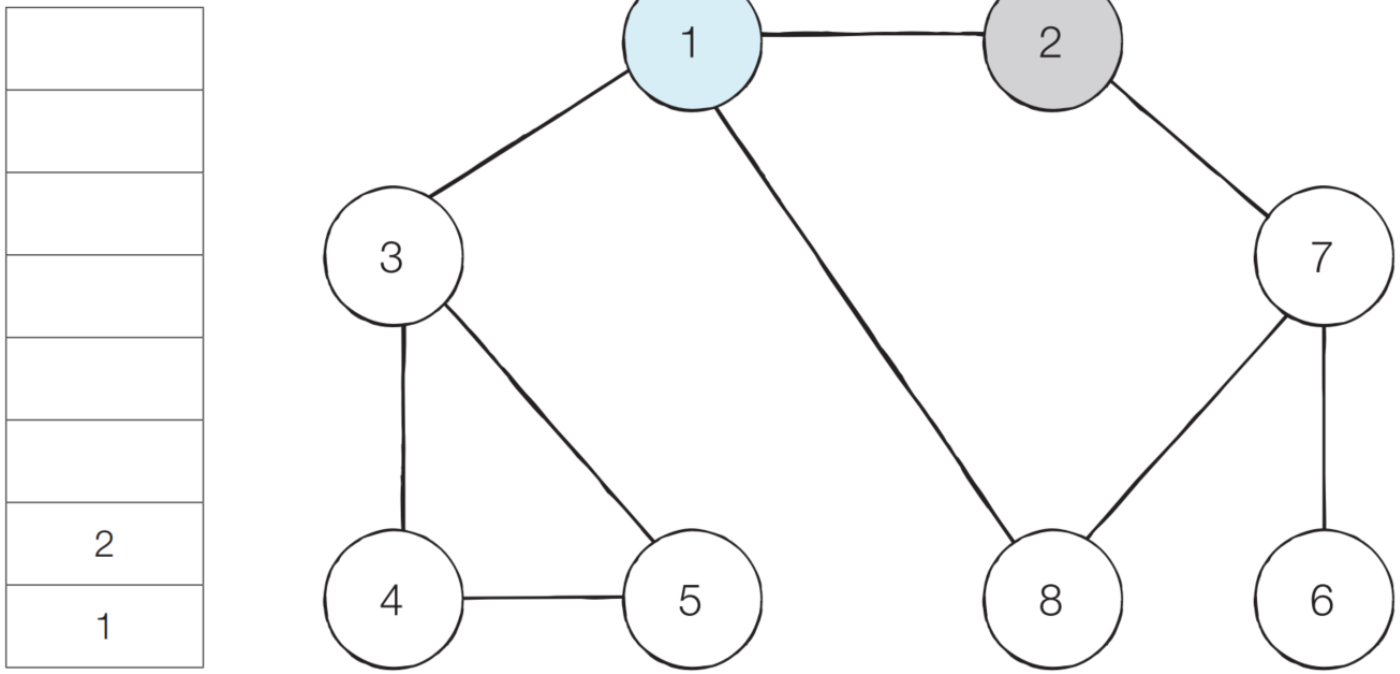


DFS 동작 예시

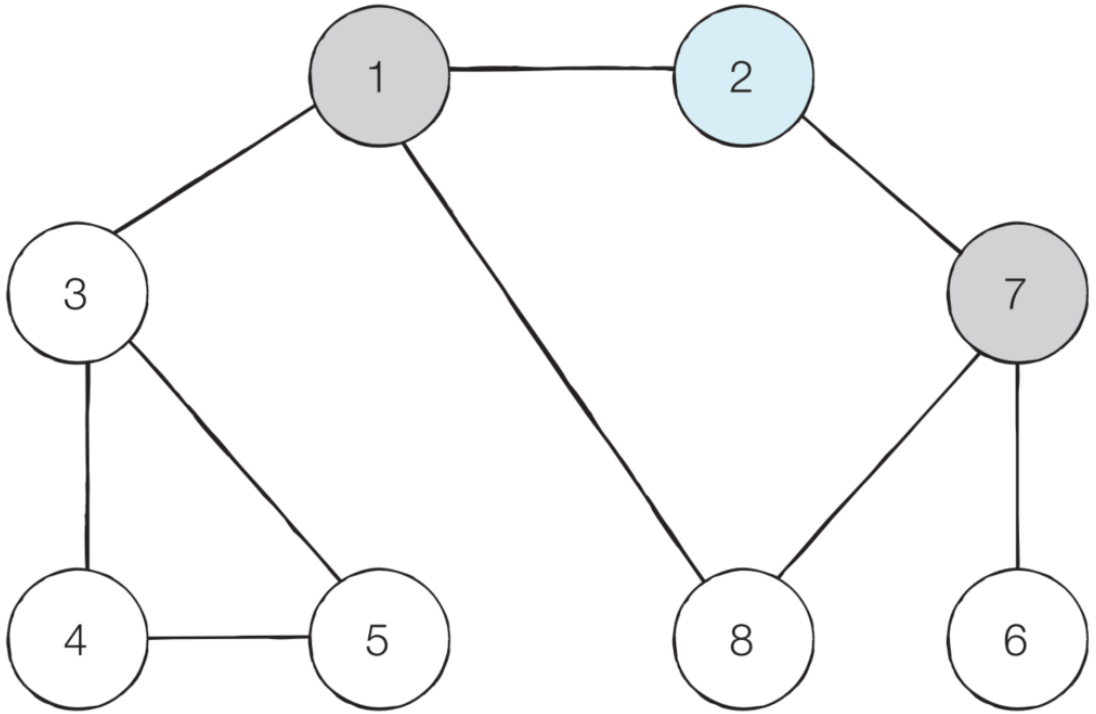
- [Step 1] 시작 노드인 '1'을 스택에 삽입하고 방문 처리를 합니다.



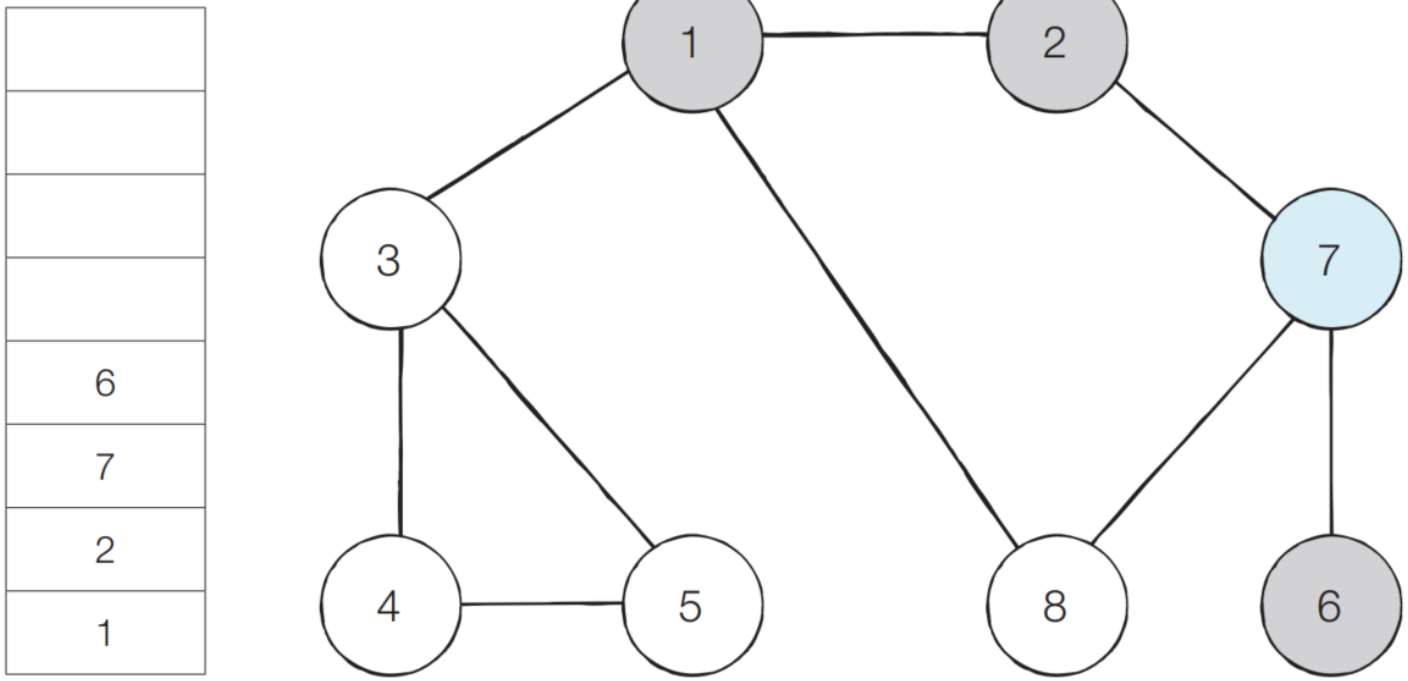
- [Step 2] 스택의 최상단 노드인 '1'에 방문하지 않은 인접 노드 '2', '3', '8'이 있습니다.
  - 이 중에서 가장 작은 노드인 '2'를 스택에 넣고 방문 처리를 합니다.



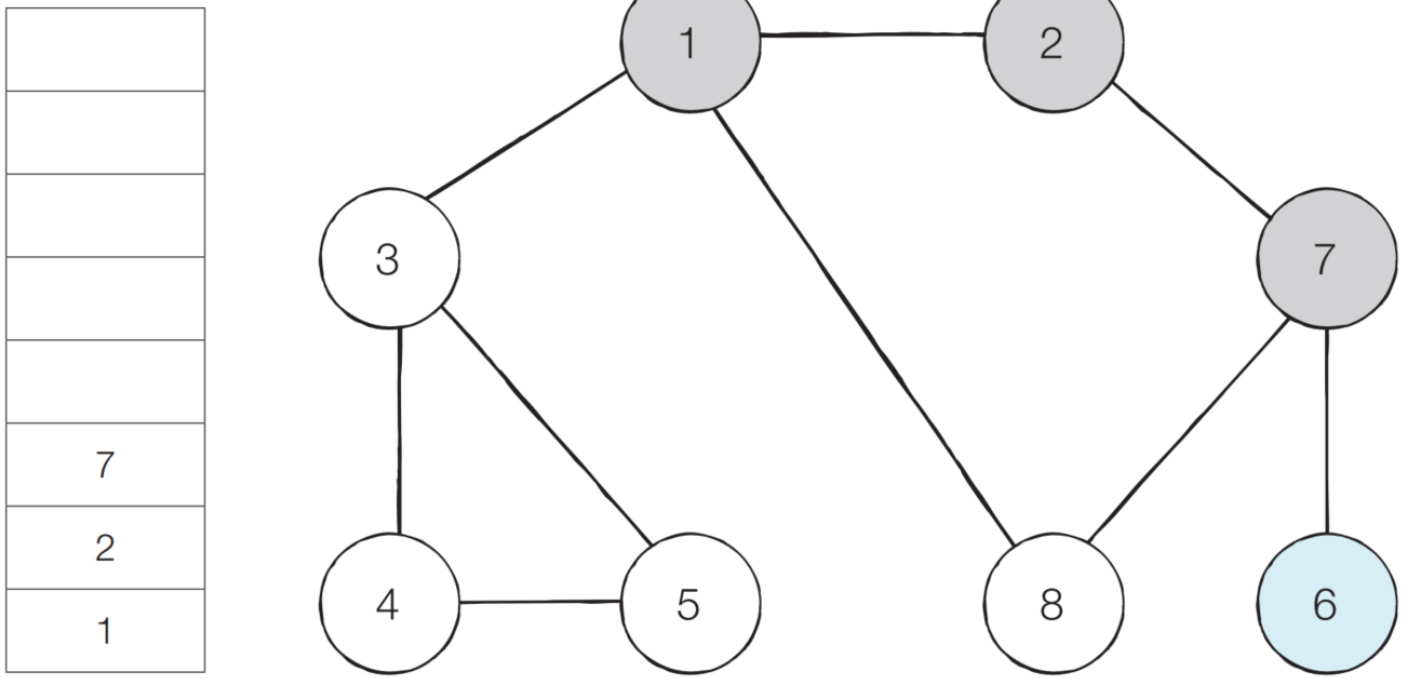
- [Step 3] 스택의 최상단 노드인 '2'에 방문하지 않은 인접 노드 '7'이 있습니다.
  - 따라서 '7'번 노드를 스택에 넣고 방문 처리를 합니다.



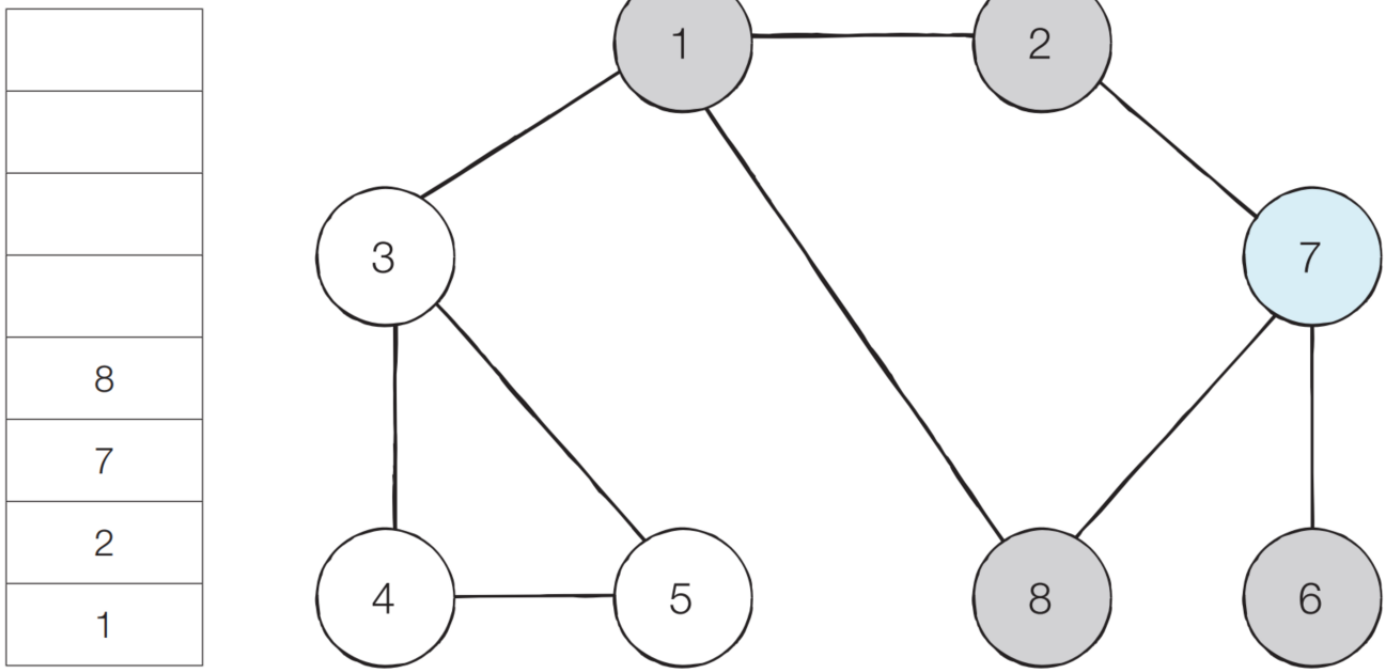
- [Step 4] 스택의 최상단 노드인 '7'에 방문하지 않은 인접 노드 '6', '8'이 있습니다.
  - 이 중에서 가장 작은 노드인 '6'을 스택에 넣고 방문 처리를 합니다.



- [Step 5] 스택의 최상단 노드인 '6'에 방문하지 않은 인접 노드가 없습니다.
  - 따라서 스택에서 '6'번 노드를 꺼냅니다.

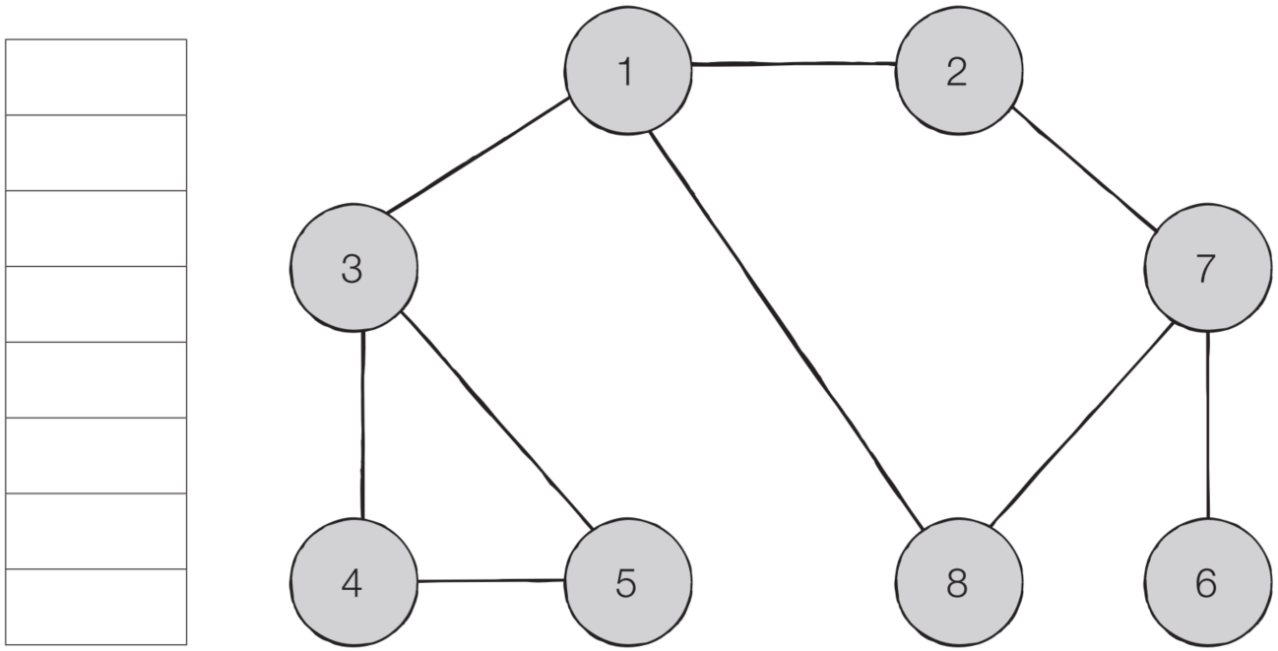


- [Step 6] 스택의 최상단 노드인 '7'에 방문하지 않은 인접 노드 '8'이 있습니다.
  - 따라서 '8'번 노드를 스택에 넣고 방문 처리를 합니다.



DFS 동작 예시

- 이러한 과정을 반복하였을 때 전체 노드의 탐색 순서(스택에 들어간 순서)는 다음과 같습니다.



탐색 순서: 1 → 2 → 7 → 6 → 8 → 3 → 4 → 5

DFS 깊이우선탐색

깊이 우선 탐색(DFS: depth-first search)은 그래프 완전 탐색 기법 중 하나다.
그래프의 시작 노드에서 출발하여 탐색할 한 쪽 분기를 정하여 최대 깊이까지 탐색을 마친다.
그 후 다른 쪽 분기로 이동하여 다시 탐색을 수행한다.

깊이 우선 탐색

기능	특징	시간 복잡도(노드 수: V, 에지 수: E)
그래프 완전 탐색	▪ 재귀 함수로 구현 ▪ 스택 자료구조 이용	O(V + E)

DFS 깊이우선탐색

1. DFS를 시작할 노드를 정한 후 사용할 자료구조 초기화하기

DFS를 위해 필요한 초기 작업은 인접 리스트로 그래프 표현하기, 방문 배열 초기화 하기, 시작 노드 스택에 삽입하기다.

인접리스트 그래프 표현

1 -> 2, 3,8
2->1,7
3->1,4,5
4->3,5
5->3,4
6->7
7->2,6,8
8->1,7

방문배열

1	2	3	4	5	6	7	8
t	F	F	F	F	F	F	F

DFS 소스코드 예제(Python)

```python
# DFS 메서드 정의
def dfs(graph, v, visited):
    # 현재 노드를 방문 처리
    visited[v] = True
    print(v, end=' ')
    # 현재 노드와 연결된 다른 노드를 재귀적으로 방문
    for i in graph[v]:
        if not visited[i]:
            dfs(graph, i, visited)
```

실행 결과

```
1 2 7 6 8 3 4 5
```

```python
# 각 노드가 연결된 정보를 표현 (2차원 리스트)
graph = [
    [],
    [2, 3, 8],
    [1, 7],
    [1, 4, 5],
    [3, 5],
    [3, 4],
    [7],
    [2, 6, 8],
    [1, 7]
]

# 각 노드가 방문된 정보를 표현 (1차원 리스트)
visited = [False] * 9

# 정의된 DFS 함수 호출
dfs(graph, 1, visited)
```

DFS 소스코드 예제(C++)

```
#include <bits/stdc++.h>

using namespace std;

bool visited[9];
int graph[9][3] = {{}, {2,3,8}, {1,7}, {1,4,5}, {3,5}, {3,4}, {7}, {2,6,8}, {1,7}};

void dfs(int x) {
    visited[x] = true;
    cout << x << ' ';
    for (int node : graph[x]) {
    // 인접한 노드가 방문한 적이 없다면 DFS 수행
        if(!visited[node]) {
            dfs(node);
        }
    }
}

int main(void) {
    dfs(1);
}
```

DFS 소스코드 예제(Java)

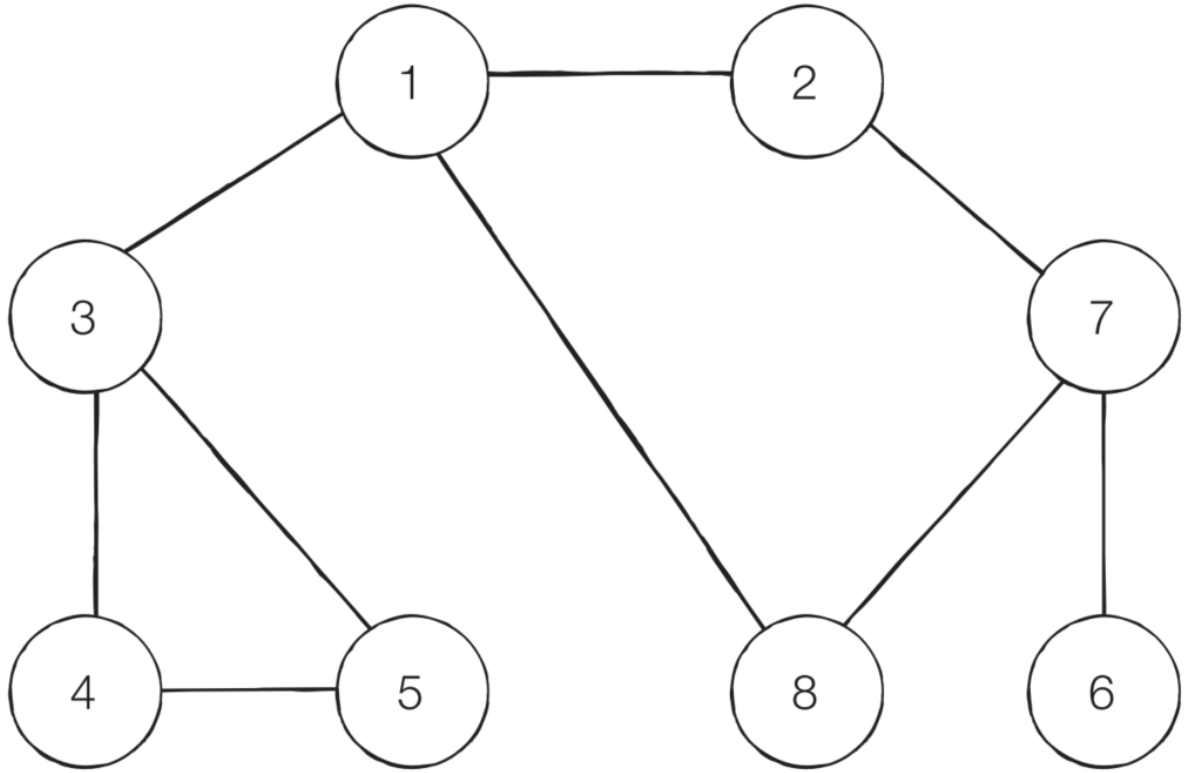
```
import java.util.*;
public class Main {
    public static boolean[] visited = new boolean[9];
     static int[][] graph = {{}, {2,3,8}, {1,7}, {1,4,5}, {3,5}, {3,4}, {7}, {2,6,8}, {1,7}};

    // DFS 함수 정의
    public static void dfs(int x) {
        // 현재 노드를 방문 처리
        visited[x] = true;
        System.out.print(x + " ");
        // 현재 노드와 연결된 다른 노드를 재귀적으로 방문
        for (int node : graph[x]) {
            if  (!visited[node]) {
                dfs(node);
            }
        }
    }

    public static void main(String[] args) {
        dfs(1);
    }
}
```

BFS(Breadth-First Search)

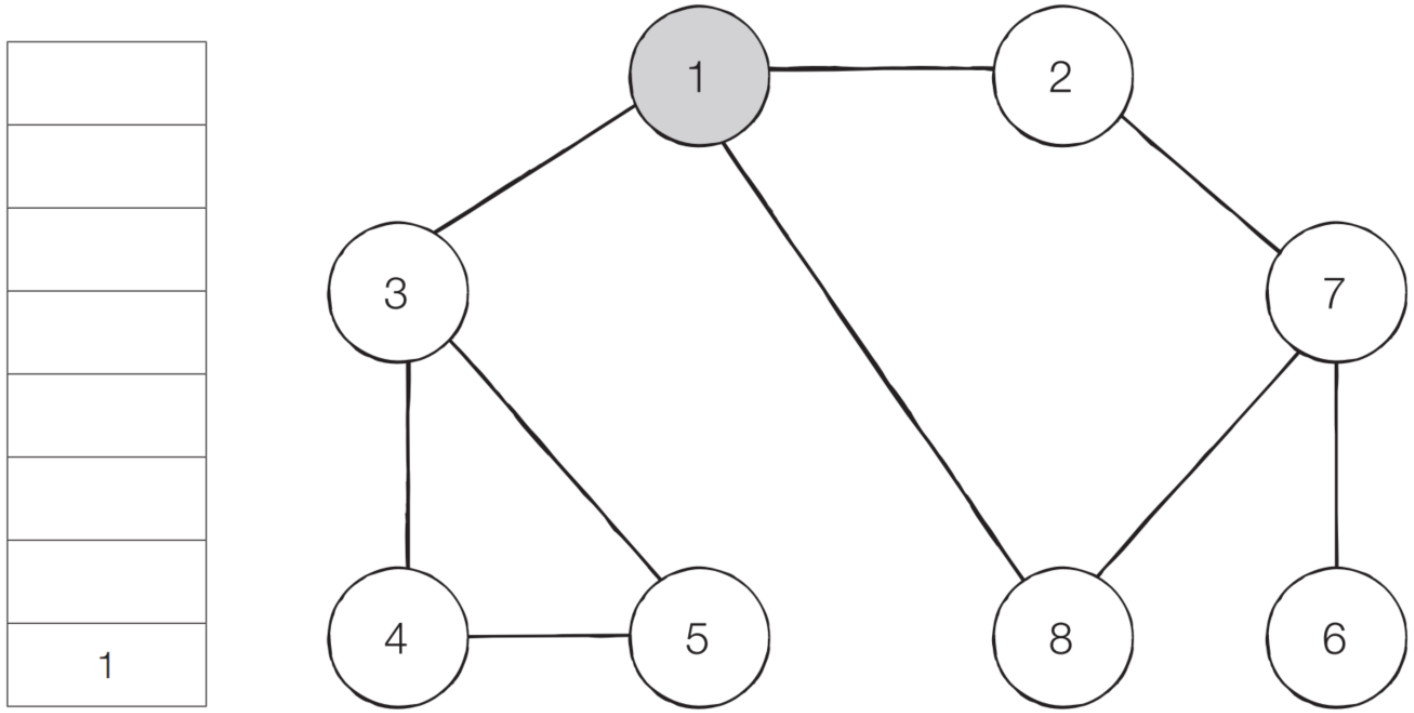
- BFS는 너비 우선 탐색이라고도 부르며, 그래프에서 가까운 노드부터 우선적으로 탐색하는 알고리즘입니다.
- BFS는 큐 자료구조를 이용하며, 구체적인 동작 과정은 다음과 같습니다.
  1. 탐색 시작 노드를 큐에 삽입하고 방문 처리를 합니다.
  2. 큐에서 노드를 꺼낸 뒤에 해당 노드의 인접 노드 중에서 방문하지 않은 노드를 모두 큐에 삽입하고 방문 처리합니다.
  3. 더 이상 2번의 과정을 수행할 수 없을 때까지 반복합니다.

BFS 동작 예시

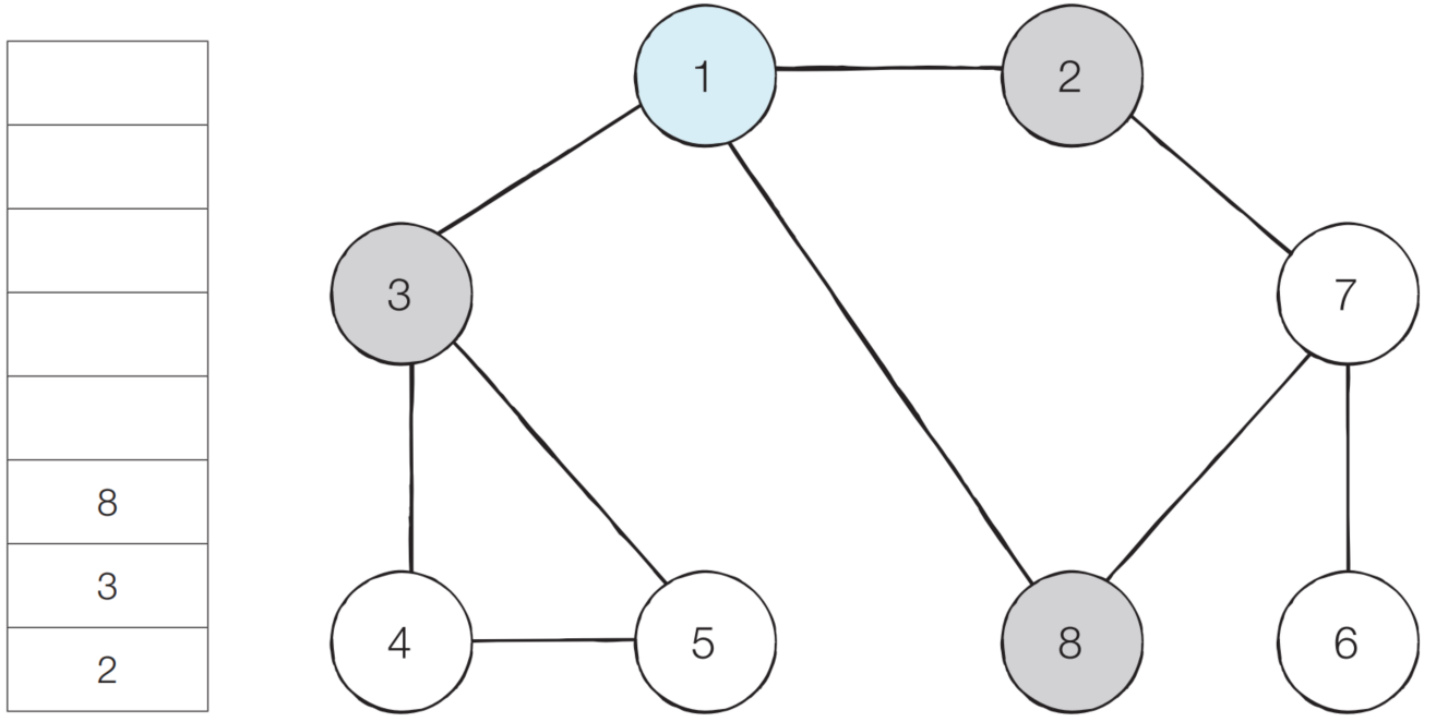
- [Step 0] 그래프를 준비합니다. (방문 기준: 번호가 낮은 인접 노드부터)
  - 시작 노드: 1



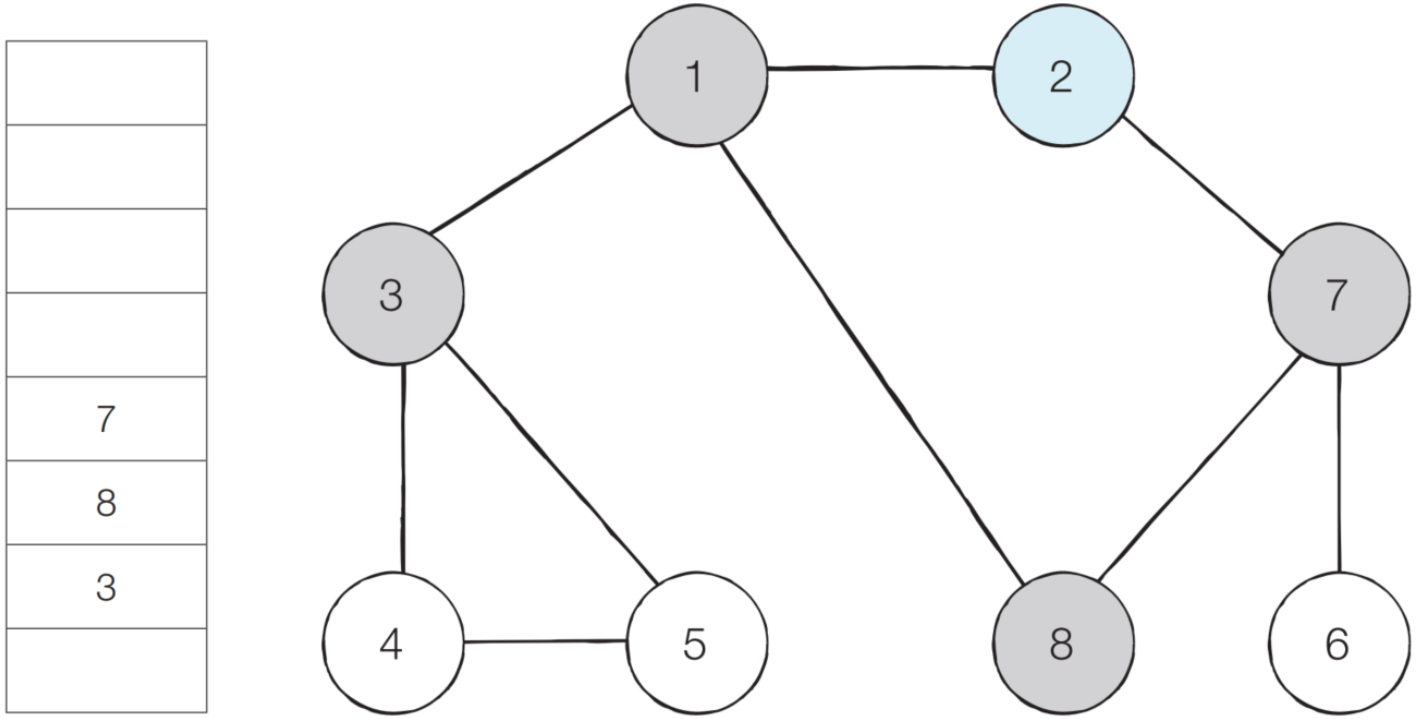
- [Step 1] 시작 노드인 '1'을 큐에 삽입하고 방문 처리를 합니다.



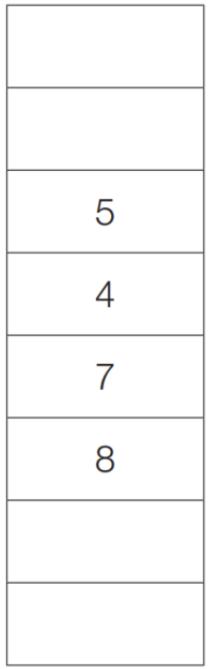
- [Step 2] 큐에서 노드 '1'을 꺼내 방문하지 않은 인접 노드 '2', '3', '8'을 큐에 삽입하고 방문 처리합니다.

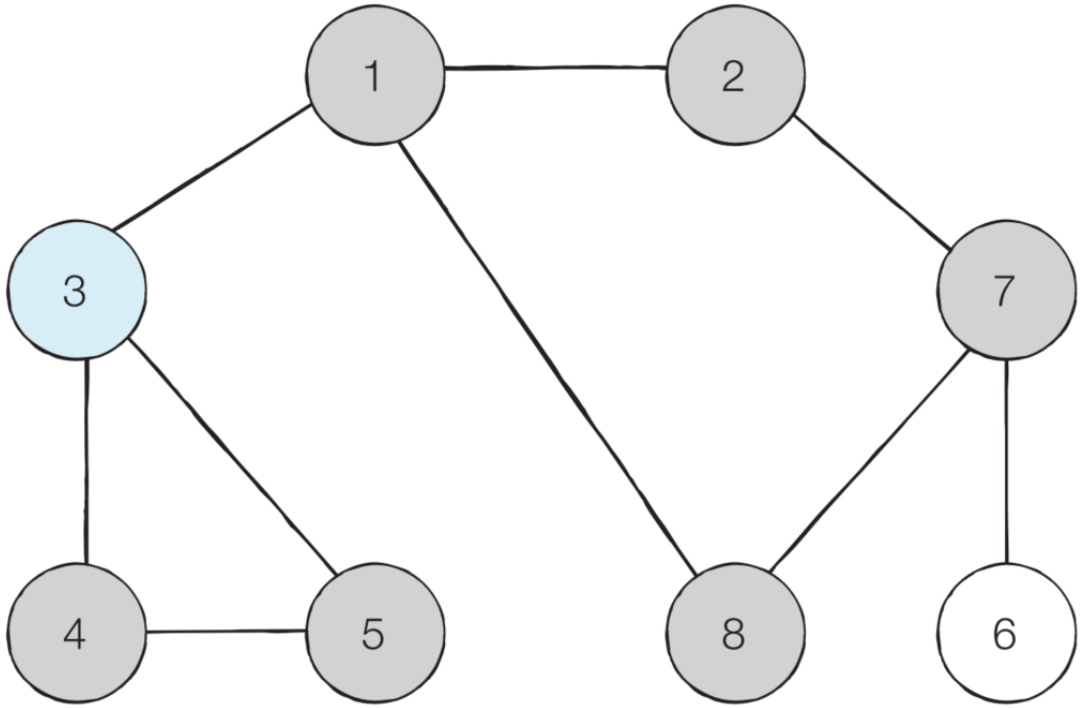


- [Step 3] 큐에서 노드 '2'를 꺼내 방문하지 않은 인접 노드 '7'을 큐에 삽입하고 방문 처리합니다.

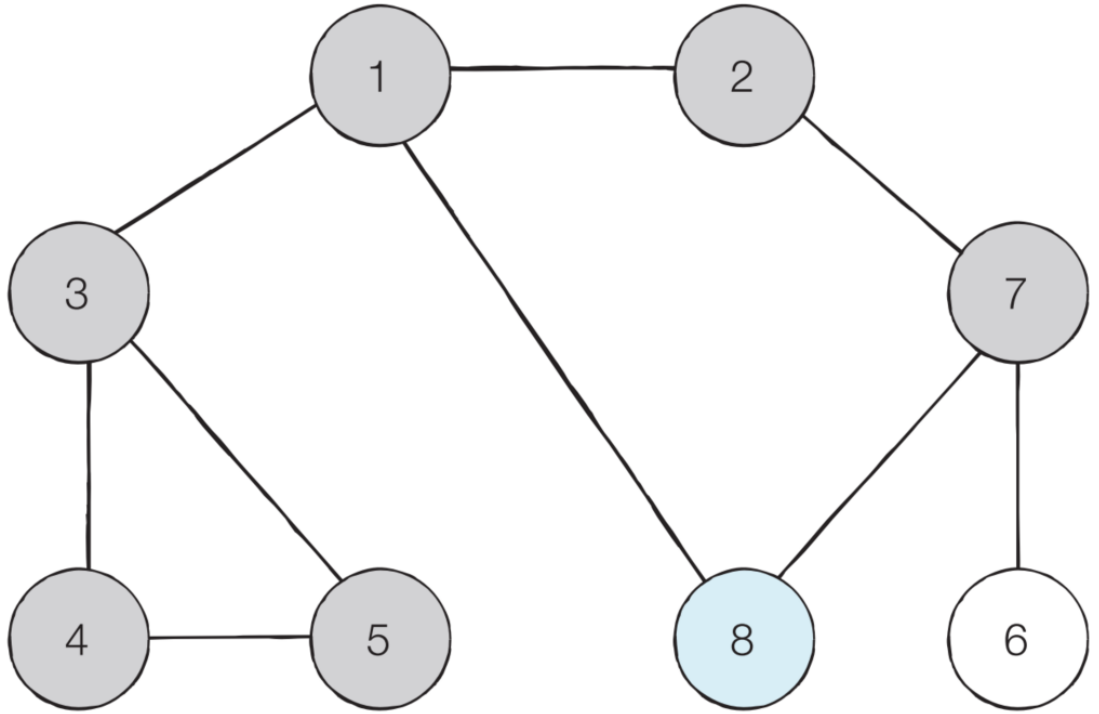


- [Step 4] 큐에서 노드 '3'을 꺼내 방문하지 않은 인접 노드 '4', '5'를 큐에 삽입하고 방문 처리합니다.



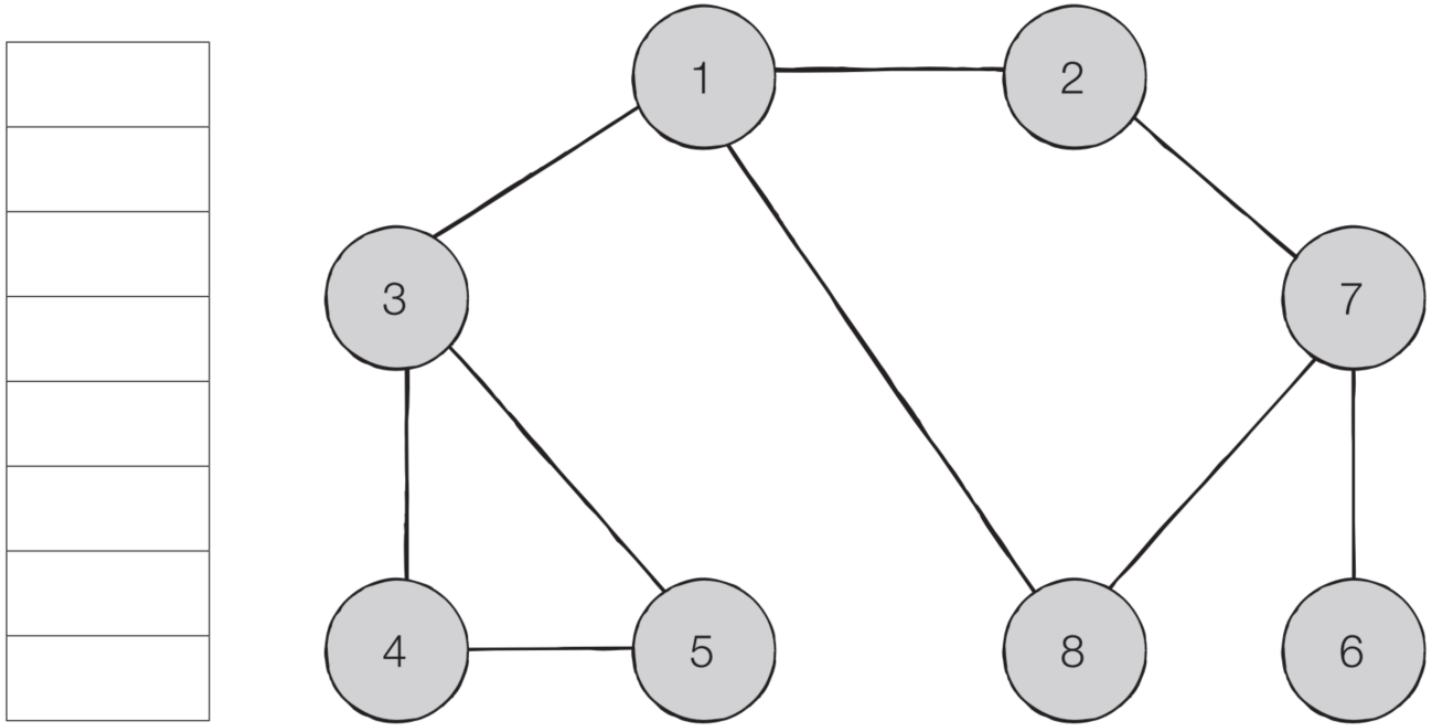


- [Step 5] 큐에서 노드 '8'을 꺼내고 방문하지 않은 인접 노드가 없으므로 무시합니다.



BFS 동작 예시

- 이러한 과정을 반복하여 전체 노드의 탐색 순서(큐에 들어간 순서)는 다음과 같습니다.



탐색 순서: 1 → 2 → 3 → 8 → 7 → 4 → 5 → 6

BFS 소스 코드 예제(Python)

```python
from collections import deque

# BFS 메서드 정의
def bfs(graph, start, visited):
    # 큐(Queue) 구현을 위해 deque 라이브러리 사용
    queue = deque([start])
    # 현재 노드를 방문 처리
    visited[start] = True
    # 큐가 빌 때까지 반복
    while queue:
        # 큐에서 하나의 원소를 뽑아 출력하기
        v = queue.popleft()
        print(v, end=' ')
        # 아직 방문하지 않은 인접한 원소들을 큐에 삽입
        for i in graph[v]:
            if not visited[i]:
                queue.append(i)
                visited[i] = True
```

```python
# 각 노드가 연결된 정보를 표현 (2차원 리스트)
graph = [
    [],
    [2, 3, 8],
    [1, 7],
    [1, 4, 5],
    [3, 5],
    [3, 4],
    [7],
    [2, 6, 8],
    [1, 7]
]

# 각 노드가 방문된 정보를 표현 (1차원 리스트)
visited = [False] * 9

# 정의된 BFS 함수 호출
bfs(graph, 1, visited)
```

```
1 2 3 8 7 4 5 6
```

BFS 소스 코드 예제(C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

bool visited[9];
int graph[9][3] = {{}, {2,3,8}, {1,7}, {1,4,5}, {3,5}, {3,4}, {7}, {2,6,8}, {1,7}};
void bfs(int start) {
    queue<int> q;
    q.push(start);
    visited[start] = true;
    while(!q.empty()) {
        int x = q.front();
        q.pop();
        cout << x << ' ';
        for(int node : graph[x]) {
            if(!visited[node]) {
                q.push(node);
                visited[node] = true;
            }
        }
    }
}

int main(void) {
    bfs(1);
    return 0;
}
```

BFS 소스 코드 예제(Java)

```java
import java.util.*;

public class Main {

    public static boolean[] visited = new boolean[9];
    public static int[][] graph = {{}, {2,3,8}, {1,7}, {1,4,5}, {3,5}, {3,4}, {7}, {2,6,8}, {1,7}};

    // BFS 함수 정의
    public static void bfs(int start) {
        Queue<Integer> q = new LinkedList<>();
        q.offer(start);
        // 현재 노드를 방문 처리
        visited[start] = true;
        // 큐가 빌 때까지 반복
        while(!q.isEmpty()) {
            // 큐에서 하나의 원소를 뽑아 출력
            int x = q.poll();
            System.out.print(x + " ");
            // 해당 원소와 연결된, 아직 방문하지 않은 원소들을 큐에 삽입
            for(int node : graph[x]) {
                if(!visited[node]) {
                    q.offer(node);
                    visited[node] = true;
                }
            }
        }
    }
    // 메인 함수 생략
}
```

<문제> 음료수 얼려 먹기: 문제 조건

난이도 ●◐○ | **풀이 시간** 30분 | **시간제한** 1초 | **메모리 제한** 128MB

입력 조건

- 첫 번째 줄에 얼음 틀의 세로 길이 N과 가로 길이 M이 주어집니다. (1 <= N, M <= 1,000)
- 두 번째 줄부터 N + 1번째 줄까지 얼음 틀의 형태가 주어집니다.
- 이때 구멍이 뚫려있는 부분은 0, 그렇지 않은 부분은 1입니다.

출력 조건

- 한 번에 만들 수 있는 아이스크림의 개수를 출력합니다.

입력 예시

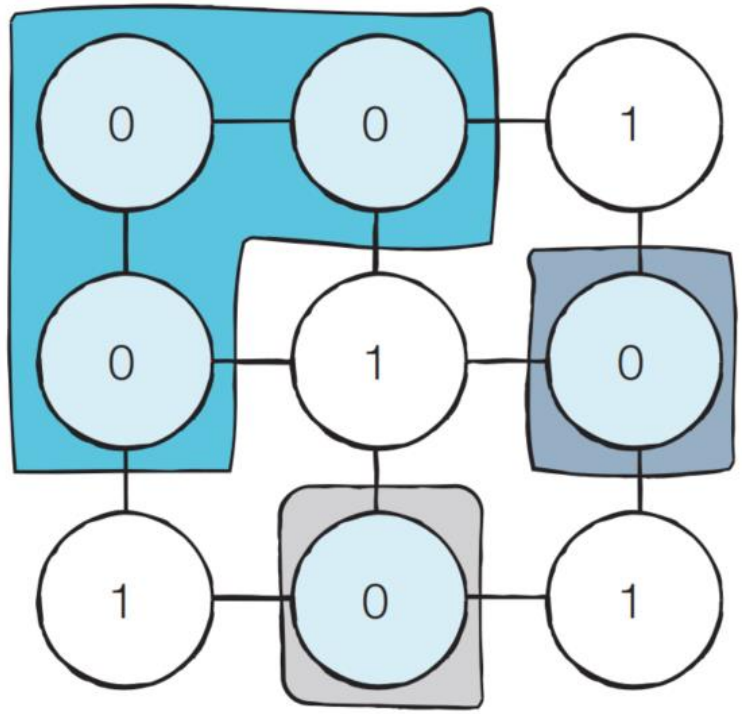
```
4 5
00110
00011
11111
00000
```

출력 예시

```
3
```

<문제> 음료수 얼려 먹기: 문제 해결 아이디어

- 이 문제는 DFS 혹은 BFS로 해결할 수 있습니다. 일단 앞에서 배운 대로 얼음을 얼릴 수 있는 공간이 상, 하, 좌, 우로 연결되어 있다고 표현할 수 있으므로 그래프 형태로 모델링 할 수 있습니다. 다음과 같이 3 × 3 크기의 얼음 틀이 있다고 가정하고 생각해 봅시다.



<문제> 음료수 얼려 먹기: 문제 해결 아이디어

- DFS를 활용하는 알고리즘은 다음과 같습니다.
  1. 특정한 지점의 주변 상, 하, 좌, 우를 살펴본 뒤에 주변 지점 중에서 값이 '0'이면서 아직 방문하지 않은 지점이 있다면 해당 지점을 방문합니다.
  2. 방문한 지점에서 다시 상, 하, 좌, 우를 살펴보면서 방문을 진행하는 과정을 반복하면, 연결된 모든 지점을 방문할 수 있습니다.
  3. 모든 노드에 대하여 1 ~ 2번의 과정을 반복하며, 방문하지 않은 지점의 수를 카운트합니다.

<문제> 음료수 얼려 먹기: 답안 예시(Python)

<문제> 음료수 얼려 먹기: 답안 예시(C++)

<문제> 음료수 얼려 먹기: 답안 예시(Java)

<문제> DFS와 BFS 프로그램

출처:백준온라인저지1260번

1260번 | 제출 | 맞힌 사람 | 숏코딩 | 재채점 결과 | 채점 현황 | 강의 | 질문 검색

DFS와 BFS

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
2 초	128 MB	202128	74417	44172	35.845%

문제

그래프를 DFS로 탐색한 결과와 BFS로 탐색한 결과를 출력하는 프로그램을 작성하시오. 단, 방문할 수 있는 정점이 여러 개인 경우에는 정점 번호가 작은 것을 먼저 방문하고, 더 이상 방문할 수 있는 점이 없는 경우 종료한다. 정점 번호는 1번부터 N번까지이다.

입력

첫째 줄에 정점의 개수 N(1 ≤ N ≤ 1,000), 간선의 개수 M(1 ≤ M ≤ 10,000), 탐색을 시작할 정점의 번호 V가 주어진다. 다음 M개의 줄에는 간선이 연결하는 두 정점의 번호가 주어진다. 어떤 두 정점 사이에 여러 개의 간선이 있을 수 있다. 입력으로 주어지는 간선은 양방향이다.

출력

첫째 줄에 DFS를 수행한 결과를, 그 다음 줄에는 BFS를 수행한 결과를 출력한다. V부터 방문된 점을 순서대로 출력하면 된다.

예제 입력 1 복사

```
4 5 1
1 2
1 3
1 4
2 4
3 4
```

예제 출력 1 복사

```
1 2 4 3
1 2 3 4
```