2021 신촌 연합 여름캠프 초급반 11회차

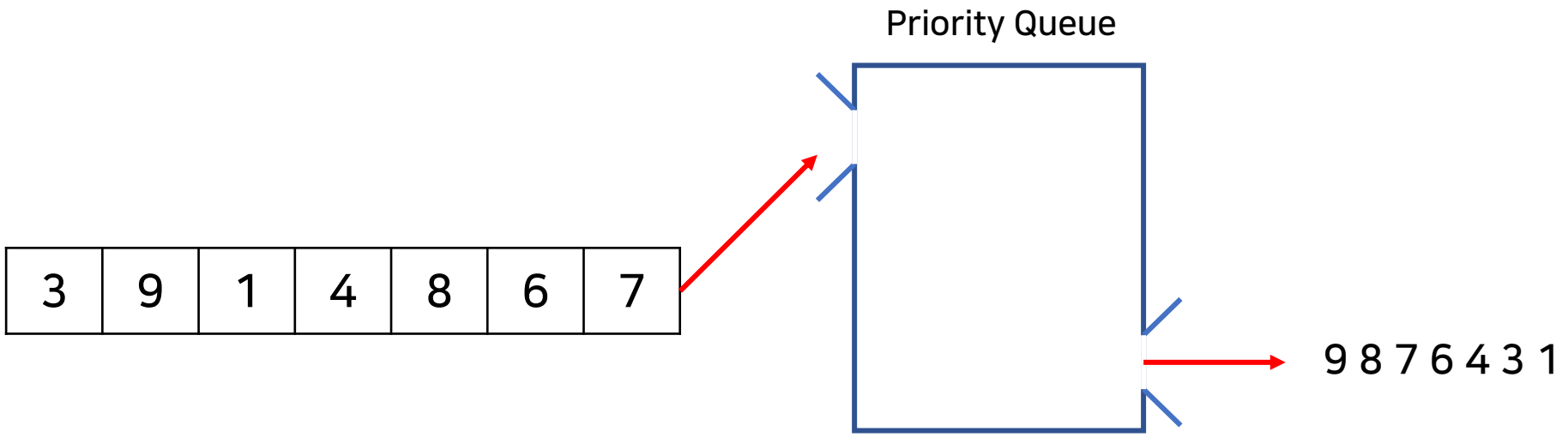
우선순위 큐 & 다익스트라

서강대학교 박재형

우선순위 큐 (Priority Queue)

- 우선순위가 가장 높은 원소부터 pop되는 큐

ex) 원소의 값이 클수록 우선순위가 높다



우선순위 큐 (Priority Queue)

- 배열(Array)로 구현

1) 삽입 : O(N) / 삭제 : O(1)

2) 삽입 : O(1) / 삭제 : O(N)

→ 힙(Heap)으로 구현

힙 (Heap)

- 완전 이진 트리로 이루어진 자료구조

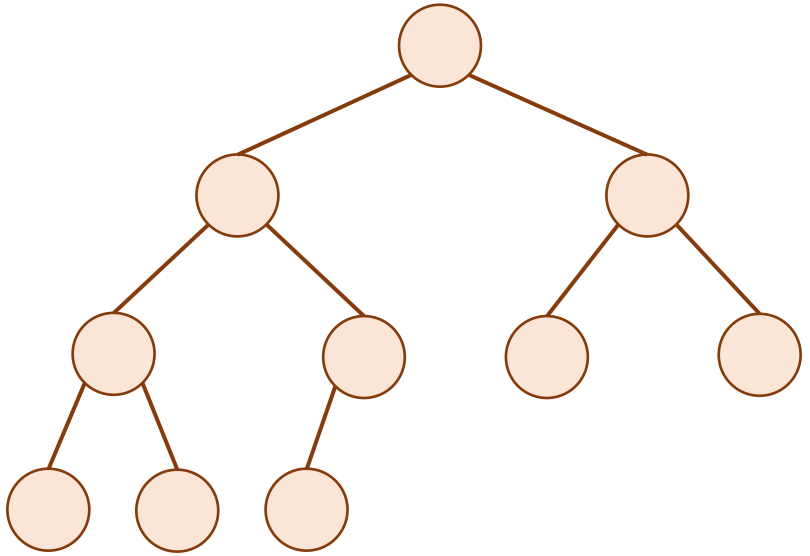
- 최솟값 or 최댓값을 빠르게 구하기 위한 자료구조

❖ Review) 완전 이진 트리?

- 모든 노드의 자식이 최대 2개

- 마지막 레벨을 제외한 모든 노드의 자식이 꽉 채워진 형태

- 마지막 레벨의 노드는 최대한 왼쪽으로



힙 (Heap)

- 최대 힙(Max Heap) : 부모 노드의 값이 자식 노드의 값보다 항상 크거나 같은 힙

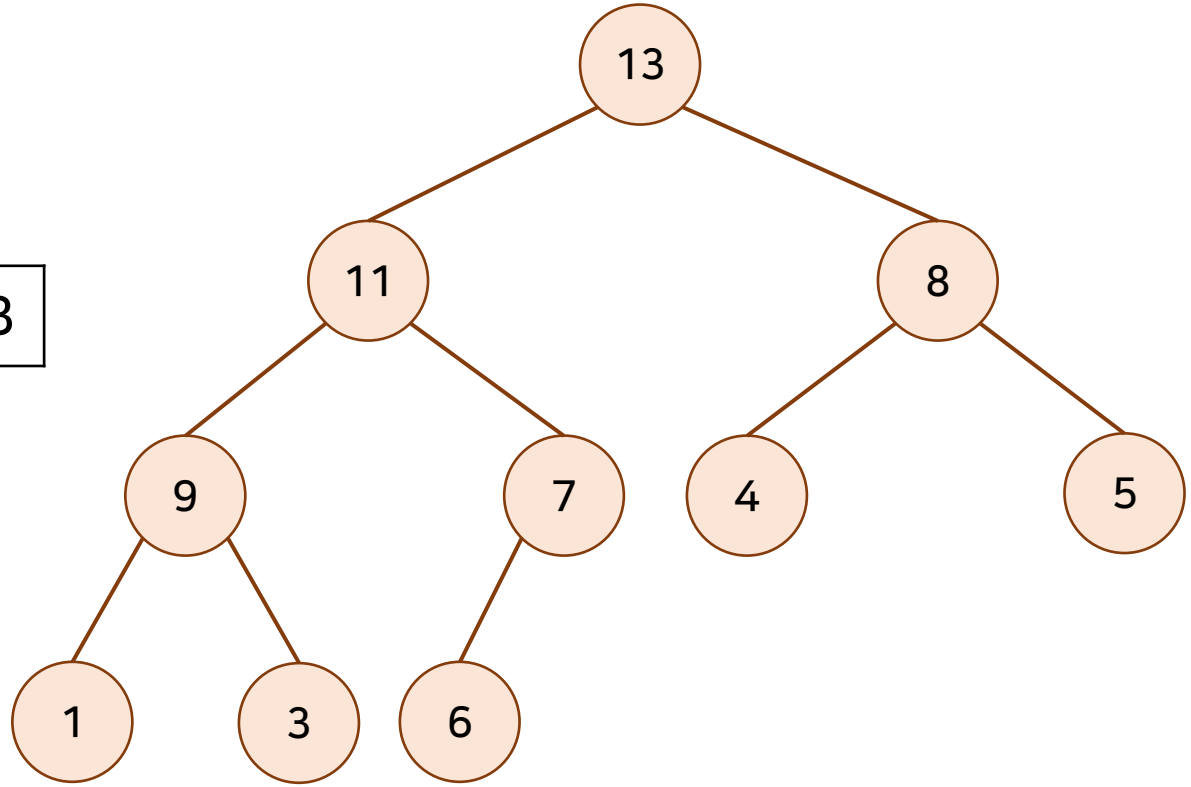
- 최소 힙(Max Heap) : 부모 노드의 값이 자식 노드의 값보다 항상 작거나 같은 힙

- 느슨한 정렬상태(반정렬 상태) 유지

힙 (Heap)

ex) 최대 힙(Max Heap)

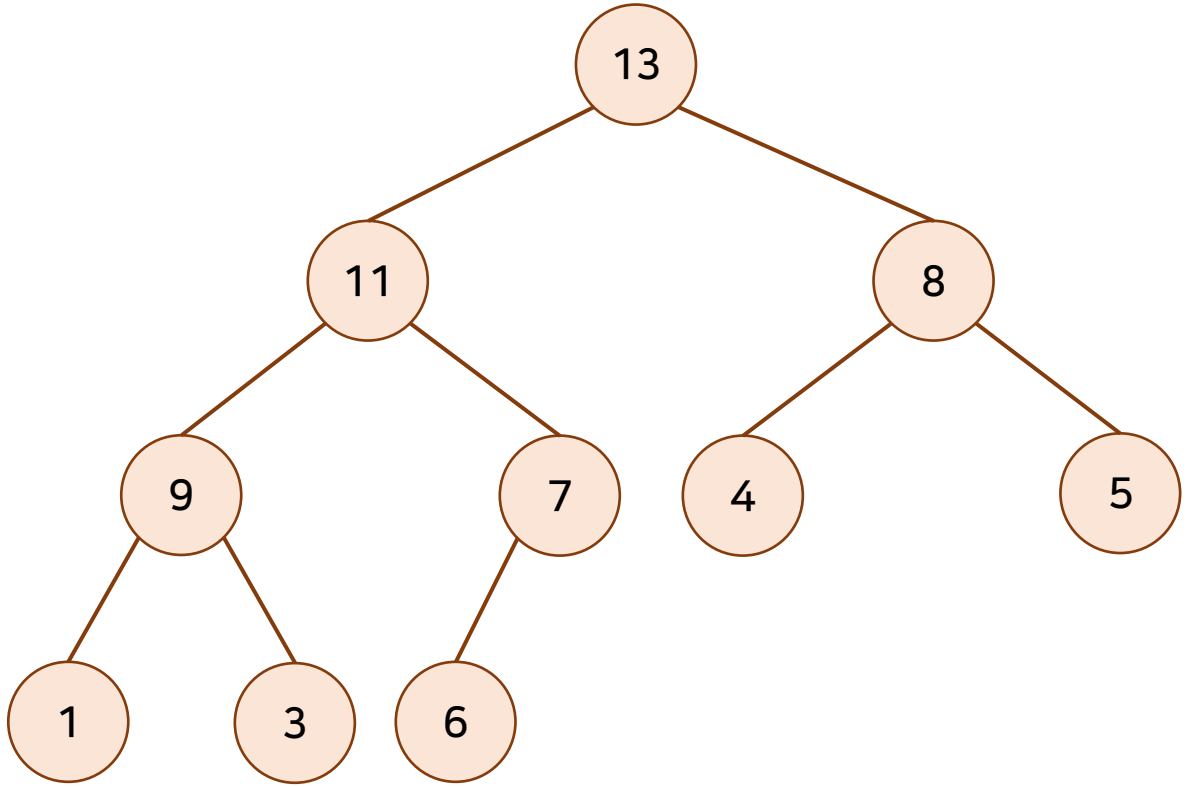
3	9	1	4	8	6	7	11	5	13



힙 (Heap)

ex) 최대 힙(Max Heap)

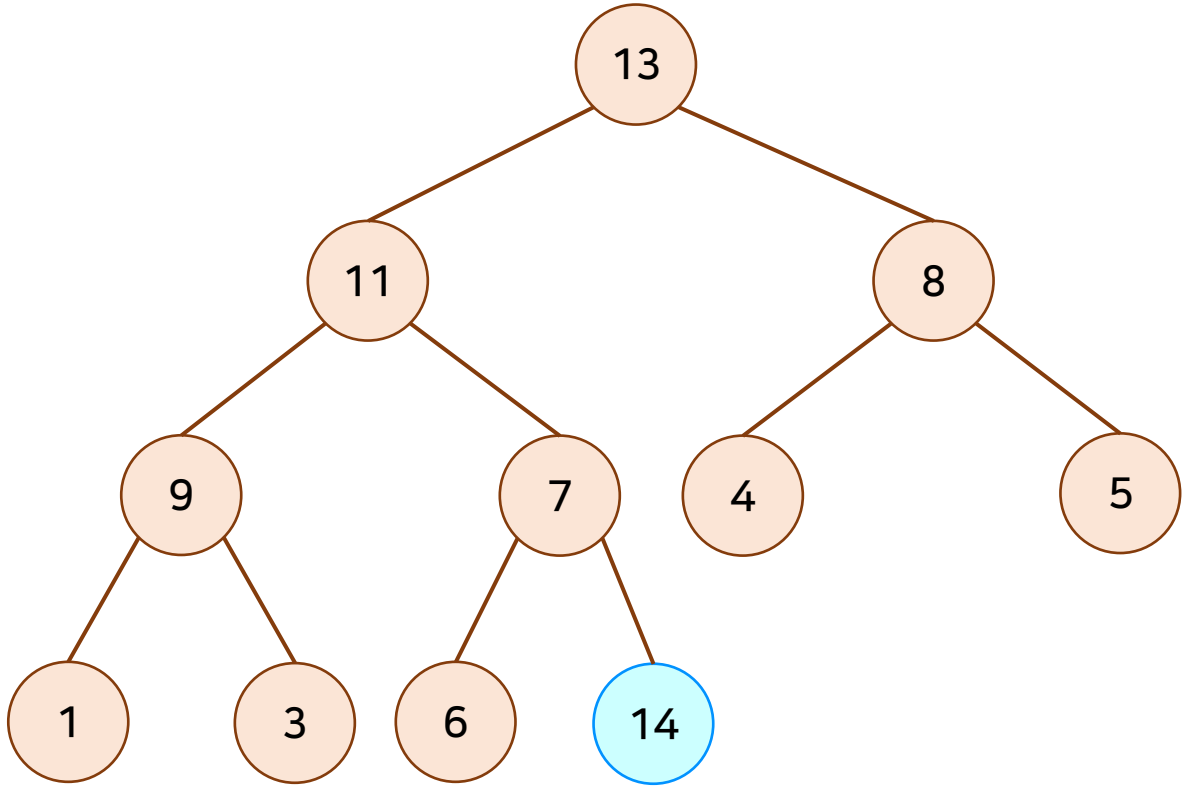
- 새로운 원소 삽입 14



힙 (Heap)

ex) 최대 힙(Max Heap)

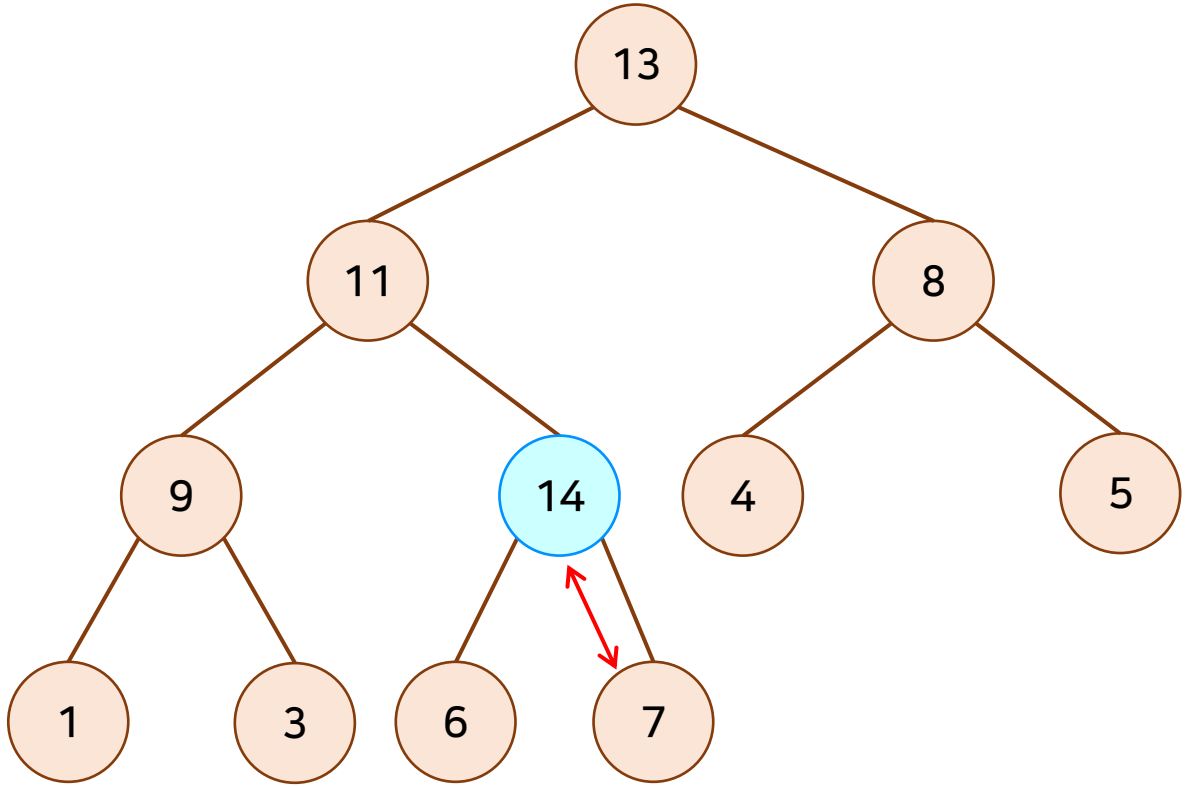
- 새로운 원소 삽입



힙 (Heap)

ex) 최대 힙(Max Heap)

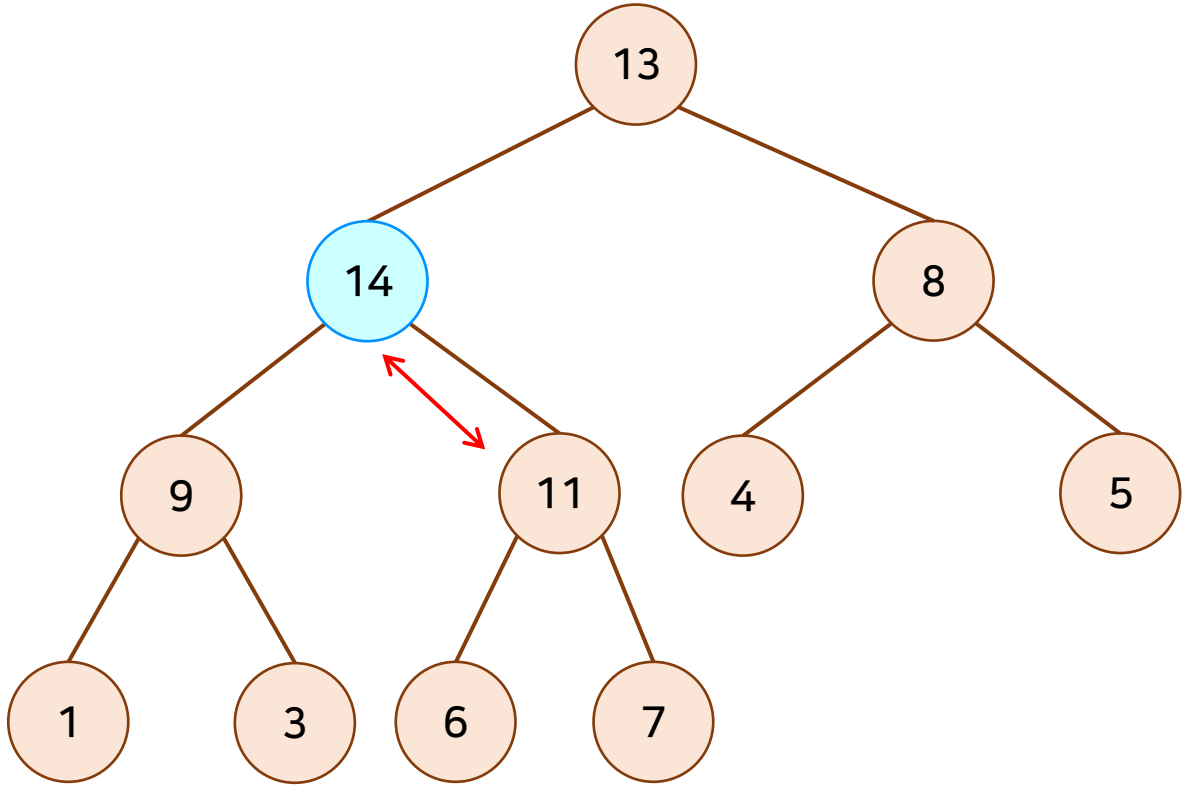
- 새로운 원소 삽입



힙 (Heap)

ex) 최대 힙(Max Heap)

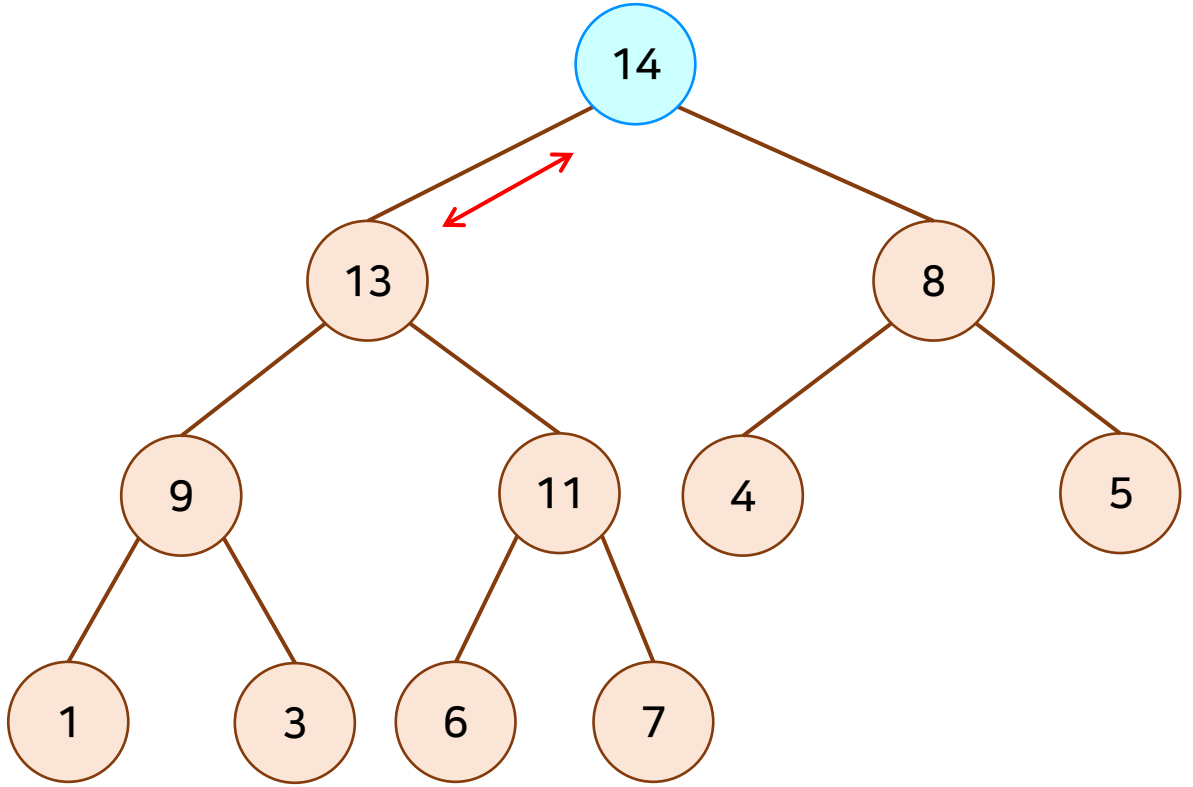
- 새로운 원소 삽입



힙 (Heap)

ex) 최대 힙(Max Heap)

- 새로운 원소 삽입

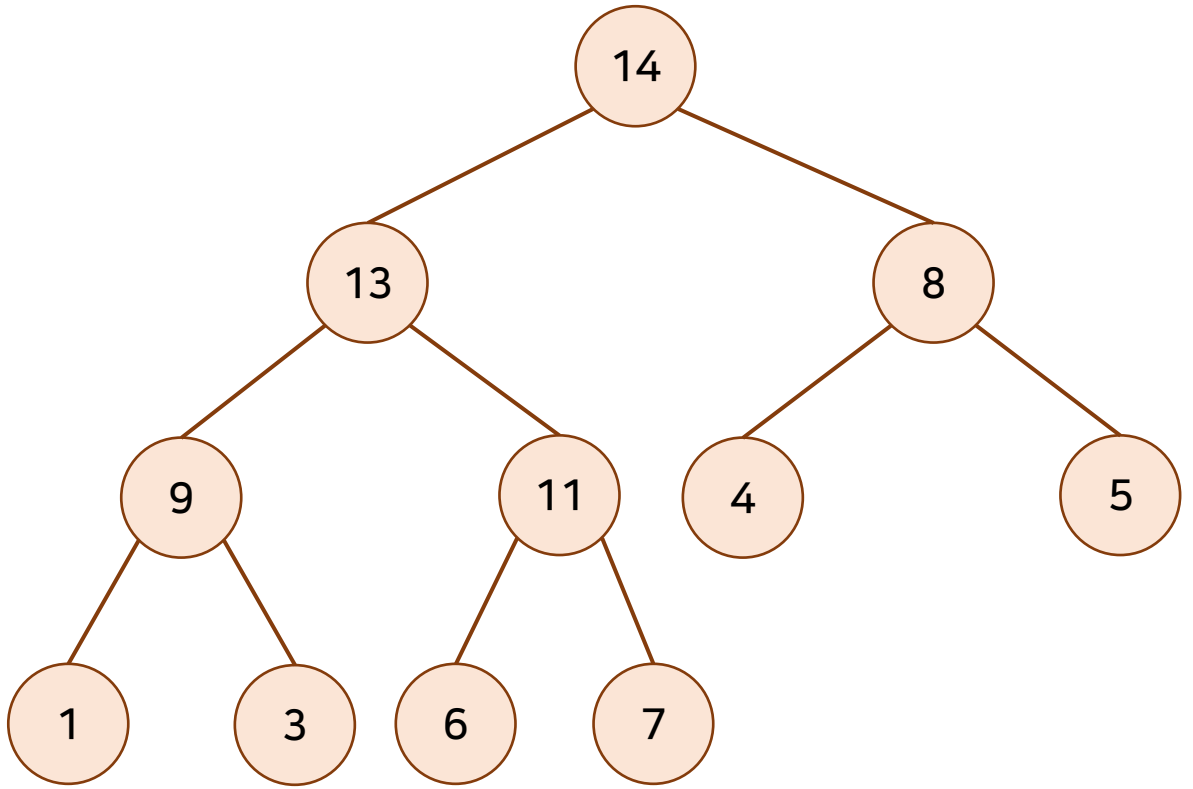


힙 (Heap)

ex) 최대 힙(Max Heap)

- 새로운 원소 삽입

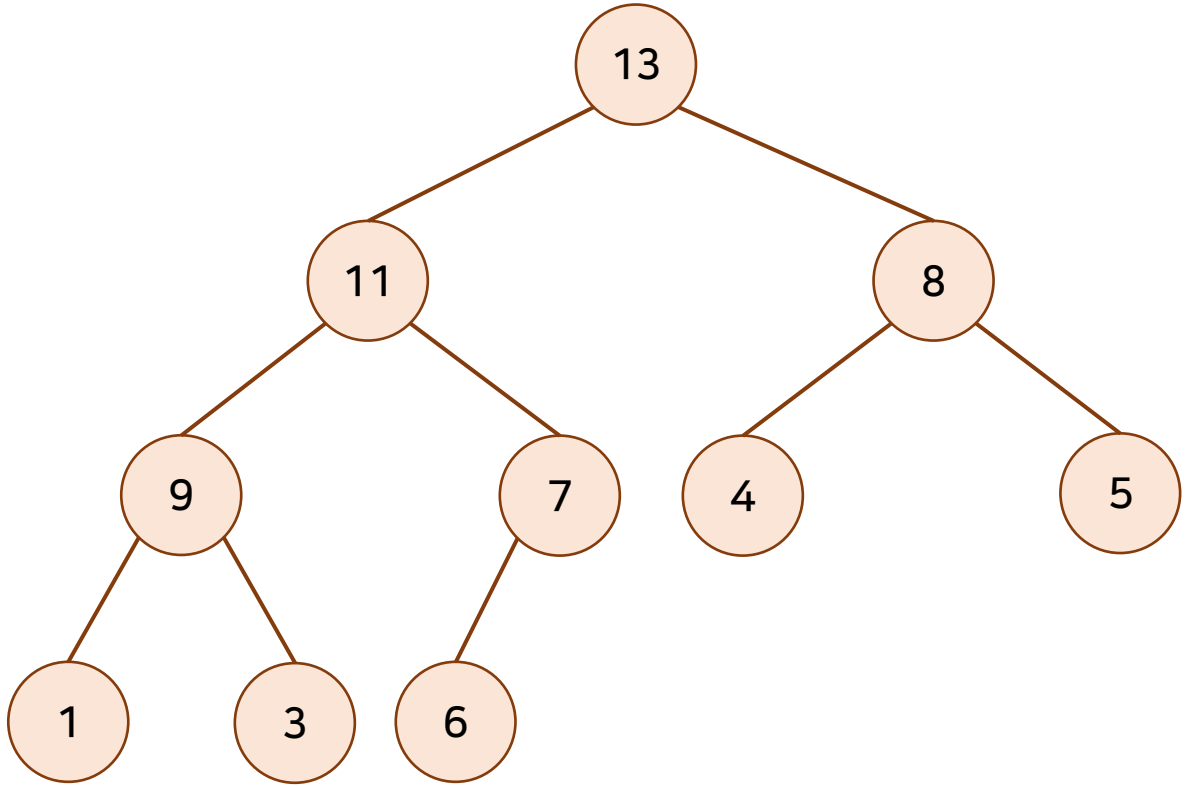
시간복잡도 : O(logN)



힙 (Heap)

ex) 최대 힙(Max Heap)

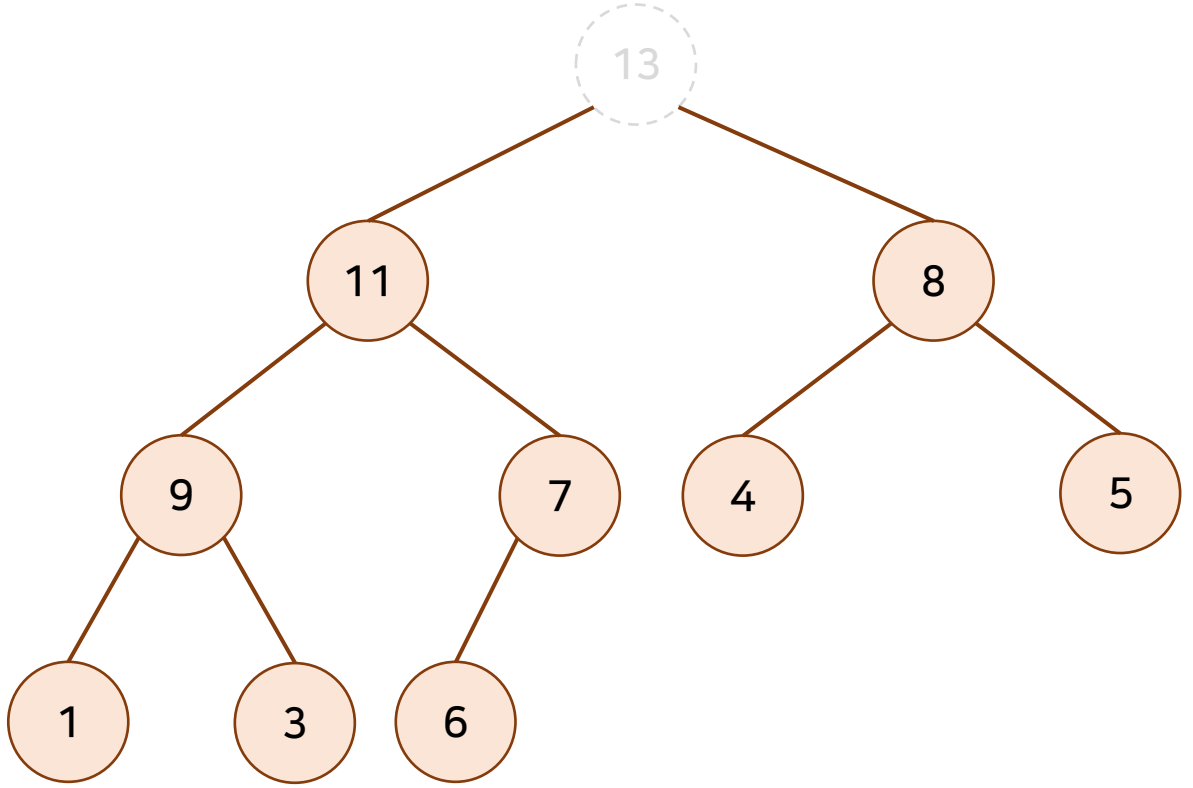
- 원소 삭제



힙 (Heap)

ex) 최대 힙(Max Heap)

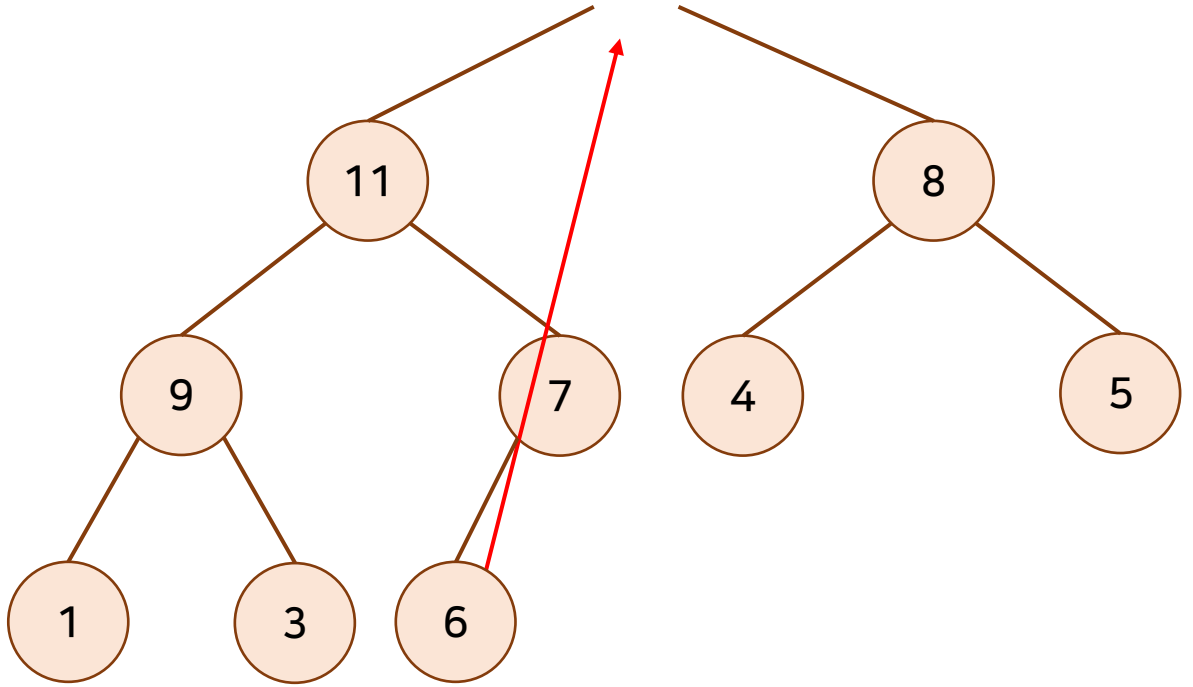
- 원소 삭제



힙 (Heap)

ex) 최대 힙(Max Heap)

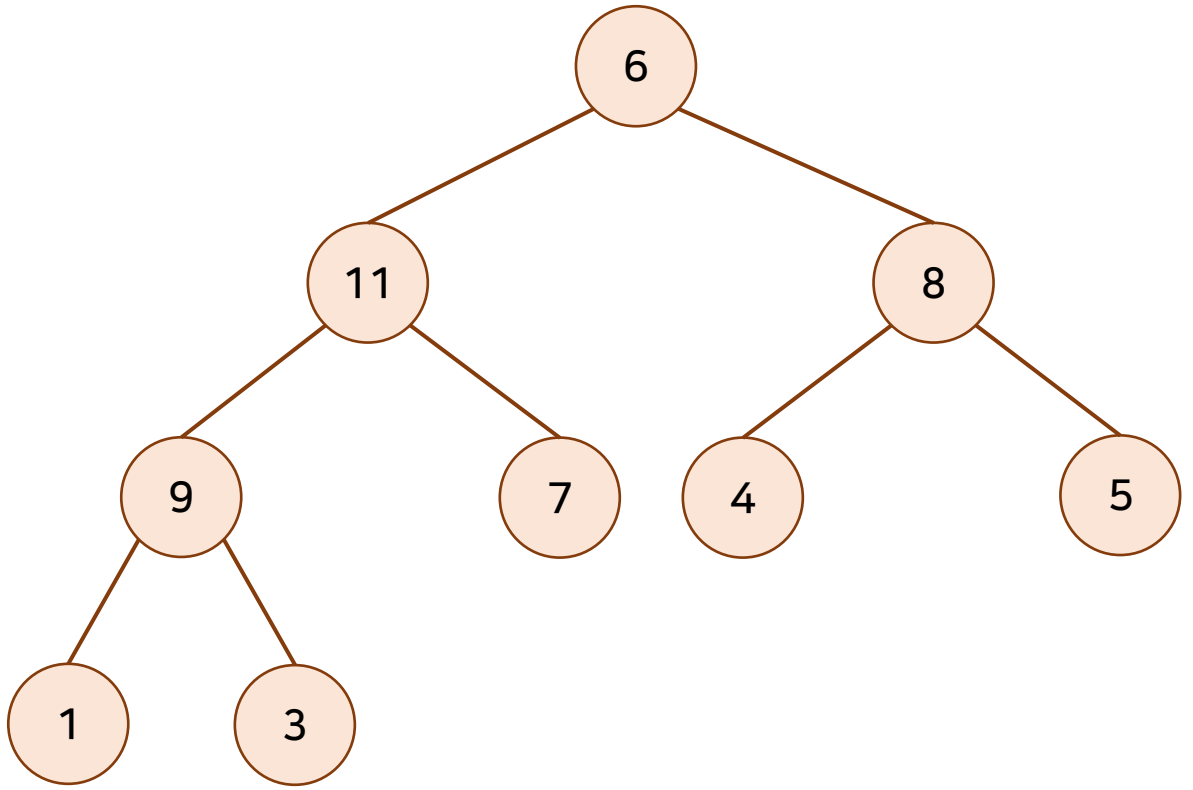
- 원소 삭제



힙 (Heap)

ex) 최대 힙(Max Heap)

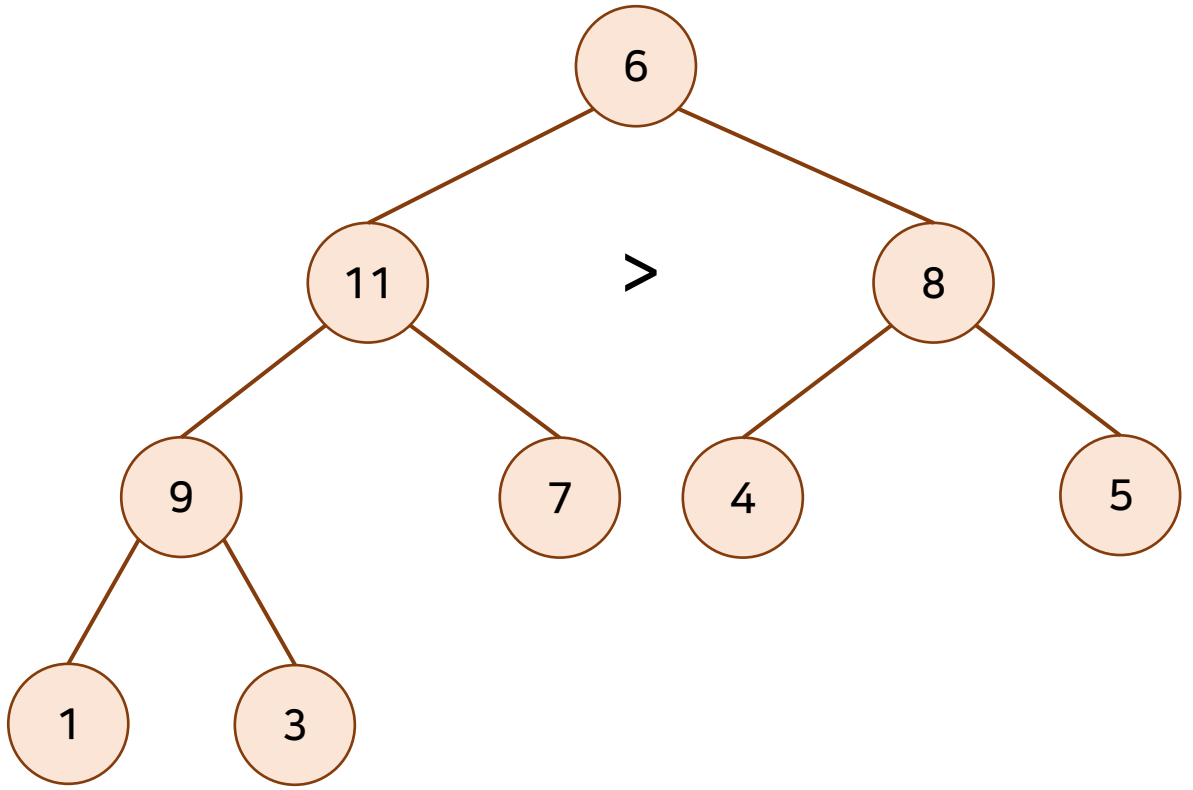
- 원소 삭제



힙 (Heap)

ex) 최대 힙(Max Heap)

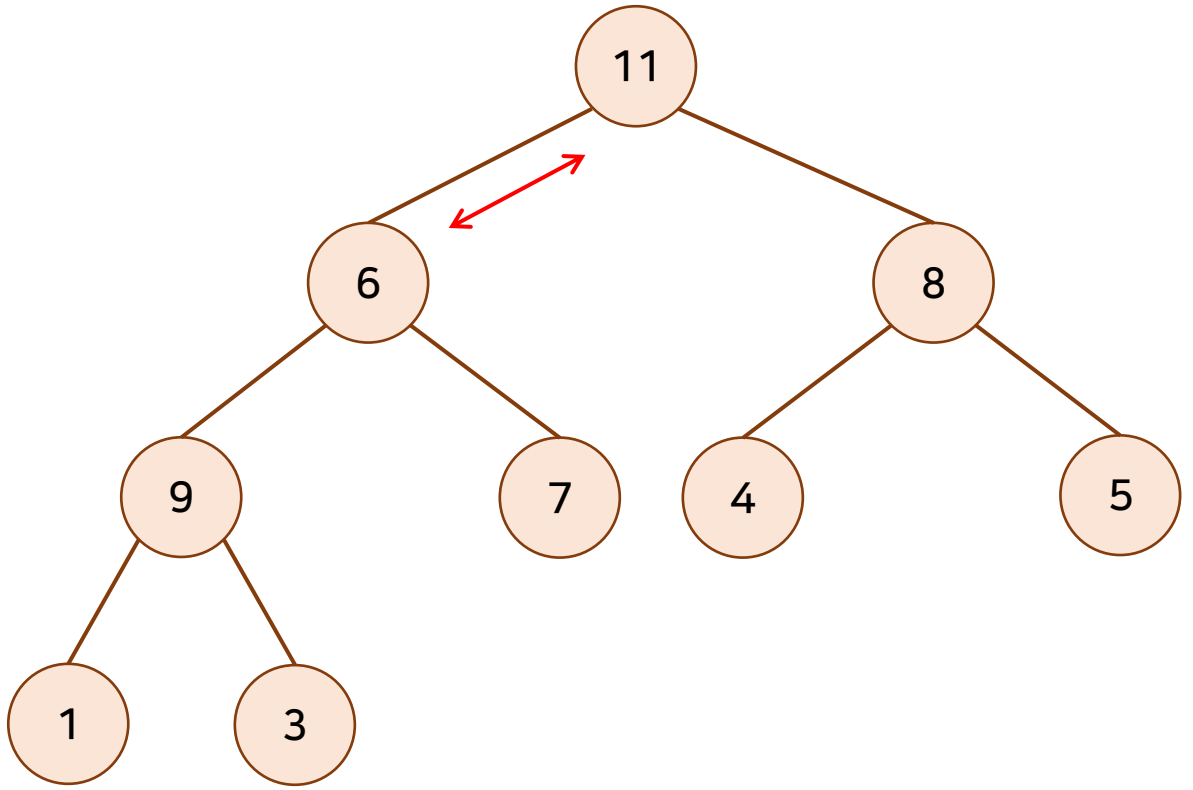
- 원소 삭제



힙 (Heap)

ex) 최대 힙(Max Heap)

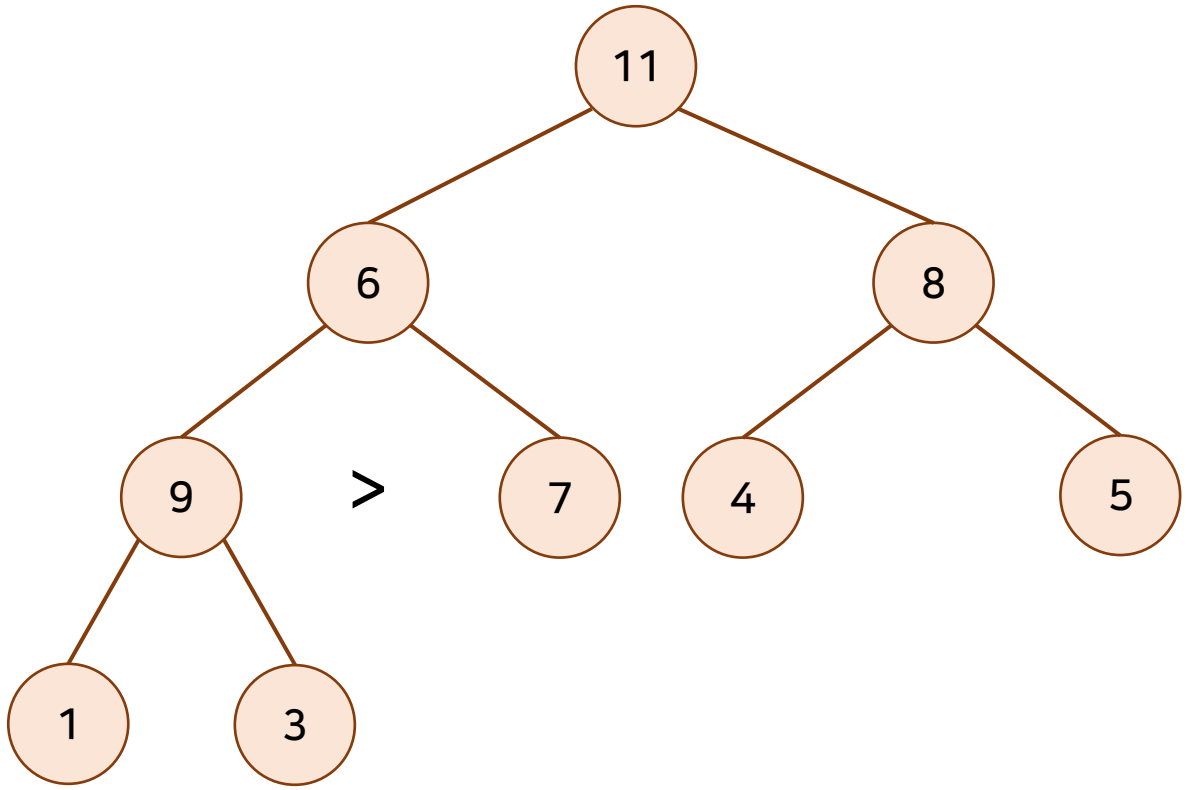
- 원소 삭제



힙 (Heap)

ex) 최대 힙(Max Heap)

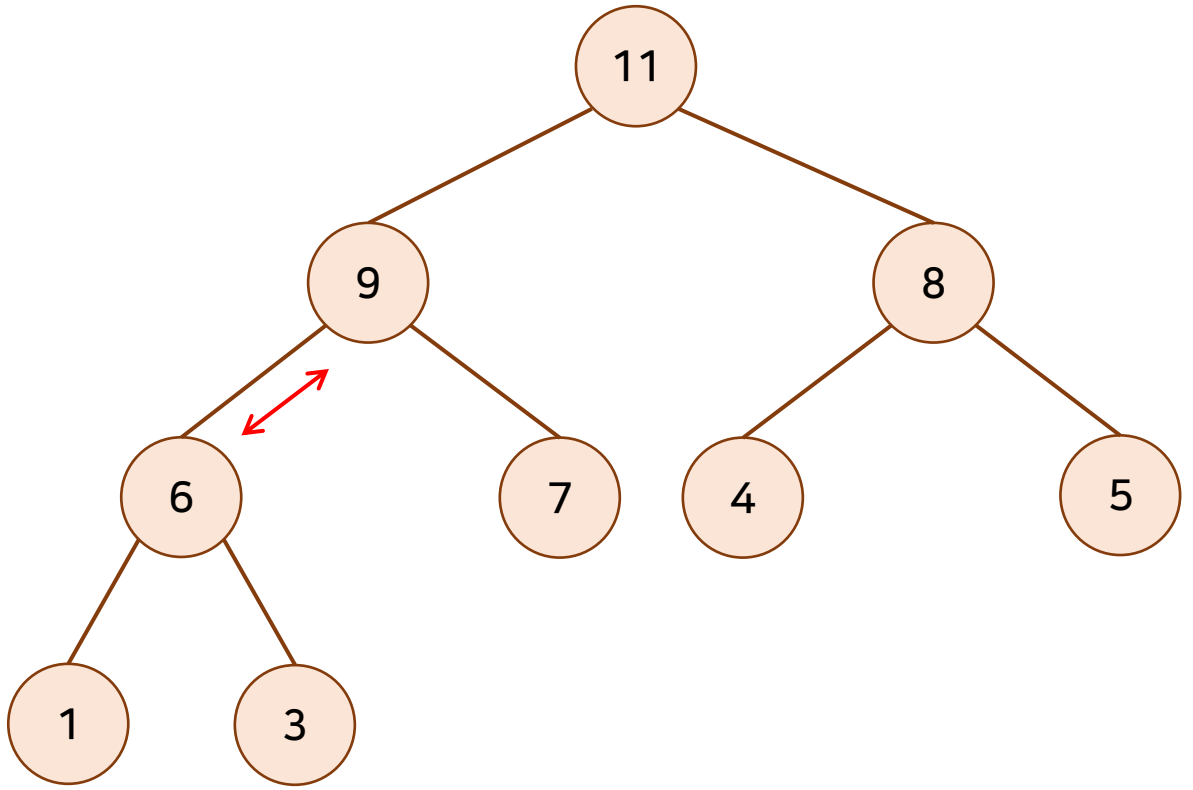
- 원소 삭제



힙 (Heap)

ex) 최대 힙(Max Heap)

- 원소 삭제

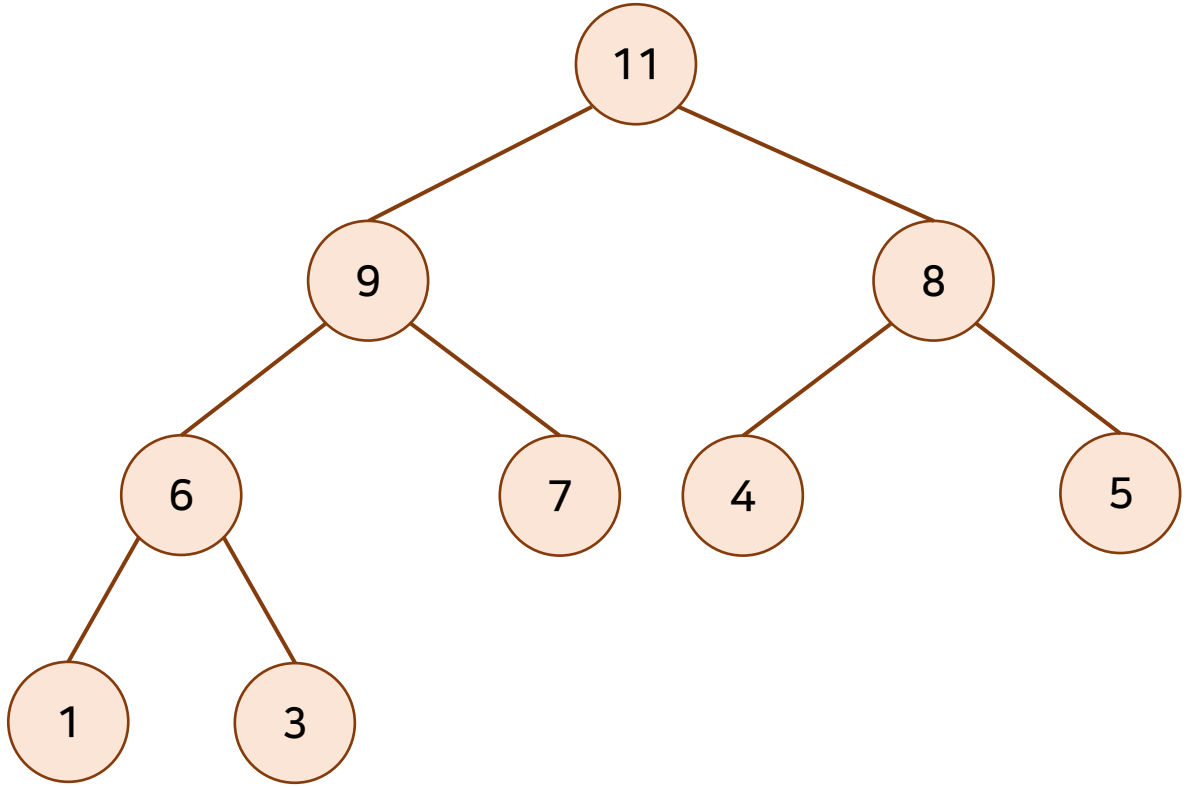


힙 (Heap)

ex) 최대 힙(Max Heap)

- 원소 삭제

시간복잡도 : O(logN)



우선순위 큐 (Priority Queue)

- 힙(Heap)으로 구현

  삽입 : O(logN) / 삭제 : O(logN)

- C++ STL에 템플릿 존재

Priority Queue in C++

std::priority_queue

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

- T : 원소의 타입
- Container : priority_queue의 구현에 사용되는 컨테이너
- Compare : 우선순위 기준
- #include<queue>

Priority Queue in C++

```
priority_queue<int> pq;

priority_queue<int, vector<int>> pq;

priority_queue<int, vector<int>, greater<int>> pq;
```

✓ 주의!

❖ greater<int> != greater<int>()

(비교 클래스) (비교 함수)

↑

PQ

❖ Priority_queue에서는 Sort에서의 비교함수와 반대 (링크)

- Priority_queue<int, vector<int>, greater<int>> : 작은 원소부터 pop

- sort(v.begin(), v.end(), greater<int>()) : 내림차순

Priority Queue in C++

ex) 3차원 좌표 (x, y, z)

y가 큰 순서대로, y가 같다면 x가 큰 순서대로 pop

```
struct point {
    int x;
    int y;
    int z;
};
```

```
bool cmp(point p1, point p2) {
    if (p1.y == p2.y) return p1.x > p2.x;
    else return p1.y > p2.y;
}
```

X

```
struct cmp {
    bool operator()(point p1, point p2){
        if (p1.y == p2.y) return p1.x < p2.x;
        else return p1.y < p2.y;
    }
};
```

O

Priority Queue in C++

ex) 3차원 좌표 (x, y, z)

y가 큰 순서대로, y가 같다면 x가 큰 순서대로 pop

```
struct point {
    int x;
    int y;
    int z;
};
```

```
priority_queue<point, vector<point>, cmp> pq;
pq.push({ 1, 2, 3 });
pq.push({ 1, 1, 1 });
pq.push({ 2, 6, 4 });
pq.push({ 0, 2, 3 });
while (!pq.empty()) {
    auto [x, y, z] = pq.top();
    cout << x << ' '<<y << ' ' << z << '\n';
    pq.pop();
}

// 2 6 4
// 1 2 3
// 0 2 3
// 1 1 1
```

BOJ 19638 (센티와 마법의 뽕망치)

문제

센티는 마법 도구들을 지니고 여행을 떠나는 것이 취미인 악당이다.

거인의 나라에 도착한 센티는 자신보다 키가 크거나 같은 거인들이 있다는 사실이 마음에 들지 않았다.

센티가 꺼내 들은 마법 도구는 바로 마법의 뽕망치로, 이 뽕망치에 맞은 사람의 키가 ⌊ 뽕망치에 맞은 사람의 키 / 2 ⌋로 변하는 마법 도구이다. 단, 키가 1인 경우 더 줄어들 수가 없어 뽕망치의 영향을 받지 않는다.

하지만 마법의 뽕망치는 횟수 제한이 있다. 그래서 센티는 마법의 뽕망치를 효율적으로 사용하기 위한 전략을 수립했다. 바로 매번 가장 키가 큰 거인 가운데 하나를 때리는 것이다.

과연 센티가 수립한 전략에 맞게 마법의 뽕망치를 이용한다면 거인의 나라의 모든 거인이 센티보다 키가 작도록 할 수 있을까?

입력

첫 번째 줄에는 센티를 제외한 거인의 나라의 인구수 N ($1 \le N \le 10^5$)과 센티의 키를 나타내는 정수 H_{centi} ($1 \le H_{centi} \le 2 \times 10^9$), 마법의 뽕망치의 횟수 제한 T ($1 \le T \le 10^5$)가 빈칸을 사이에 두고 주어진다.

두 번째 줄부터 N개의 줄에 각 거인의 키를 나타내는 정수 H ($1 \le H \le 2 \times 10^9$)가 주어진다.

BOJ 19638 (센티와 마법의 뽕망치)

- ✓ Naive 풀이
  - 가장 키가 큰 거인 찾기 = O(N)
  - 시간복잡도 = O(NT) / N = 10만, T = 10만 TLE
- ✓ T번 때리는 시간은 줄이기 어려움
- ✓ 가장 키가 큰 거인을 빠르게 찾을 수 없을까?

BOJ 19638 (센티와 마법의 뽕망치)

- ✓ 가장 키가 큰 거인 → Max Heap
- ✓ Priority_queue에 모든 사람의 키를 push
- ✓ 가장 큰 키를 pop : O(logN)
- ✓ 키를 절반으로 줄이고 다시 push : O(logN)
- ✓ 모든 거인이 센티보다 키가 작은가? = 거인 중 가장 큰 키가 센티보다 작은가?

O(TlogN)

BOJ 19638 (센티와 마법의 뽕망치)

```
int main(void) {
    int N, H, T; cin >> N >> H >> T;
    priority_queue<int> pq;
    for (int i = 0; i < N; i++) {
        int h; cin >> h;
        pq.push(h);
    }
    if (pq.top() < H) {
        cout << "YES\n" << 0;
        return 0;
    }
    for (int i = 0; i < T; i++) {
        int h = pq.top(); pq.pop();
        h = max(1, h / 2);
        pq.push(h);
        if (pq.top() < H) {
            cout << "YES\n" << i + 1;
            return 0;
        }
    }
    cout << "NO\n"<<pq.top();
}
```

BOJ 1655 (가운데를 말해요)

문제

수빈이는 동생에게 "가운데를 말해요" 게임을 가르쳐주고 있다. 수빈이가 정수를 하나씩 외칠때마다 동생은 지금까지 수빈이가 말한 수 중에서 중간값을 말해야 한다. 만약, 그동안 수빈이가 외친 수의 개수가 짝수개라면 중간에 있는 두 수 중에서 작은 수를 말해야 한다.

예를 들어 수빈이가 동생에게 1, 5, 2, 10, -99, 7, 5를 순서대로 외쳤다고 하면, 동생은 1, 1, 2, 2, 2, 2, 5를 차례대로 말해야 한다. 수빈이가 외치는 수가 주어졌을 때, 동생이 말해야 하는 수를 구하는 프로그램을 작성하시오.

입력

첫째 줄에는 수빈이가 외치는 정수의 개수 N이 주어진다. N은 1보다 크거나 같고, 100,000보다 작거나 같은 자연수이다. 그 다음 N줄에 걸쳐서 수빈이가 외치는 정수가 차례대로 주어진다. 정수는 -10,000보다 크거나 같고, 10,000보다 작거나 같다.

BOJ 1655 (가운데를 말해요)

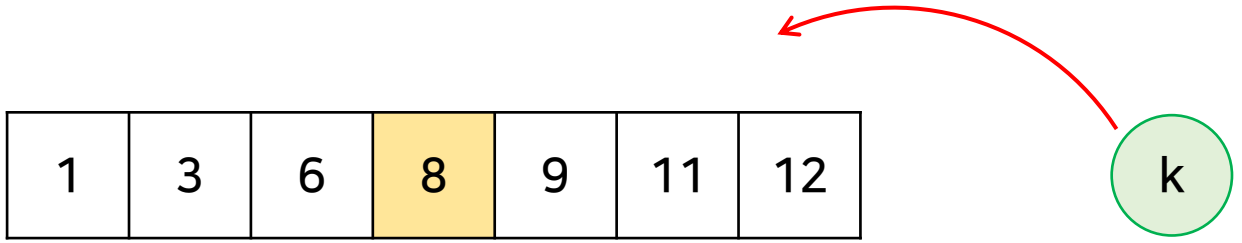
✓ Naive 풀이

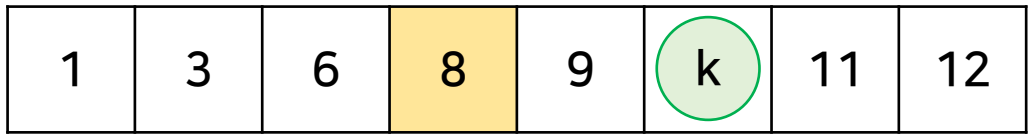
- N개의 수가 들어옴

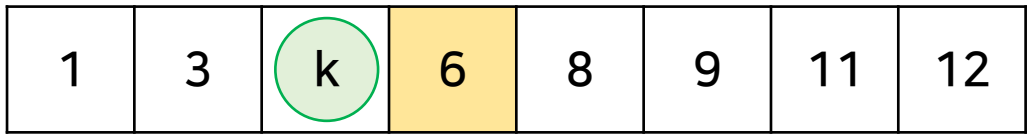
- 매번 정렬 후 중간값 찾기 → O(N × NlogN) TLE

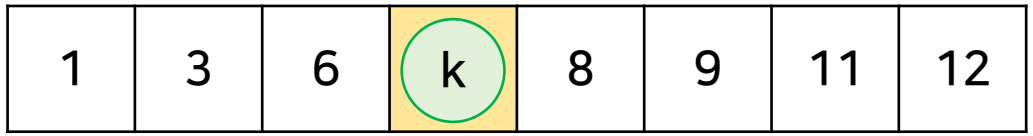
BOJ 1655 (가운데를 말해요)

✓ 새로운 값이 들어올 때 중간값이 될 수 있는 후보는?









기존의 중간값 근처의 값만 가능!

BOJ 1655 (가운데를 말해요)

- ✓ 굳이 매번 모두를 정렬할 필요는 없다
- ✓ 중간값 근처의 값? → 절반으로 나누었을 때 경계에 있는 값
- ✓ 지금까지 들어온 수를 크기 기준으로 절반씩 나누어서 관리하자!
- ✓ 작은 수 : S 집합, 큰 수 : L 집합

BOJ 1655 (가운데를 말해요)

- ✓ N = 홀수인 경우
  - S 집합의 원소가 L 집합의 원소보다 1개 더 많도록 유지
- ✓ N = 짝수인 경우
  - S와 L의 원소의 개수가 같도록 유지

→ 중간값 = S 집합의 가장 큰 원소

- ✓ 2개의 Priority_queue로 집합 관리
- ✓ 시간복잡도 : O(NlogN)

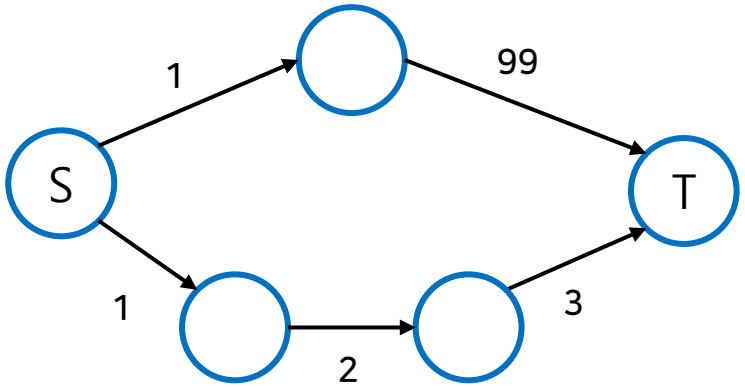
다익스트라 알고리즘

- 최단 경로 탐색 알고리즘 (Dijkstra Algorithm)

- 시작점으로부터 다른 모든 점까지의 최단 거리를 구하는 문제

- BFS와의 차이점

  - BFS : 간선의 가중치가 모두 1
  - 다익스트라 : 간선의 가중치가 음이 아닌 값



다익스트라 알고리즘

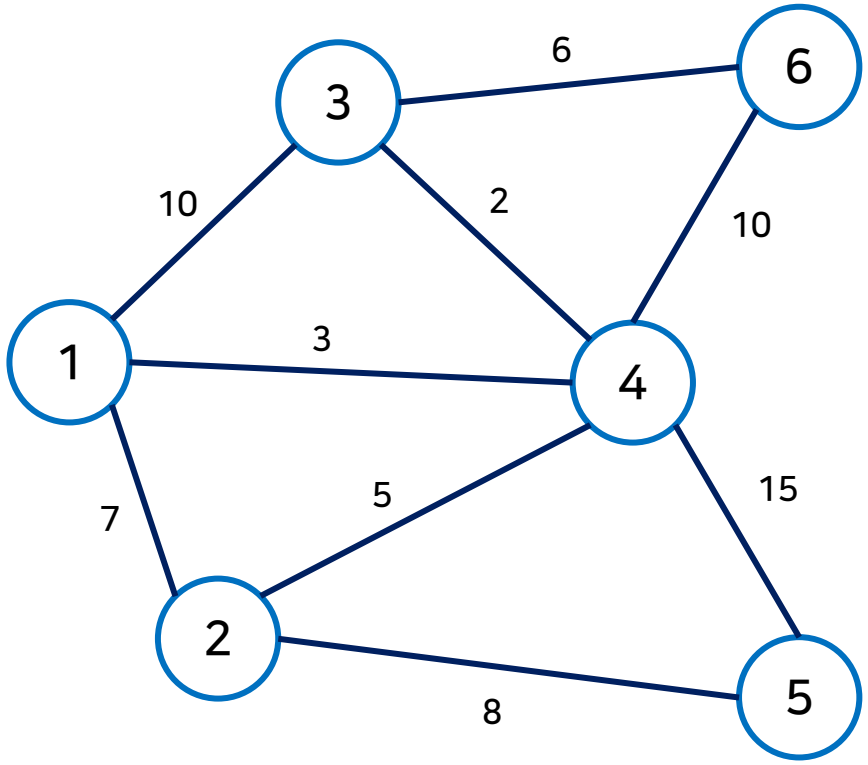
1. 시작점의 거리를 0, 나머지 정점까지의 거리를 최대(MAX)로 설정

2. 아직 방문하지 않은 정점 중 거리가 가장 짧은 정점을 방문 (처음엔 시작점 방문)

3. 인접한 정점 중 아직 방문하지 않은 정점들의 거리 갱신

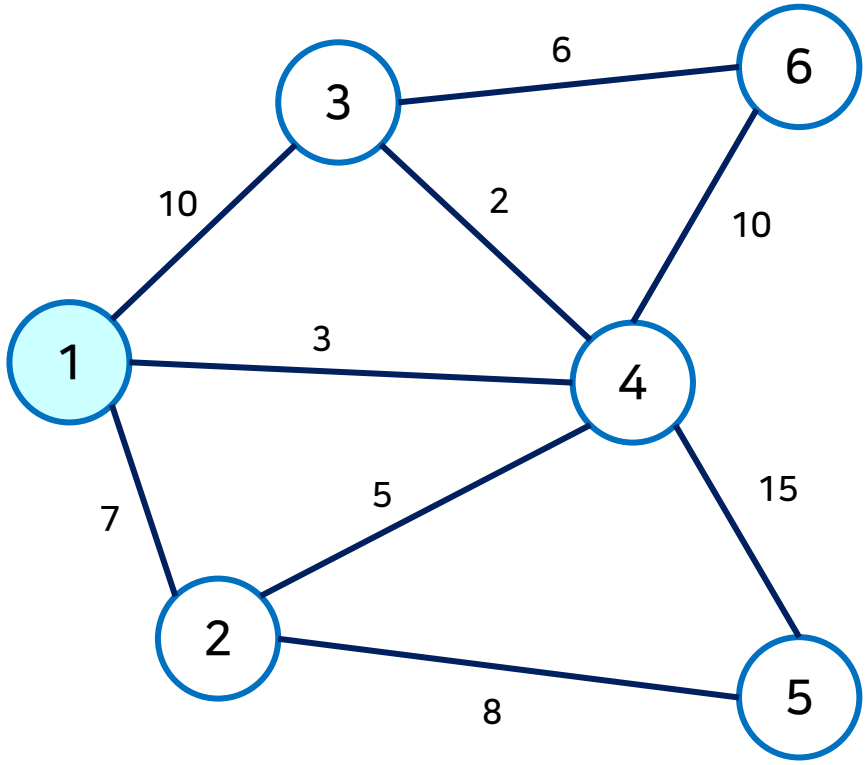
4. 2 ~ 4 반복

다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	MAX	MAX	MAX	MAX	MAX	MAX

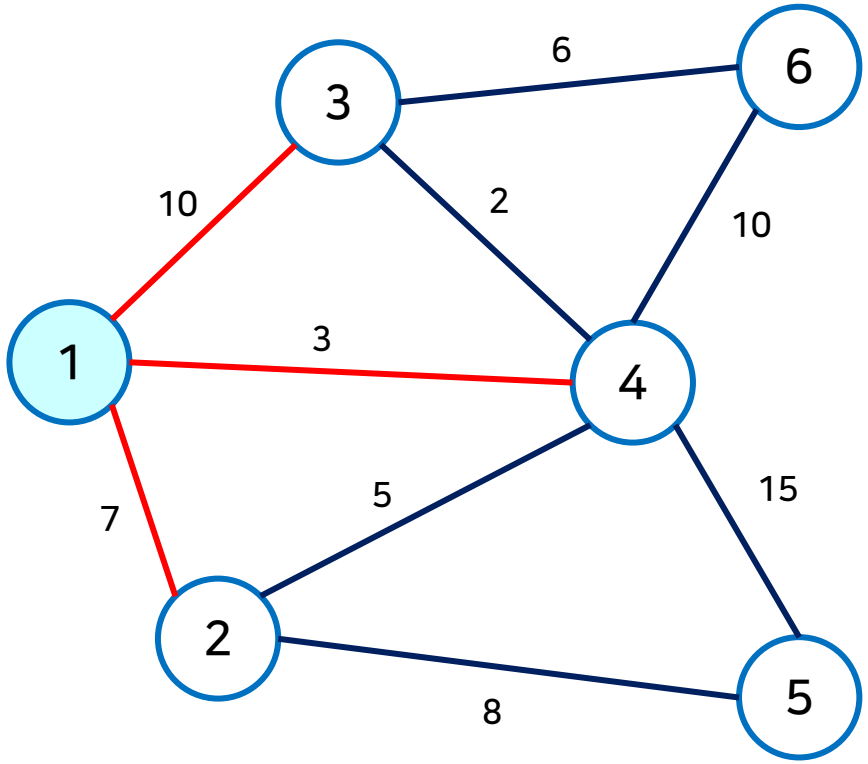
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	MAX	MAX	MAX	MAX	MAX

갱신된 정점 : 1

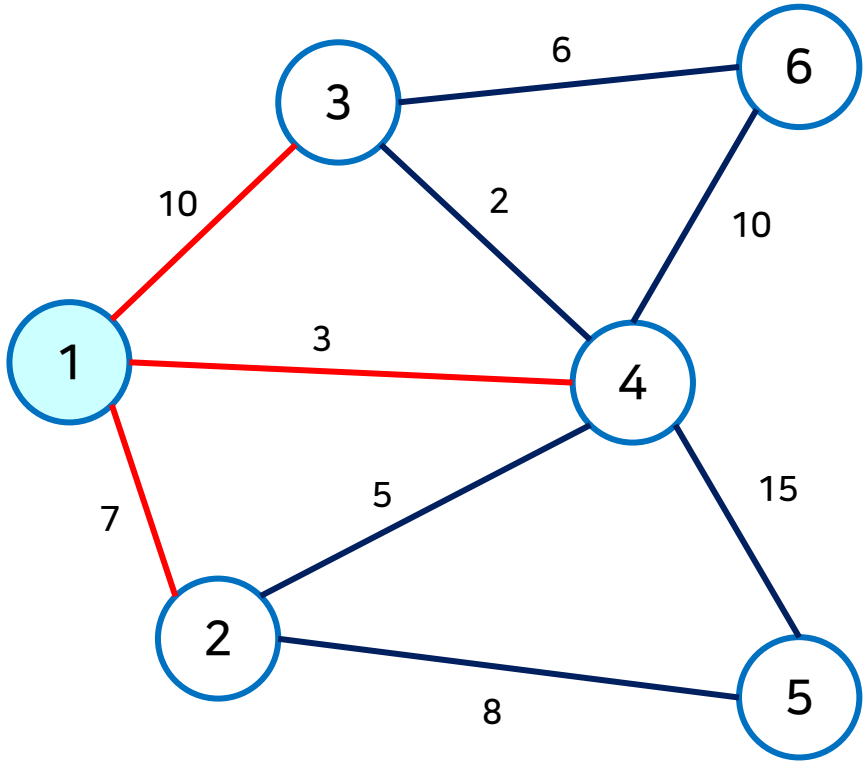
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	MAX	MAX	MAX	MAX	MAX

갱신된 정점 : 1

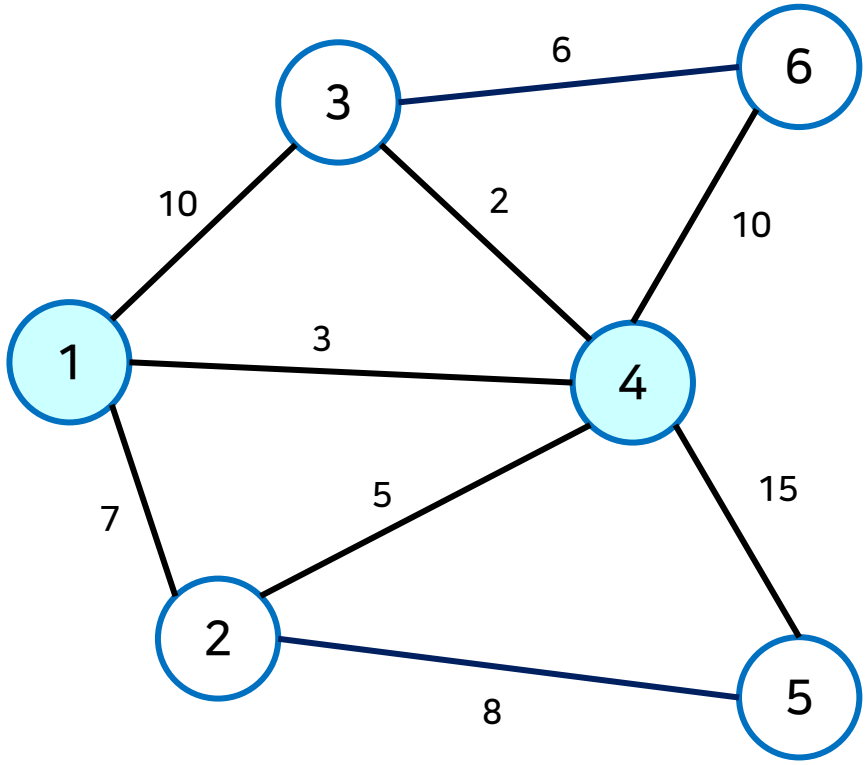
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	10	3	MAX	MAX

갱신된 정점 : X, 2, 3, 4

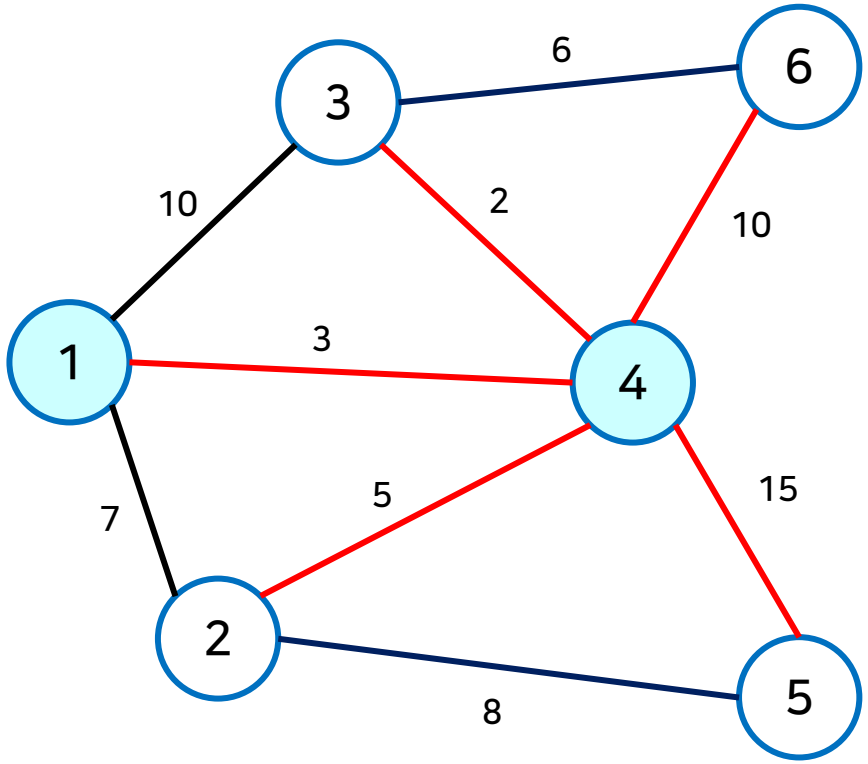
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	10	3	MAX	MAX

갱신된 정점 : 2, 3, 4

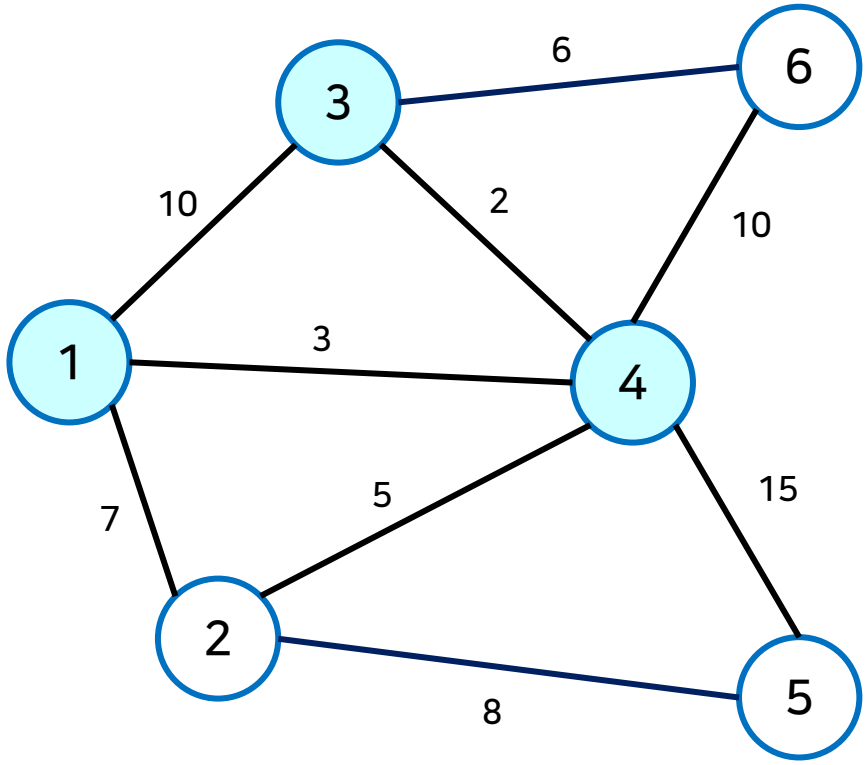
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	18	13

갱신된 정점 : 2, 3, ~~4~~, 5, 6

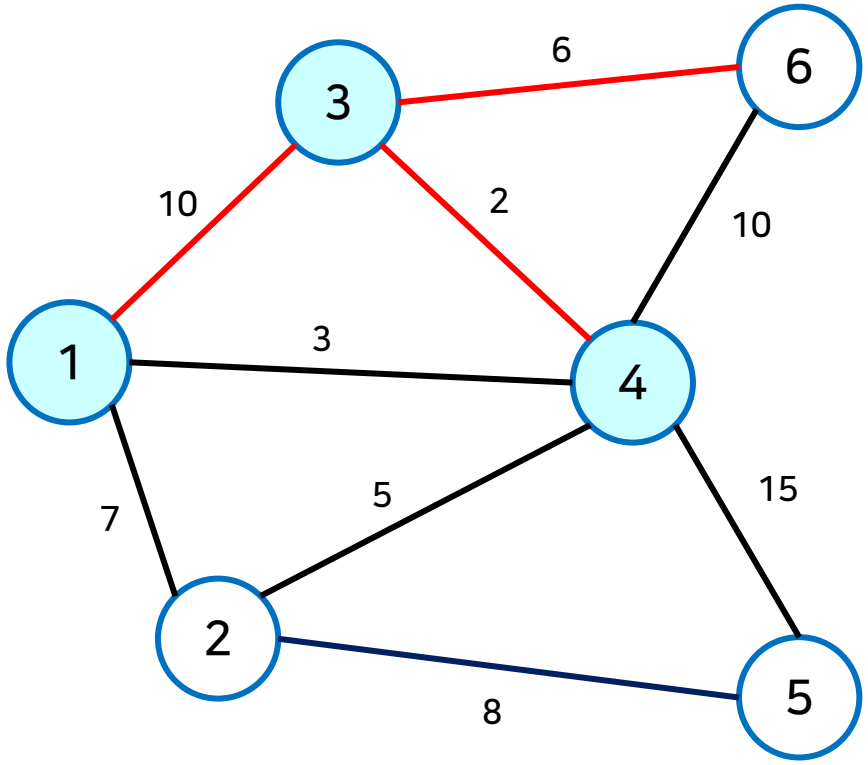
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	18	13

갱신된 정점 : 2, 3, 5, 6

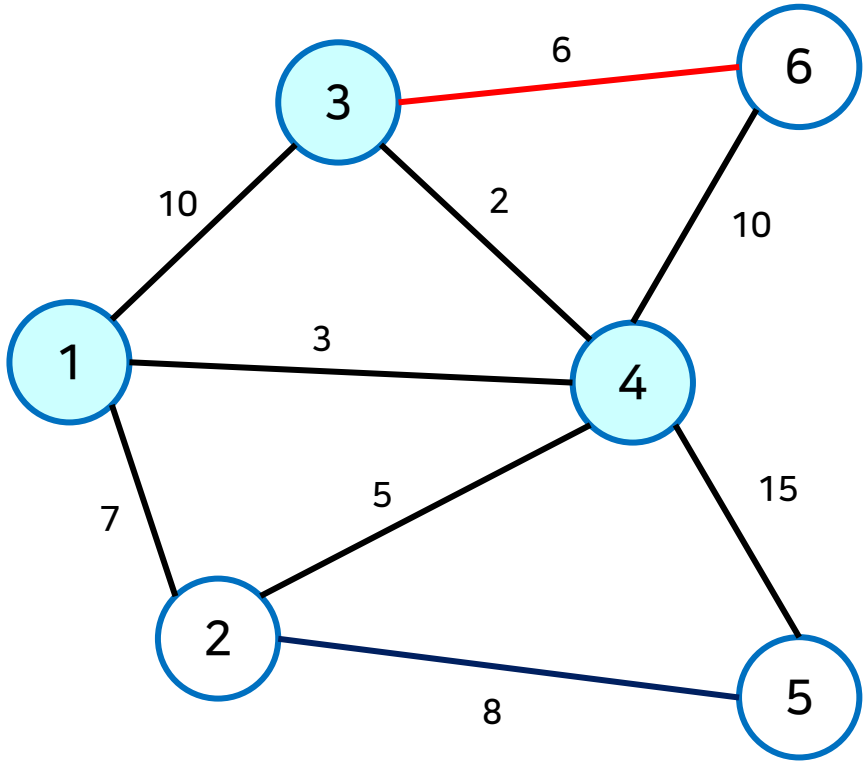
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	18	13

갱신된 정점 : 2, 3, 5, 6

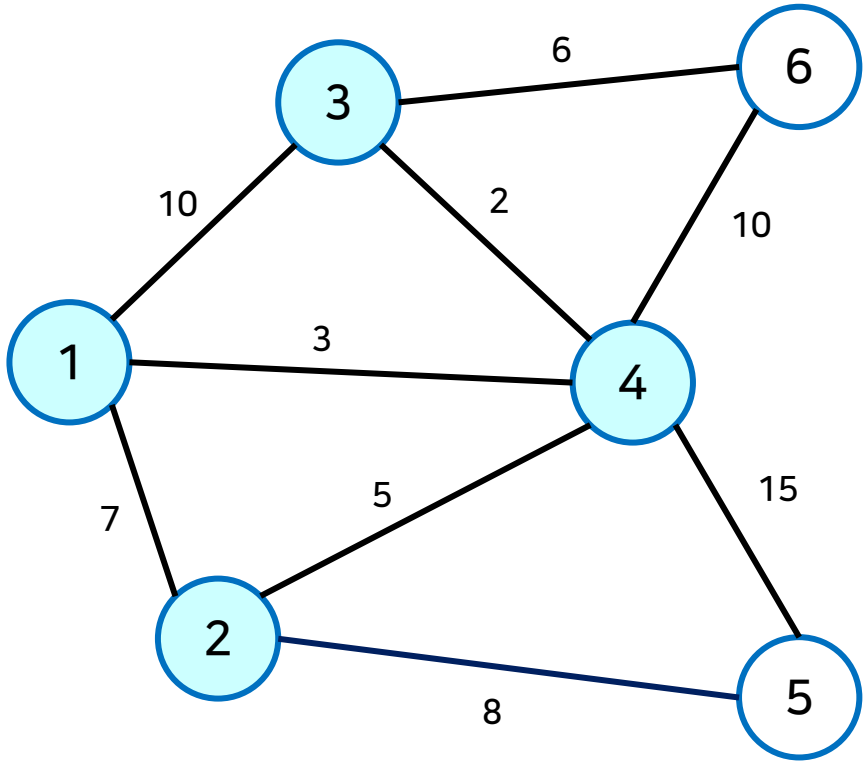
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	18	11

갱신된 정점 : 2, ~~3~~, 5, 6

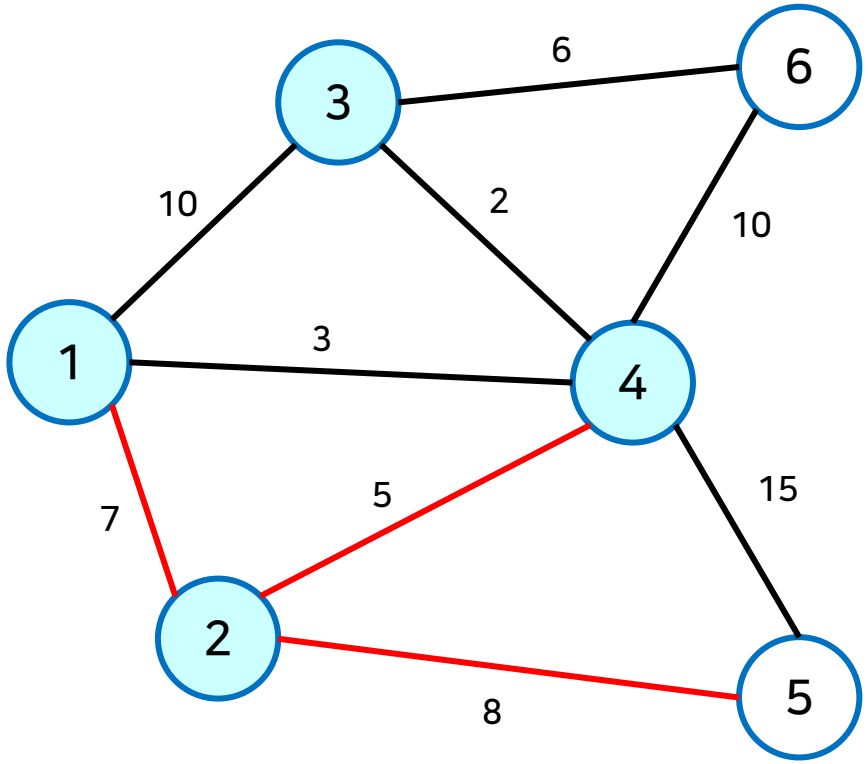
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	18	11

갱신된 정점 : 2, 5, 6

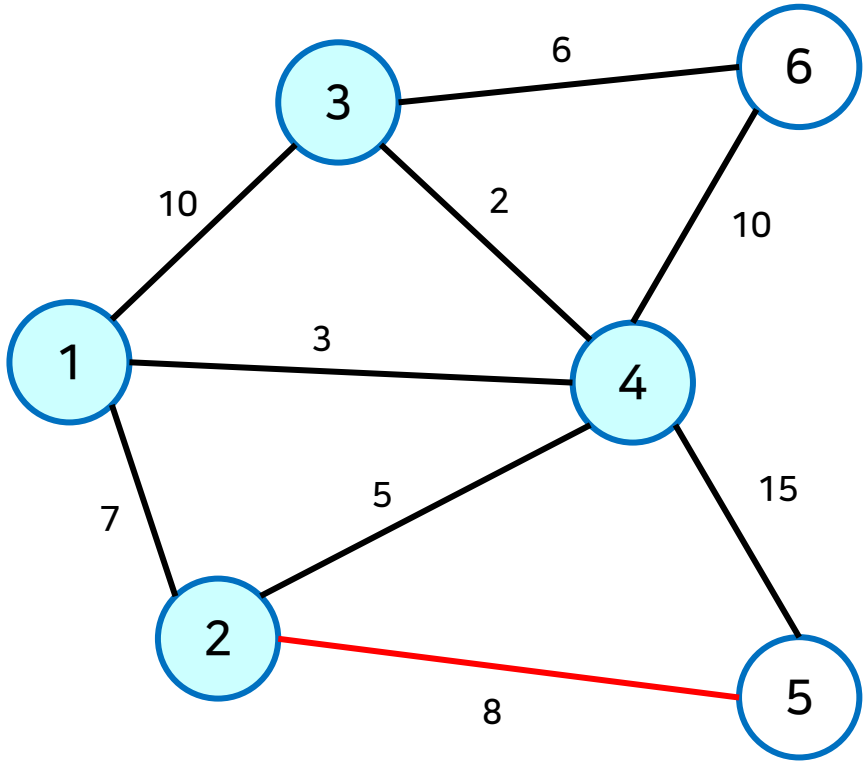
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	18	11

갱신된 정점 : 2, 5, 6

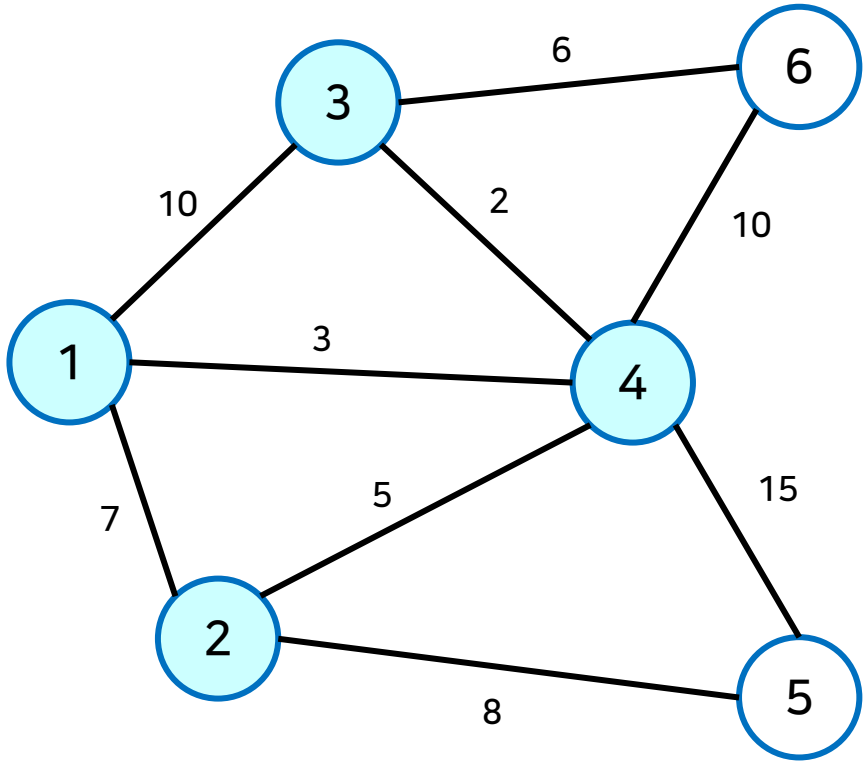
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	15	11

갱신된 정점 : ~~2~~, 5, 6

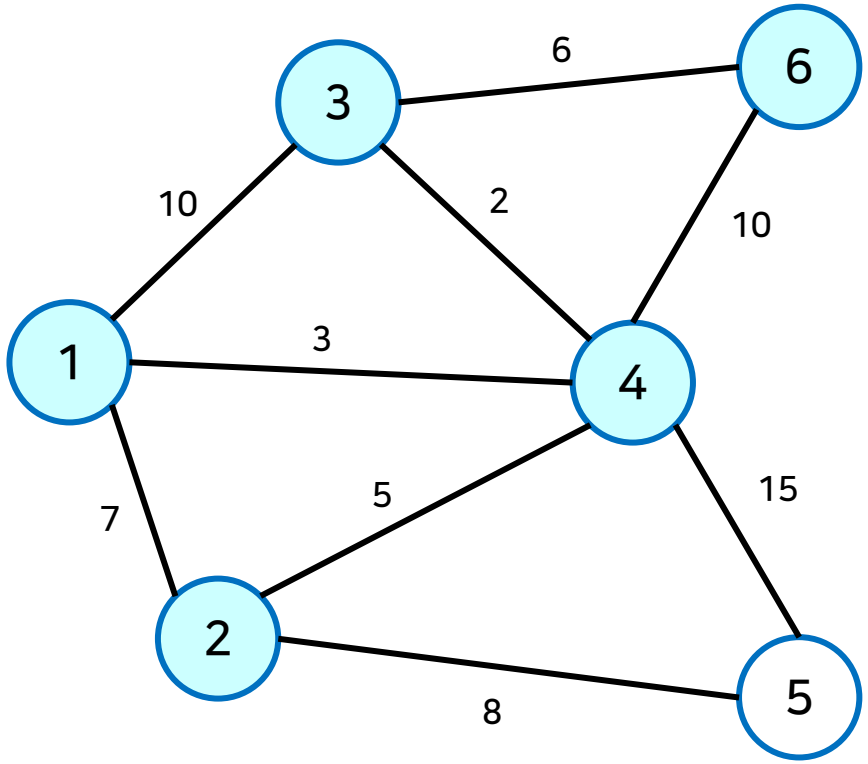
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	15	11

갱신된 정점 : 5, 6

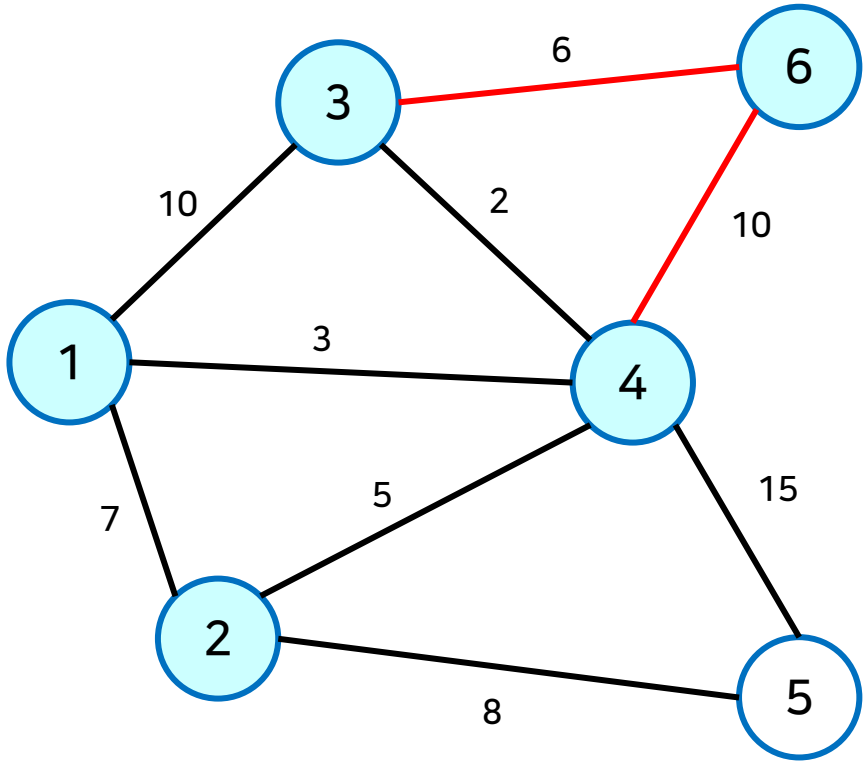
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	15	11

갱신된 정점 : 5, 6

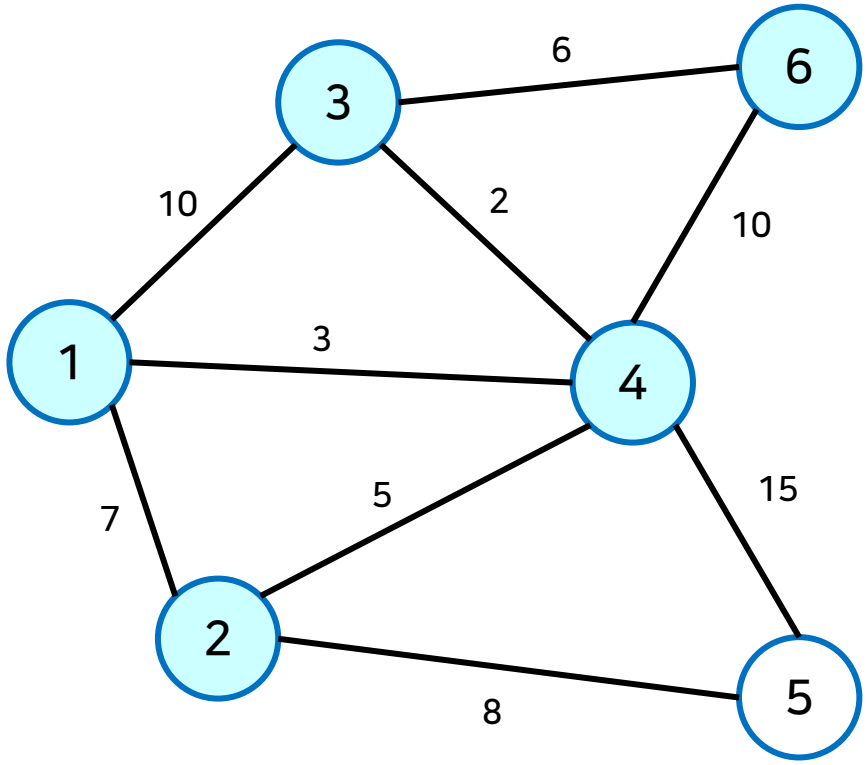
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	15	11

갱신된 정점 : 5, 6

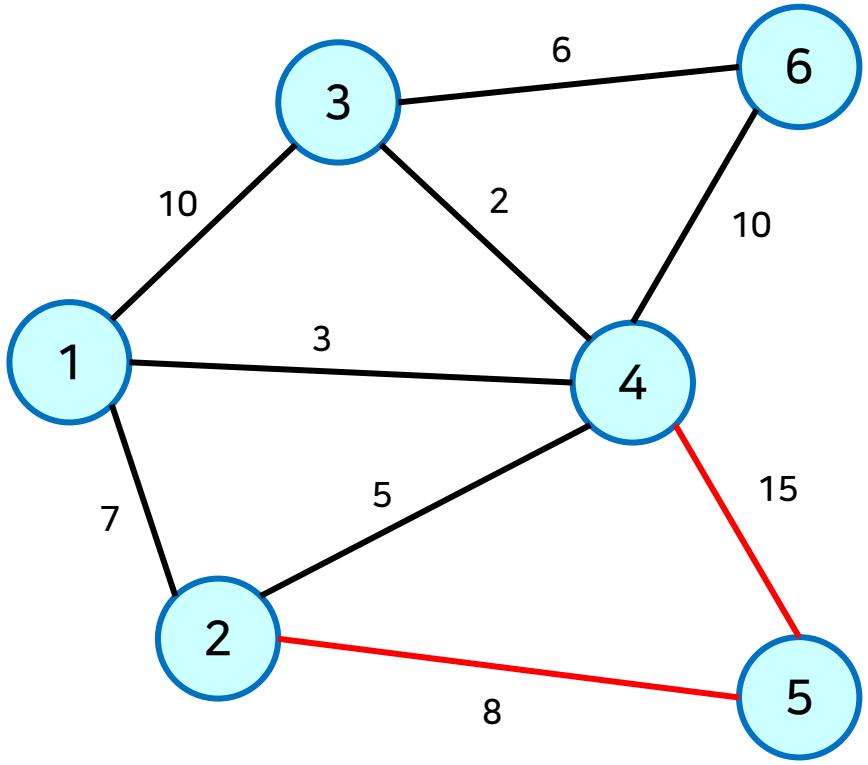
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	15	11

갱신된 정점 : 5, ~~6~~

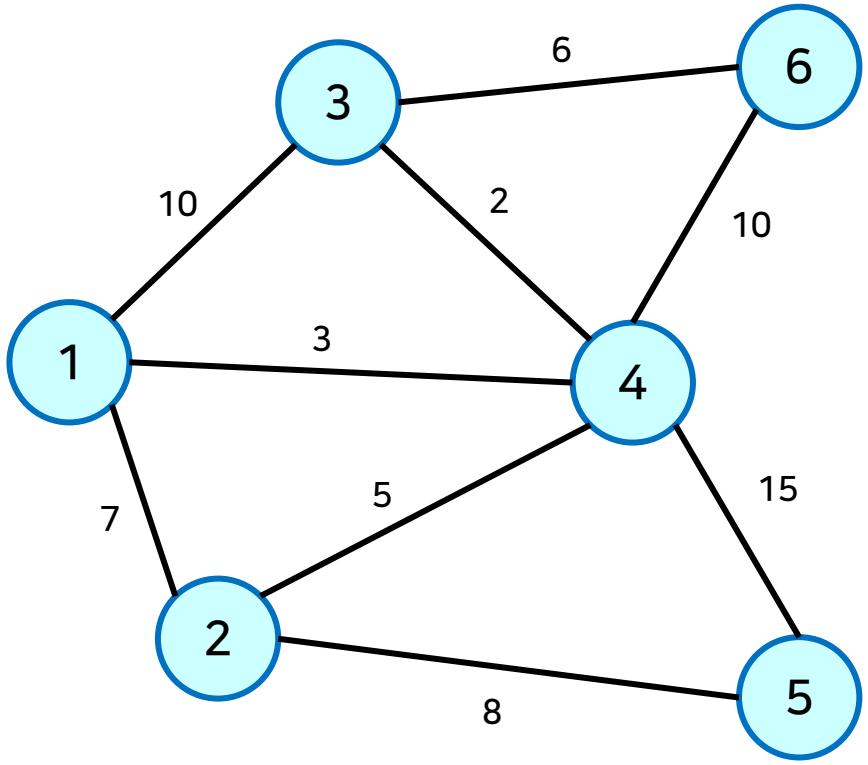
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	15	11

갱신된 정점 : 5

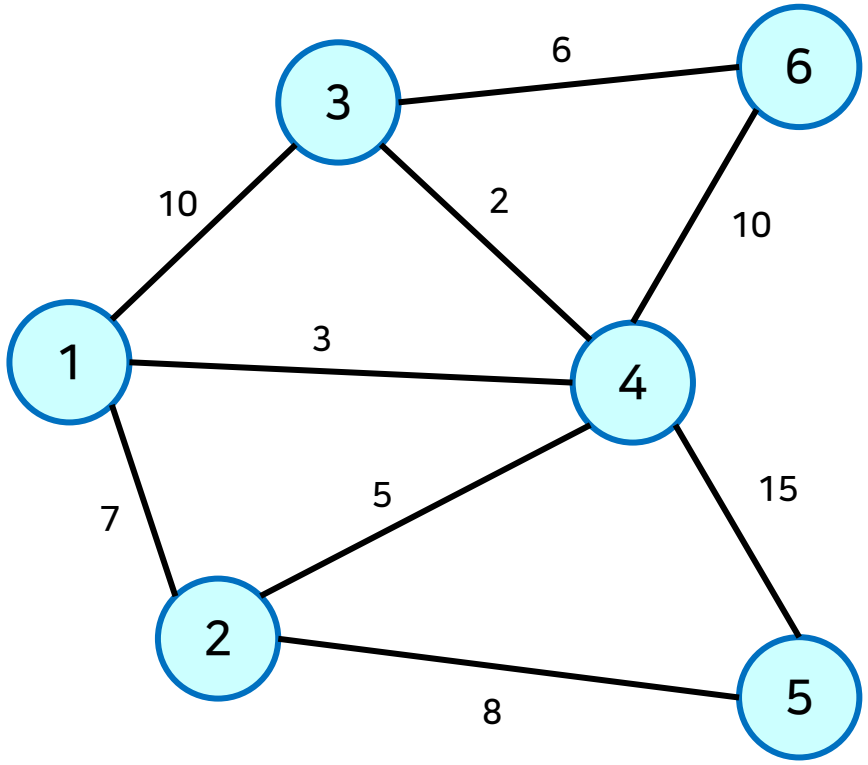
다익스트라 알고리즘



정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	15	11

갱신된 정점 : ~~5~~

다익스트라 알고리즘

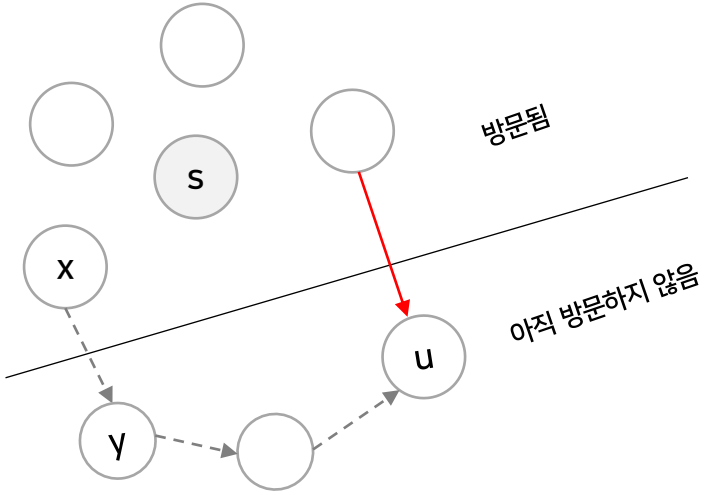


정점 번호	1	2	3	4	5	6
최단 거리	0	7	5	3	15	11

END

다익스트라 알고리즘

Q. 항상 최단거리가 구해진다고 보장할 수 있는가? (정당성)



Proof) 귀류법

1. 최단거리를 제대로 구하지 못하는 정점 u가 있다고 가정
2. u는 시작점이 될 수는 없음 (시작점은 항상 최단거리가 보장되므로)
3. u 이전에 방문한 정점 / 아직 방문하지 않은 정점으로 구분 가능
4. 최단거리를 제대로 구하지 못했다 → 아직 방문하지 않은 정점을 통해 더 빠른 경로가 존재한다 (점선)
5. dis[y] = dis[x] + d(x, y) / x가 이미 방문된 정점이므로 dis[y]에 최단거리가 계산되어 있음
6. y보다 u가 먼저 방문되었다 = dis[u] ≤ dis[y] *모순

다익스트라 알고리즘

<시간 복잡도>

- 아직 방문하지 않은 정점 중 거리가 가장 짧은 정점 찾기 = O(V)
- 정점을 방문하는 횟수 = O(V)
- Total = O(V^2) Slow

- 거리가 가장 짧은 정점 찾기 → Priority_queue 사용 → O(logV)

다익스트라 알고리즘

<시간 복잡도>

- 각 정점은 한 번씩 방문되고 인접한 간선들을 모두 체크

- 모든 간선을 한 번씩 체크함 (양방향 간선인 경우 2번) → O(|E|)

- Priority_queue에 보관되는 원소의 최대 개수 → O(|E|)

∴ O(|E|log|E|) = O(|E|log|V|)

다익스트라 알고리즘

```cpp
int dis[V];
vector<pair<int, int>> adj[V];

void dijkstra(int S) {
    fill(dis, dis + V, MAX);
    priority_queue<pair<int, int>> pq;
    dis[S] = 0;
    pq.push({ 0, S });
    while (!pq.empty()) {
        int d = -pq.top().first;
        int u = pq.top().second;
        pq.pop();
        if (d > dis[u]) continue;
        for (int i = 0; i < adj[u].size(); i++) {
            int v = adj[u][i].first;
            int c = adj[u][i].second;
            if (dis[v] > dis[u] + c) {
                dis[v] = dis[u] + c;
                pq.push({ -dis[v], v });
            }
        }
    }
}
```

```cpp
int dis[V];
vector<pair<int, int>> adj[V];

void dijkstra(int S) {
    fill(dis, dis + V, MAX);
    priority_queue<pair<int, int>> pq;
    dis[S] = 0;
    pq.push({ 0, S });
    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (-d > dis[u]) continue;
        for (auto [v, c] : adj[u]) {
            if (dis[v] > dis[u] + c) {
                dis[v] = dis[u] + c;
                pq.push({ -dis[v], v });
            }
        }
    }
}
```

C++ auto 사용

다익스트라 알고리즘

<주의할 점>

- 음의 가중치를 갖는 간선이 존재하면 안된다
- 최장거리를 구하는 문제에는 사용할 수 없다

ex) [BOJ 22358] 스키장

- 잘못된 코드 구현 조심!

```
int dis[V];
vector<pair<int, int>> adj[V];

void dijkstra(int S) {
    fill(dis, dis + V, MAX);
    priority_queue<pair<int, int>> pq;
    dis[S] = 0;
    pq.push({ 0, S });
    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (-d > dis[u]) continue;
        for (auto [v, c] : adj[u]) {
            if (dis[v] > dis[u] + c) {
                dis[v] = dis[u] + c;
                pq.push({ -dis[v], v });
            }
        }
    }
}
```

[BOJ 1504] 특정한 최단 경로

문제

방향성이 없는 그래프가 주어진다. 세준이는 1번 정점에서 N번 정점으로 최단 거리로 이동하려고 한다. 또한 세준이는 두 가지 조건을 만족하면서 이동하는 특정한 최단 경로를 구하고 싶은데, 그것은 바로 임의로 주어진 두 정점은 반드시 통과해야 한다는 것이다.

세준이는 한번 이동했던 정점은 물론, 한번 이동했던 간선도 다시 이동할 수 있다. 하지만 반드시 최단 경로로 이동해야 한다는 사실에 주의하라. 1번 정점에서 N번 정점으로 이동할 때, 주어진 두 정점을 반드시 거치면서 최단 경로로 이동하는 프로그램을 작성하시오.

입력

첫째 줄에 정점의 개수 N과 간선의 개수 E가 주어진다. ($2 \le N \le 800$, $0 \le E \le 200,000$) 둘째 줄부터 E개의 줄에 걸쳐서 세 개의 정수 a, b, c가 주어지는데, a번 정점에서 b번 정점까지 양방향 길이 존재하며, 그 거리가 c라는 뜻이다. ($1 \le c \le 1,000$) 다음 줄에는 반드시 거쳐야 하는 두 개의 서로 다른 정점 번호 v_1과 v_2가 주어진다. ($v_1 \neq v_2$, $v_1 \neq N$, $v_2 \neq 1$)

[BOJ 1504] 특정한 최단 경로

- 1번 정점에서 출발, N번 정점으로 가는 최단거리

- 두 정점(v1, v2)을 반드시 거쳐가야 함

- 1 → v1 → v2 → N or 1 → v2 → v1 → N

- 1번 정점, v1, v2에서 각각 다익스트라 알고리즘 수행

필수 / 연습문제

[필수문제]

[BOJ 1655] 가운데를 말해요

[BOJ 17396] 백도어

[BOJ 1504] 특정한 최단 경로

[BOJ 7662] 이중 우선순위 큐

[BOJ 19640] 화장실의 규칙

[BOJ 18223] 민준이와 마산 그리고 건우

[연습문제]

[BOJ 16211] 백채원

[BOJ 2423] 전구를 켜라

[BOJ 17833] 홍익대학교 지하캠퍼스

[BOJ 20183] 골목 대장 호석 - 효율성 2

[BOJ 16118] 달빛 여우

[BOJ 2075] N번째 큰 수