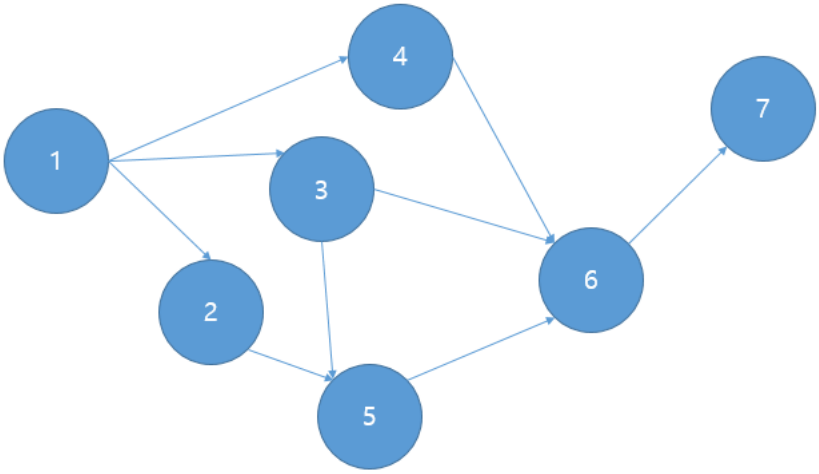
너비 우선 탐색(BFS)

'너비 우선 탐색(BFS: Breadth First Search)'은 **그래프를 방문하거나 탐색하는 방법의 하나**이다. BFS 를 이용하여 **최단 거리, 최소비용과 같이 최솟값과 관련된 문제**를 해결할 수 있는데, 이때 **그래프의 가중치(시간, 비용, 거리 등)가 1** 이어야만 한다. 다음 강의에서 다룰 깊이 우선 탐색(DFS: Depth First Search)은 그래프의 최대 경로(최대 깊이)가 예측할 수 있나 유한한 범위여야만 사용할 수 있다. 하지만, BFS 는 모든 경로에 대한 동시 탐색이 가능하여 최대 경로를 몰라도 사용할 수 있다는 장점이 있고, 이 장점으로 인해 최단 거리, 최소비용 등을 구할 수 있다.

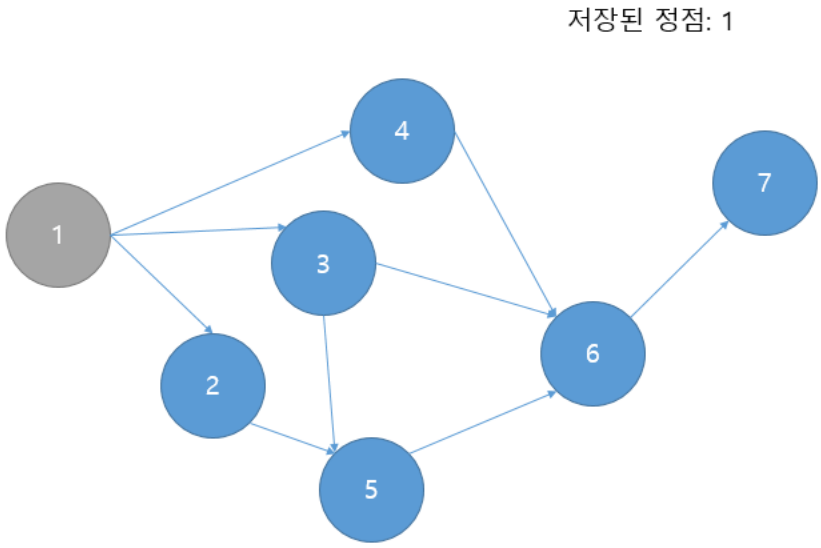
BFS 는 다음과 같은 절차로 탐색을 진행한다.

1. 저장된 정점 중 첫 번째 정점을 선택하여 저장된 정점에서 제거
2. 제거한 정점에서 해야 할 작업 진행
3. 제거한 정점과 연결된 정점 중 방문하지 않은 모든 정점 저장 (2 번과 3 번은 순서가 바뀌어도 상관없음)
4. 저장된 정점에 모든 노드가 제거될 때까지 1~3 번 과정을 반복

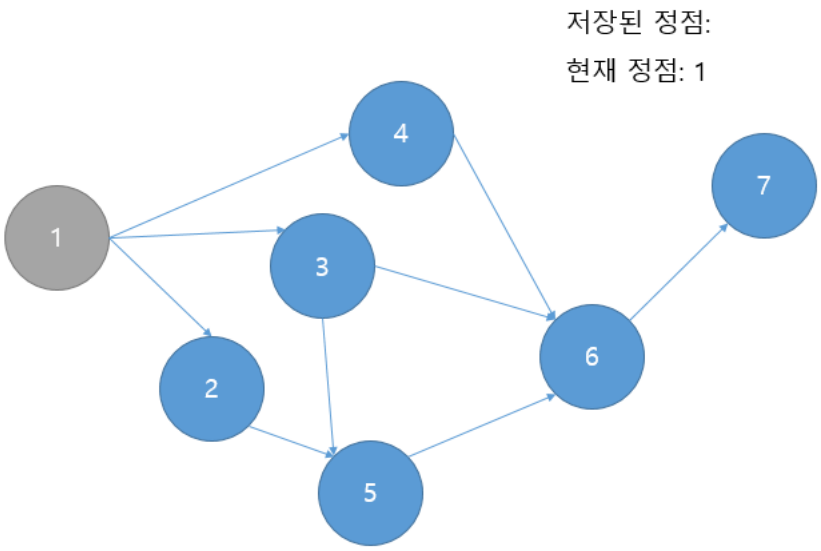
글만으로는 과정이 어떻게 진행되는 것인지 크게 와닿지 않을 수 있으니 그림을 통해 살펴보면,



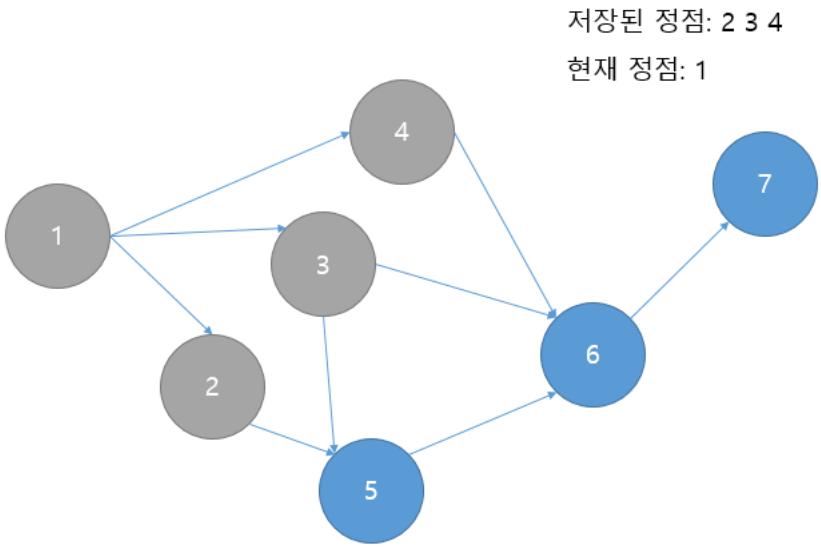
그래프가 위와 같이 정의되어 있을 때 1 번 정점부터 탐색을 시작할때,



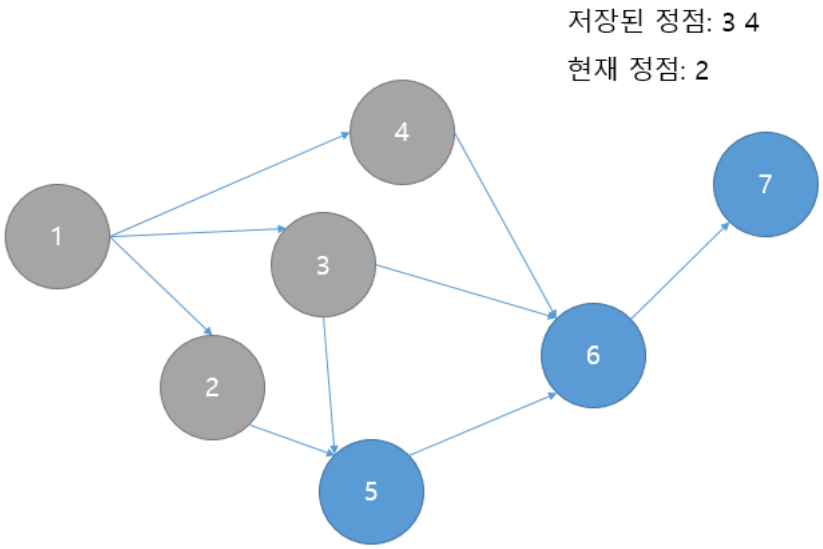
가장 먼저 시작 정점을 저장한 뒤 방문했다는 표시를 남겨야 한다. 그림에서는 방문했다는 표시를 회색 처리했다.



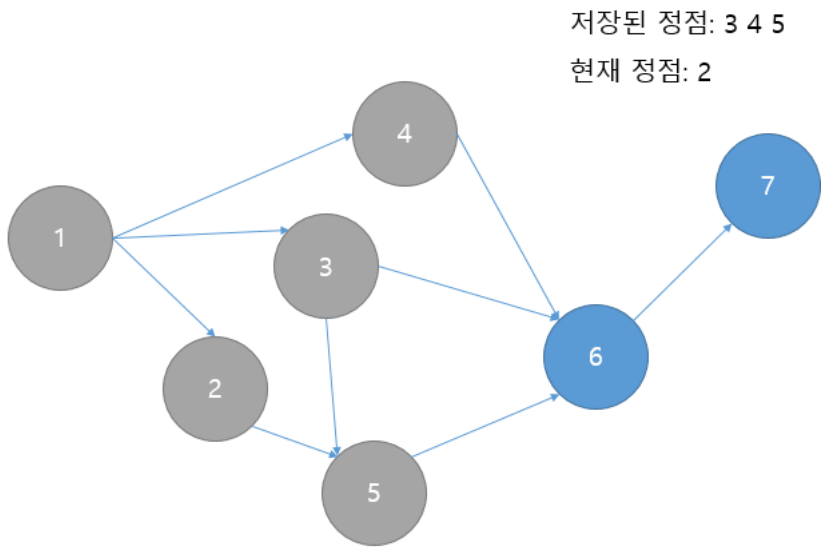
첫 번째 단계로 저장된 가장 첫 번째 정점인 1 을 제거했다.



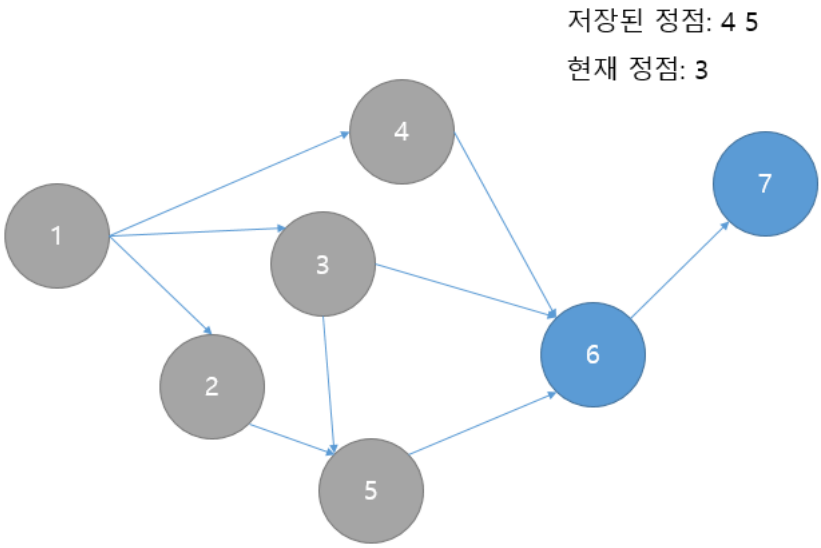
제거된 1 번 정점과 연결되어 있고 아직 방문하지 않은 모든 정점(2, 3, 4)을 저장한다. 방문하지 않은 정점을 저장할 때 반드시 방문했다는 표시를 함께 남겨준다.



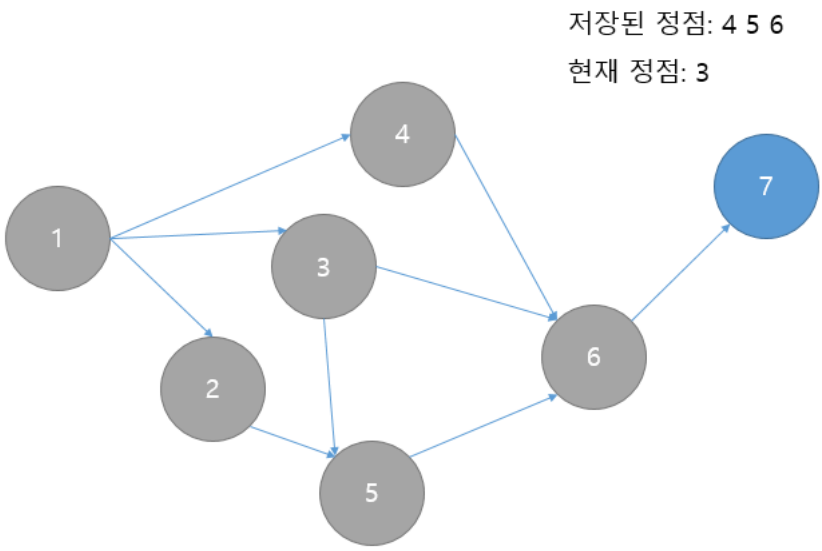
1 번 정점에 대한 모든 작업이 끝났으니 다시 첫 번째 과정으로 돌아갑니다. 저장된 정점 중 가장 첫 번째 정점은 2 번 정점이므로 2 번 정점을 제거한다.



이제 2 번 정점과 연결되어 있으며, 아직 방문하지 않은 모든 정점(5)을 저장한다. 이로써 2 번 정점에서의 모든 작업 또한 끝났으니 다시 첫 번째 과정으로 돌아가면,

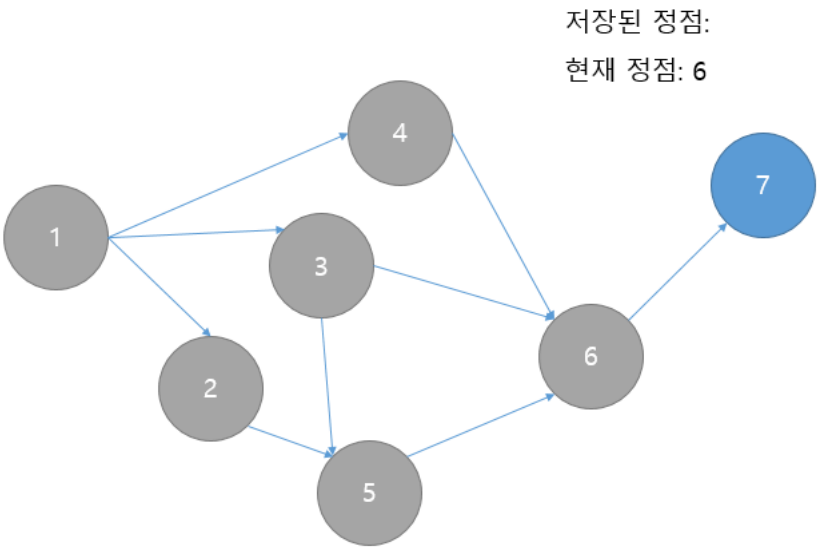


저장된 정점 중 가장 첫 번째 정점은 3 번 정점이므로 3 번 정점을 제거한다.

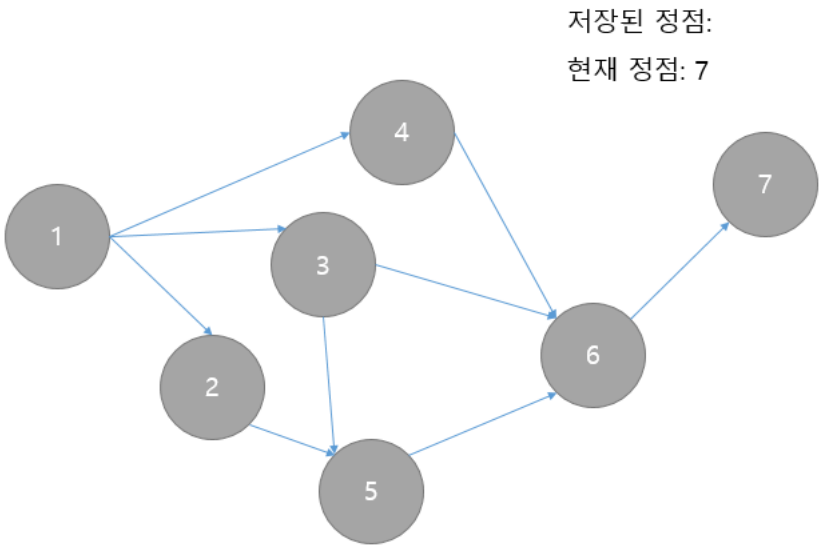


3 번 정점과 연결되어 있고 아직 방문하지 않은 6 번 정점을 저장한다. 과정 설명 초반에 '연결된 정점을 저장할 때 방문 표시를 함께 한다'!! 명심, 바로 지금 단계 같은 경우 때문이다. 만약 5 번 정점에 방문 표시가 남아있지 않았다면 이 단계에서 5 번 정점이 다시 저장되어 필요 없는 작업이 반복된다.

자, 이제 3 번 정점에 대한 작업도 모두 끝났다. 4 번 정점과 5 번 정점에 대해 작업을 진행해야 하지만 두 정점과 연결된 모든 정점이 모두 방문한 상태이니 넘어가면,.

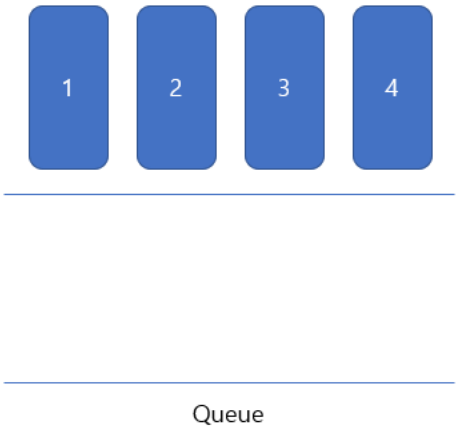


6 번 정점 차례이다. 6 번 정점과 연결되어 있으며 방문하지 않은 정점인 7 을 저장한 뒤 첫 번째 과정으로 돌아가 7 번 정점을 선택하고 저장된 정점에서 삭제한다.



이제 7 번 정점까지 방문했다면 그래프에 있는 모든 정점을 방문했으므로 BFS 가 종료됩니다. 과정이 꽤 길어 도대체 이걸 어떻게 구현해야 하는지 막막하실 수 있지만, 지금부터 구현 방법을 설명해 드리니 천천히 읽어보시기 바란다..

예시에서 **정점을 저장하고 저장된 정점 중 가장 먼저 저장된 정점을 선택하며 해당 정점에 대한 작업**까지의 과정이 반복된다. 이는 큐 자료구조를 사용하여 구현할 수 있다. **큐(Queue)는 FIFO(First In First Out) 구조**로, 위 설명과 같이 **가장 먼저 들어온 데이터를 먼저 처리하는 구조**를 갖는다. 가장 나중에 들어온 데이터가 가장 먼저 처리되는 스택(Stack)과는 차이가 있죠? 큐는 일상생활에서 영화관에 입장하기 위해 줄을 서 있는 사람들을 예로 들 수 있다. 가장 먼저 줄을 선 사람이 가장 먼저 입장하기 때문이죠. 그렇다면 큐를 이용하여 어떻게 데이터를 넣고 뺄 수 있는지 그림을 통해 살펴볼까요?

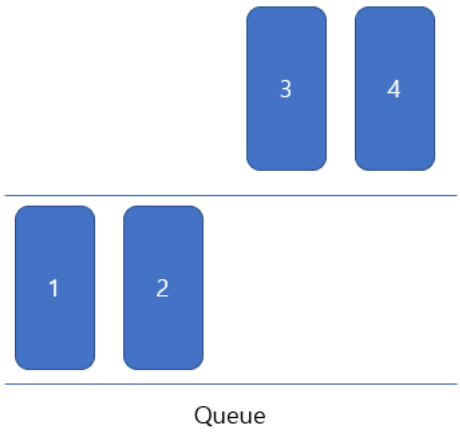


위 그림의 {1, 2, 3, 4}라는 데이터를 통해 큐에 데이터를 넣고 빼보겠습니다. **큐에 데이터를 넣을 때는 push 함수를 사용**하며, C++를 기준으로 다음과 같이 큐를 선언한다.

```
queue<int> q;
```

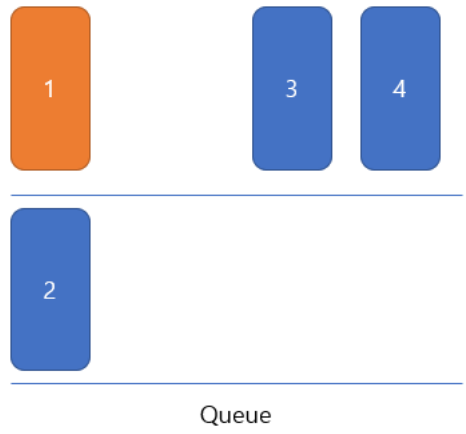
1, 2 데이터를 큐에 넣으려면 다음과 같이 push 함수를 사용한다.

```
q.push(1);
q.push(2);
```

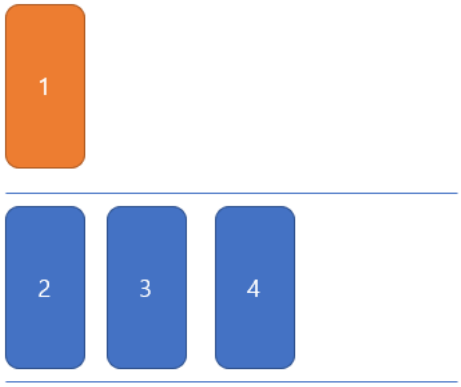


두 번의 push 연산이 끝나면 1, 2 는 위 그림처럼 큐에 입력된다. **큐에 넣은 데이터를 꺼내려면 pop 함수를 사용**한다. 현재 상태에서 pop 연산을 한 번 실행하면 다음 그림과 같은 결과가 나타난다.

```
q.pop();
```



가장 앞에 위치한, 즉 가장 먼저 들어온 데이터인 1 이 큐 바깥으로 빠져나온 것을 볼 수 있다. 이번에는 3, 4 를 큐에 넣으면 다음과 같다.



Queue

2, 3, 4 데이터가 순서대로 큐에 입력된다. 이제 데이터를 모두 순서대로 꺼내보면,

Queue

1, 2, 3, 4 데이터가 모두 순차적으로 나온 것을 확인할 수 있다. 아래의 코드는 C++를 이용하여 구현한 BFS 예제 코드이다.

```swift
func bfs(_ start: Int) {
    var check = [Bool](repeating: false, count: 1010)
    var queue = [Int]()

    queue.append(start) // 시작점을 큐에 저장
    check[start] = true

    while queue.count > 0 {
        var cur = queue.first! // 큐에 저장되어 있는 정점 중 가장 먼저 저장된 정점을 선택
        queue.removeFirst() // 선택후 제거

        for i in 0..<arr[cur].count {
            var next = arr[cur][i]
            if !check[next] {   // 만약 해당 정점을 방문하지 않았다면
                check[next] = true  // 해당 정점에 방문했다고 표시를 해준 후,
                queue.append(next)  // 큐에 해당 정점을 저장
            }
        }
    }
}
```