JavaScript
BFS 알고리즘
BFS 알고리즘 이해하기

BFS 알고리즘 이해하기 | 코딩 테스트에서 자주 등장하는 BFS 알고리즘 이해하기

강사 나동빈

JavaScript
BFS 알고리즘

BFS 알고리즘 이해하기

DFS와 BFS

- 탐색(Search)이란 많은 양의 데이터 중에서 **원하는 데이터를 찾는 과정**을 의미한다.
- 대표적인 그래프 탐색 알고리즘으로는 DFS와 BFS가 있다.
- **DFS와 BFS**는 코딩 테스트에서 매우 자주 등장하는 유형이므로 반드시 숙지할 필요가 있다.

JavaScript 큐(Queue) 구현하기


```
class Queue {
  constructor() {
    this.items = {};
    this.headIndex = 0;
    this.tailIndex = 0;
  }
  enqueue(item) {
    this.items[this.tailIndex] = item;
    this.tailIndex++;
  }
  dequeue() {
    const item = this.items[this.headIndex];
    delete this.items[this.headIndex];
    this.headIndex++;
    return item;
  }
  peek() {
    return this.items[this.headIndex];
  }
  getLength() {
    return this.tailIndex - this.headIndex;
  }
}
```

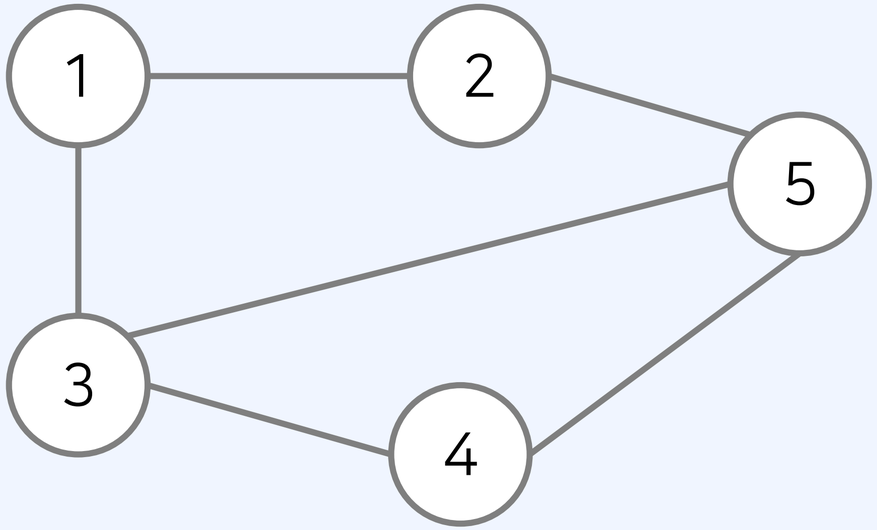
```
// 구현된 큐(Queue) 라이브러리 사용
queue = new Queue();

// 삽입(5) - 삽입(2) - 삽입(3) - 삽입(7)
// - 삭제() - 삽입(1) - 삽입(4) - 삭제()
queue.enqueue(5);
queue.enqueue(2);
queue.enqueue(3);
queue.enqueue(7);
queue.dequeue();
queue.enqueue(1);
queue.enqueue(4);
queue.dequeue();

// 먼저 들어온 순서대로 출력
while (queue.getLength() != 0) {
  console.log(queue.dequeue());
}
```

[실행 결과]

```
3
7
1
4
```

그래프의 표현

- 일반적으로 JavaScript로 DFS와 같은 그래프 문제를 해결할 때는?
- 2차원 배열(리스트)로 그래프를 표현한다.
- 인접 리스트 표현 방식의 예시는 다음과 같다.



인접 리스트

1	2, 3
2	1, 5
3	1, 4, 5
4	3, 5
5	2, 3, 4

너비 우선 탐색(BFS)이란?

- 그래프 혹은 트리에서 모든 노드를 한 번씩 탐색하기 위한 기본적인 방법이다.
- **[완전 탐색]**을 수행하기 위해 사용할 수 있는 방법 중 하나다.
- (모든 간선의 길이가 동일할 때) 최단 거리를 탐색하기 위한 목적으로 사용할 수 있다.
- **큐(queue) 자료구조**를 사용한다.

→ 기본적으로 DFS는 스택, BFS는 큐를 사용한다.

너비 우선 탐색(BFS) 기본 동작 방식

- BFS는 다음과 같은 방법으로 동작한다.

1. **시작 노드**를 큐에 넣고 **[방문 처리]**한다.
2. 큐에서 원소를 꺼내어 방문하지 않은 인접 노드가 있는지 확인한다.

- 있다면, 방문하지 않은 인접 노드를 큐에 **모두 삽입**하고 **[방문 처리]**한다.

3. 2번 과정을 더 이상 반복할 수 없을 때까지 반복한다.

너비 우선 탐색(BFS) 사용 예시

- 너비 우선 탐색이 사용되는 예시로는 다음과 같은 상황들이 있다.

1. 간선의 비용이 동일한 상황에서 **[최단 거리]** 문제를 해결하는 경우
2. 완전 탐색을 위해 사용한 DFS 솔루션이 메모리/시간 초과를 받아 BFS로 재시도하는 경우

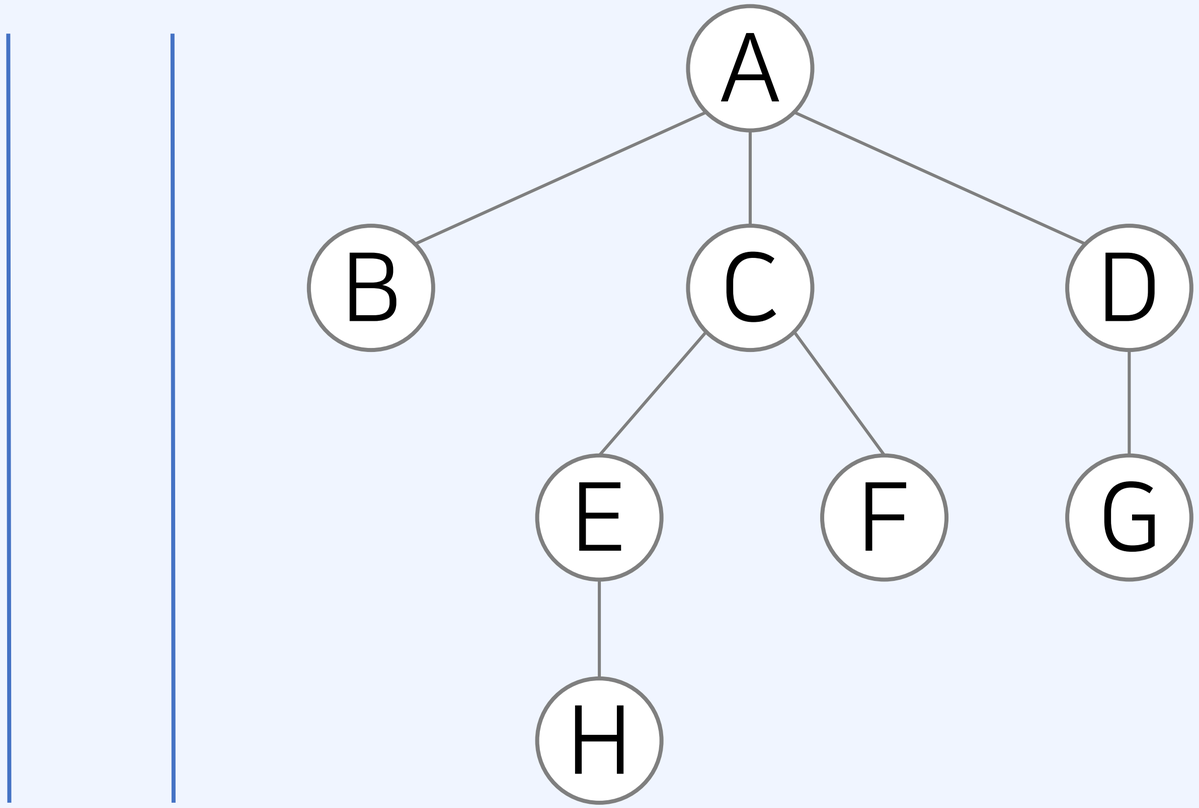
→ DFS와 BFS 모두 그래프 탐색 목적으로 사용할 수 있으나, 구현이 익숙하다면 BFS를 추천한다.

- 코딩 테스트에서 DFS로 해결할 수 있는 문제는 BFS로도 해결할 수 있는 경우가 많다.

→ DFS는 일반적인 최단 거리 문제를 해결할 수 없다.

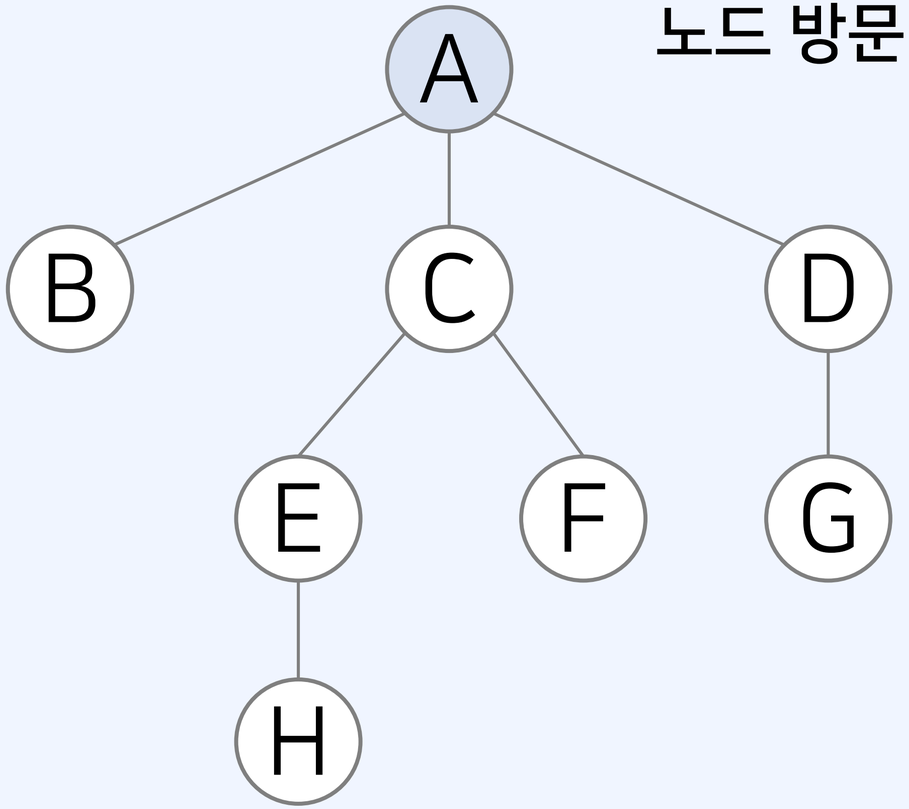
너비 우선 탐색(BFS) 기본 동작 방식

- BFS는 다음과 같은 방법으로 동작한다.



너비 우선 탐색(BFS) 기본 동작 방식

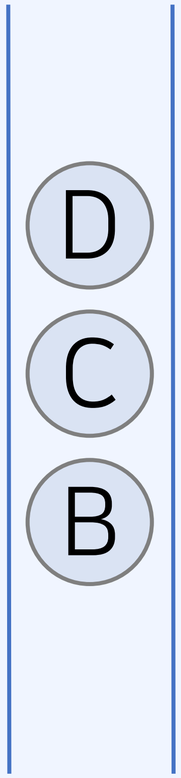
- **시작 노드**를 큐에 넣고 **[방문 처리]**한다.

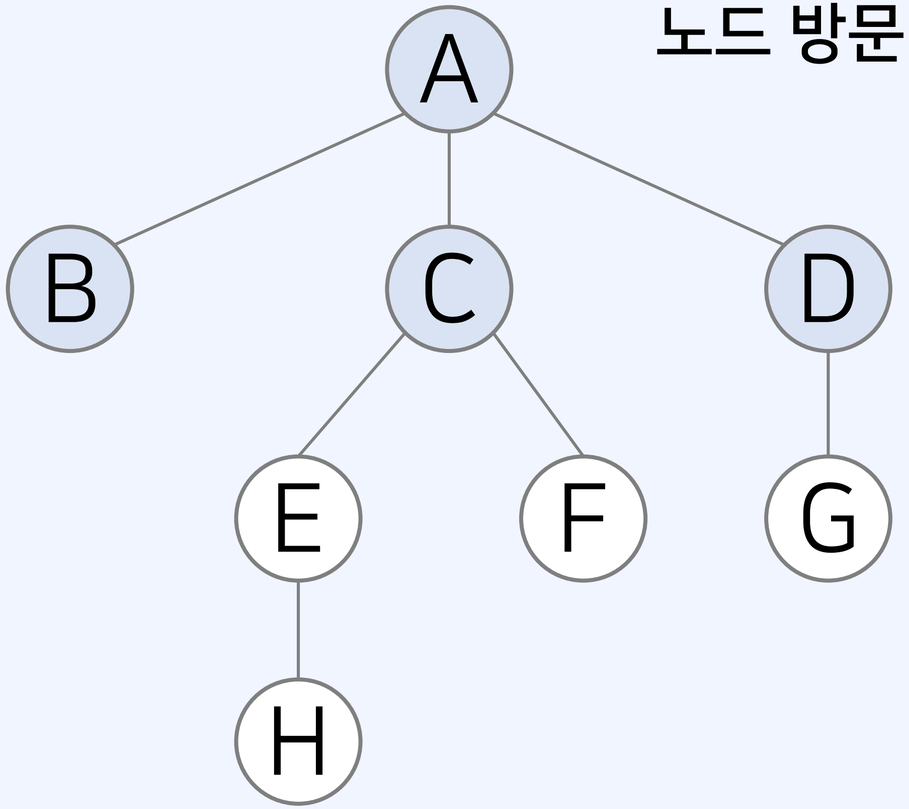


노드 방문 순서: A

너비 우선 탐색(BFS) 기본 동작 방식

- 큐에서 A를 꺼내어 인접한 노드 중에서 방문하지 않은 노드인 B, C, D를 방문한다.

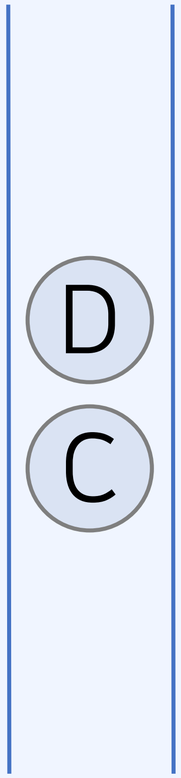


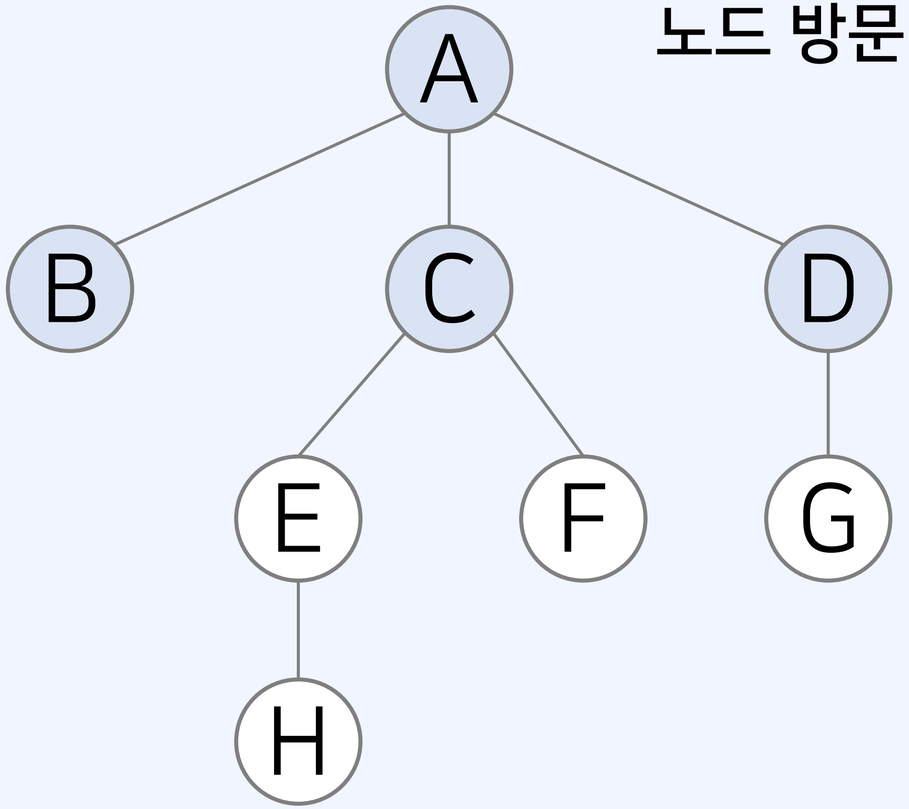


노드 방문 순서: A – B – C - D

너비 우선 탐색(BFS) 기본 동작 방식

- 큐에서 B를 꺼냈지만 인접한 노드 중에서 방문하지 않은 노드가 없으므로, 무시한다.

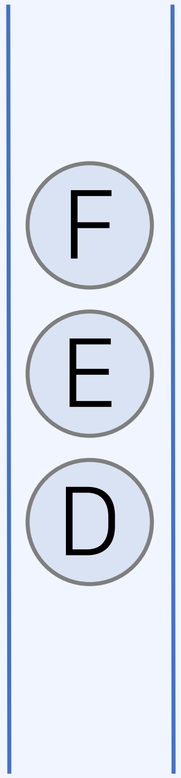


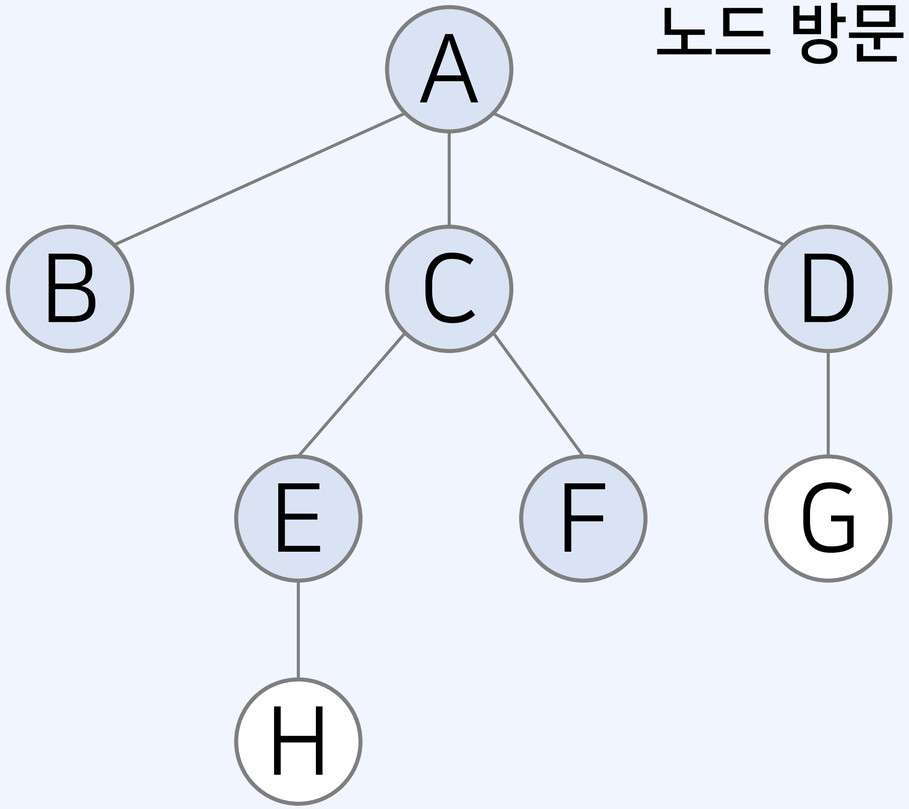


노드 방문 순서: A – B – C - D

너비 우선 탐색(BFS) 기본 동작 방식

- 큐에서 C를 꺼내어 인접한 노드 중에서 방문하지 않은 노드인 E, F를 방문한다.

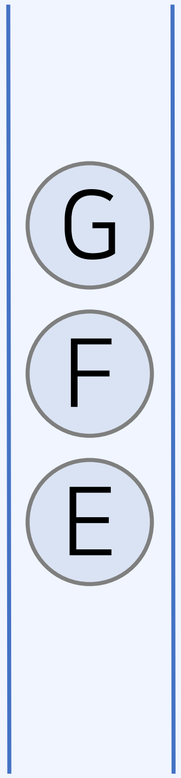


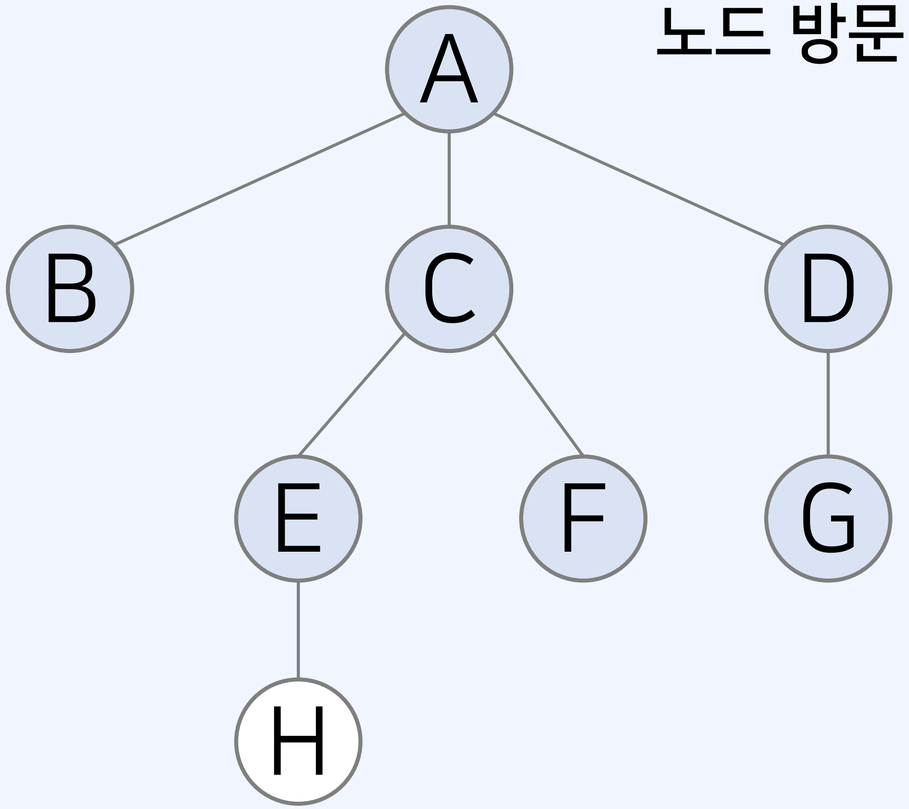


노드 방문 순서: A – B – C – D – E - F

너비 우선 탐색(BFS) 기본 동작 방식

- 큐에서 D를 꺼내어 인접한 노드 중에서 방문하지 않은 노드인 G를 방문한다.

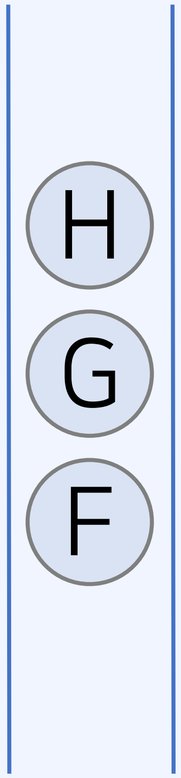


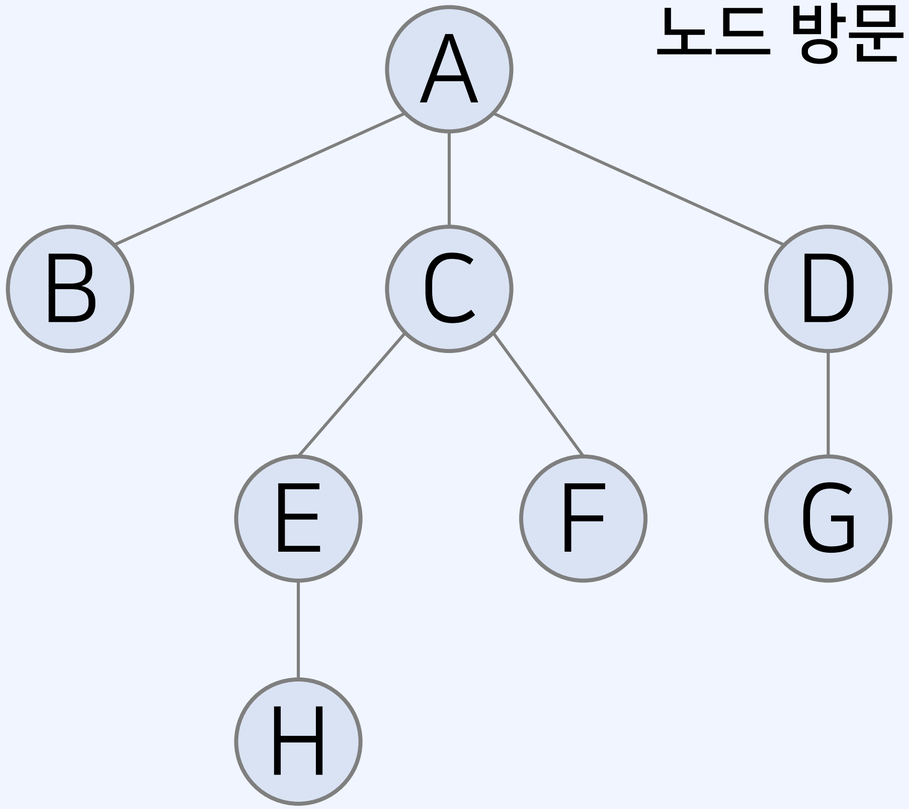


노드 방문 순서: A - B - C - D - E - F - G

너비 우선 탐색(BFS) 기본 동작 방식

- 큐에서 E를 꺼내어 인접한 노드 중에서 방문하지 않은 노드인 H를 방문한다.

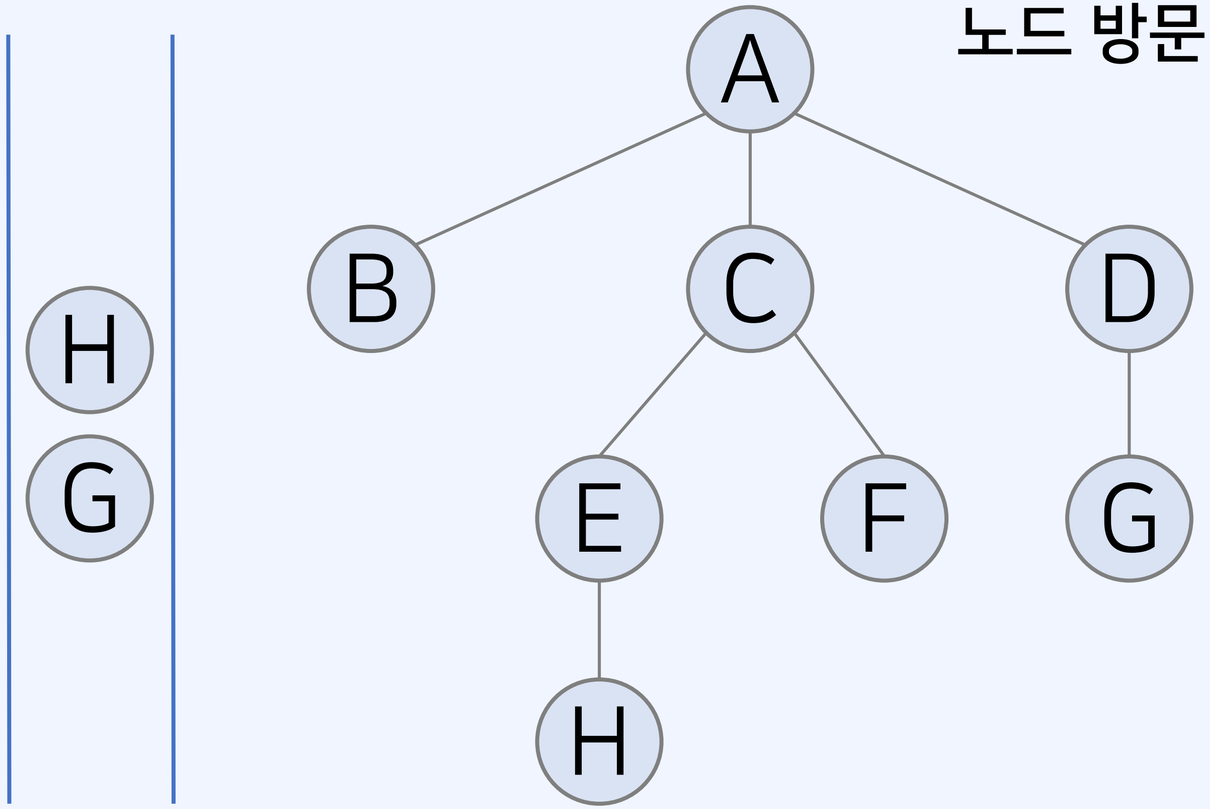




노드 방문 순서: A - B - C - D - E - F - G - H

너비 우선 탐색(BFS) 기본 동작 방식

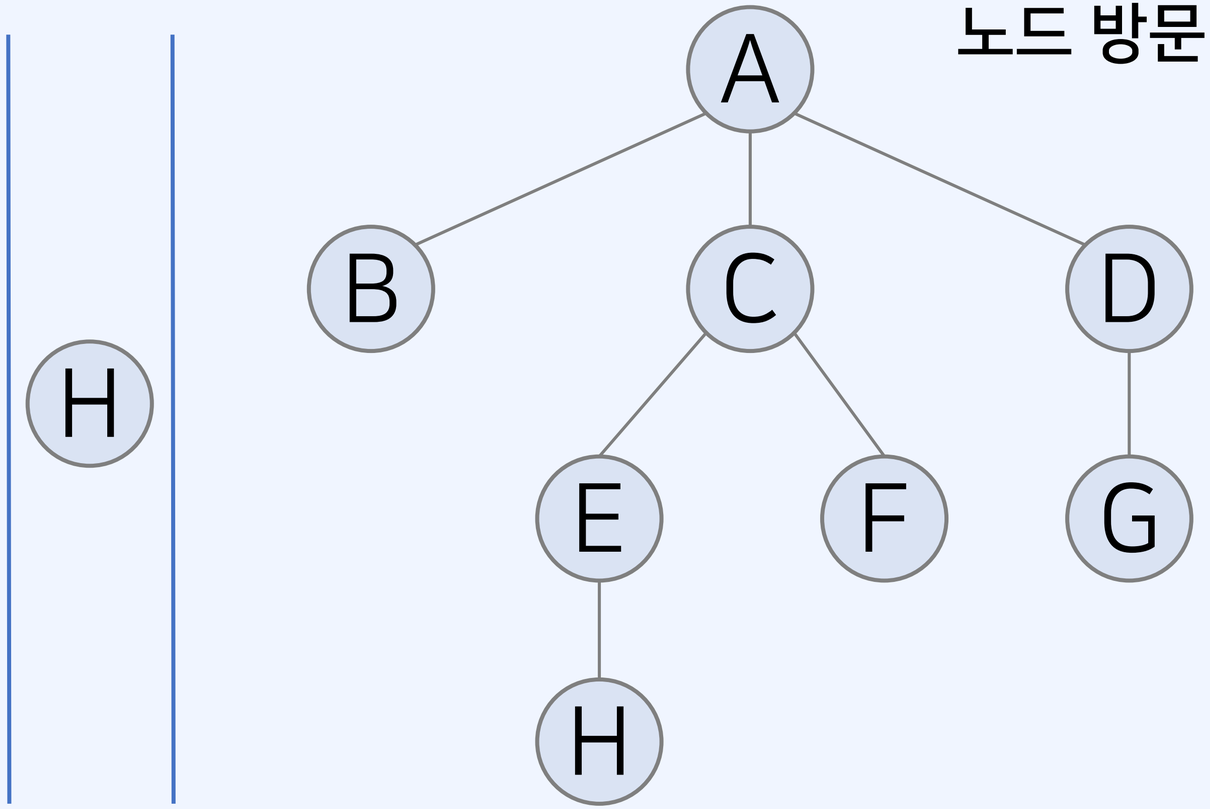
- 큐에서 F를 꺼냈지만 인접한 노드 중에서 방문하지 않은 노드가 없으므로, 무시한다.



노드 방문 순서: A - B - C - D - E - F - G - H

너비 우선 탐색(BFS) 기본 동작 방식

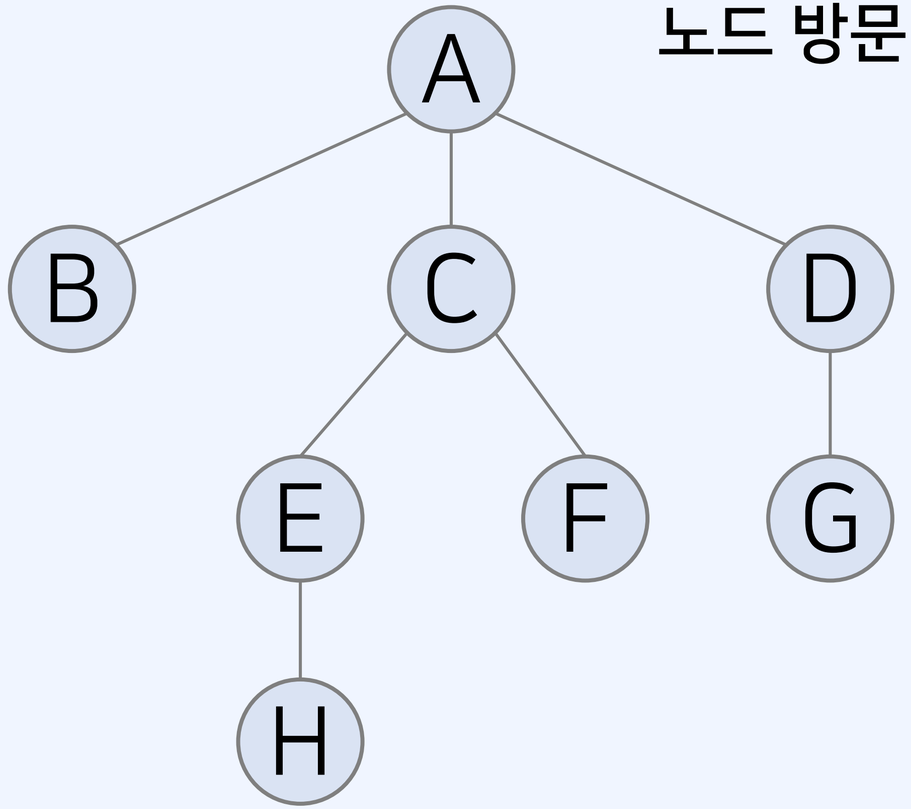
- 큐에서 G를 꺼냈지만 인접한 노드 중에서 방문하지 않은 노드가 없으므로, 무시한다.



노드 방문 순서: A - B - C - D - E - F - G - H

너비 우선 탐색(BFS) 기본 동작 방식

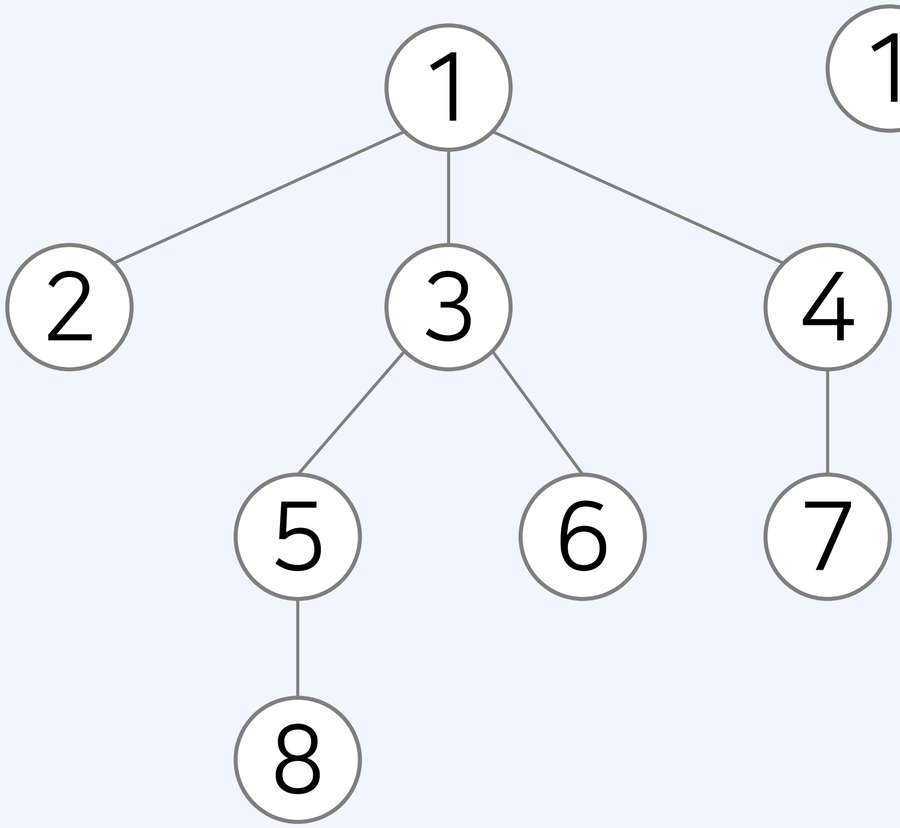
- 큐에서 H를 꺼냈지만 인접한 노드 중에서 방문하지 않은 노드가 없으므로, 무시한다.

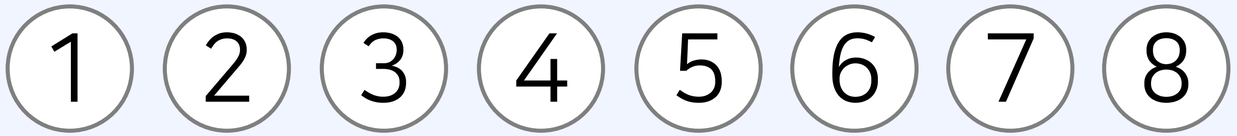


노드 방문 순서: A - B - C - D - E - F - G - H

너비 우선 탐색(BFS) 기본 동작 방식

- 실제 알고리즘/코딩 테스트 문제에서는 노드의 번호가 1부터 시작하는 경우가 많다.





너비 우선 탐색(BFS)과 최단 경로

- BFS는 간선의 비용이 동일할 때 **[최단 거리]** 문제를 해결하기 위해 사용 가능하다.
- BFS는 다익스트라 최단 경로 알고리즘과 유사한 특징이 있다.

→ 다익스트라는 간선의 비용이 서로 다를 수 있을 때 사용 가능하다.

1) 다익스트라 알고리즘은 일반 큐 대신에 우선순위 큐를 사용한다.

2) 다익스트라는 특정 노드에 대하여 **[최단 거리]** 값이 갱신될 수 있다. (더 짧은 경로를 찾는 경우)

너비 우선 탐색(BFS) 소스 코드 예시


```
// BFS 메서드 정의
function bfs(graph, start, visited) {
  queue = new Queue();
  queue.enqueue(start);
  // 현재 노드를 방문 처리
  visited[start] = true;
  // 큐가 빌 때까지 반복
  while (queue.getLength() != 0) {
    // 큐에서 하나의 원소를 뽑아 출력하기
    v = queue.dequeue();
    console.log(v);
    // 아직 방문하지 않은 인접한 원소들을 큐에 삽입
    for (i of graph[v]) {
      if (!visited[i]) {
        queue.enqueue(i);
        visited[i] = true;
      }
    }
  }
}
```

```
// 각 노드가 연결된 정보를 표현
graph = [
  [],
  [2, 3, 4],
  [1],
  [1, 5, 6],
  [1, 7],
  [3, 8],
  [3],
  [4],
  [5]
];

// 각 노드가 방문된 정보를 표현
visited = Array(9).fill(false);

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