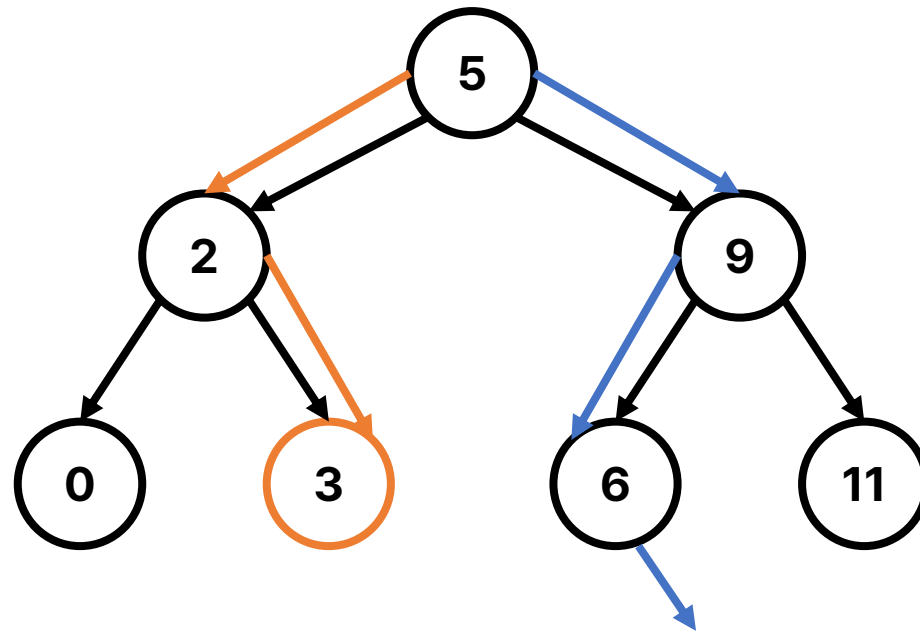


Disjoint Set, Heap

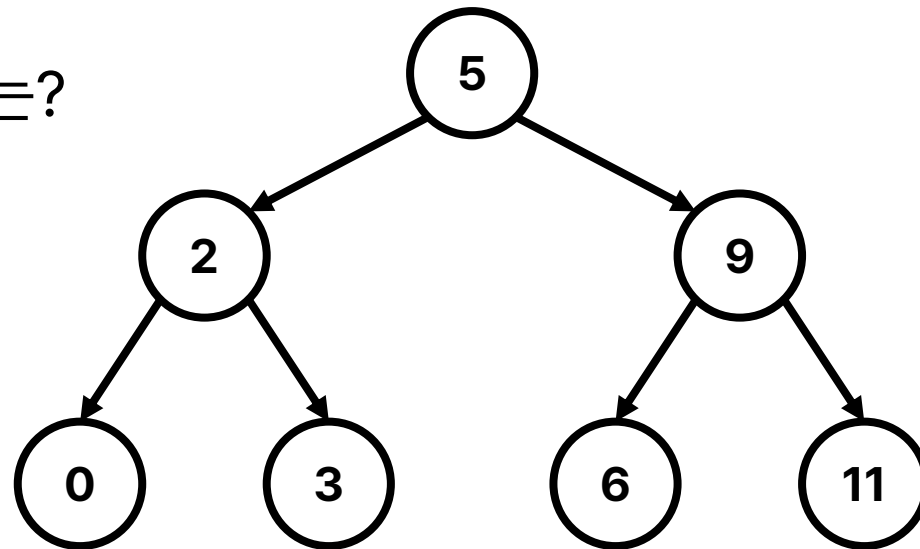
Binary Search Tree

- Binary Search의 과정을 **이진** 트리로 나타낸 것
 - 7을 찾으려면..
 - 3을 찾으려면..



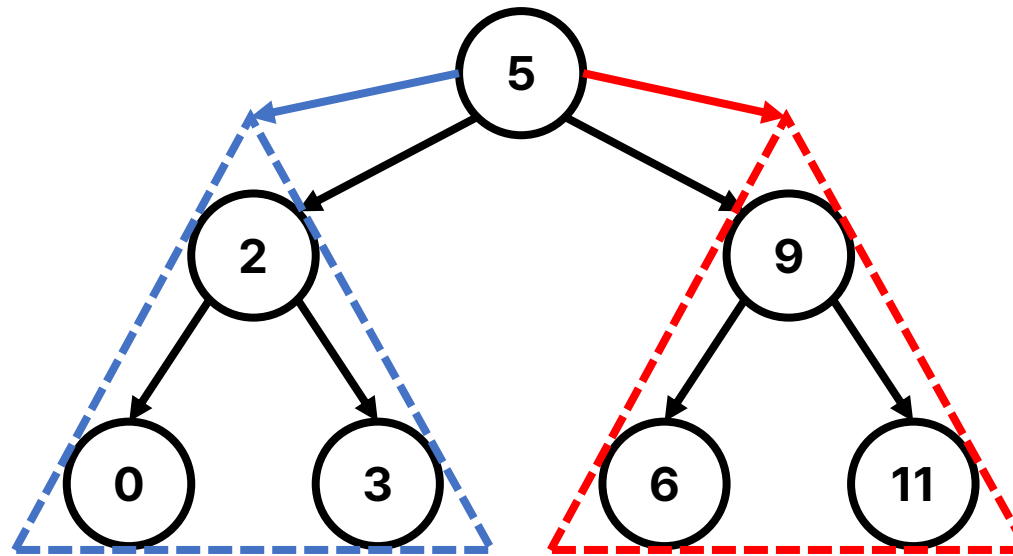
Binary Search Tree

- 가장 큰 원소는?
- 3 초과인 가장 작은 원소는?
- 5 미만인 가장 큰 원소는?



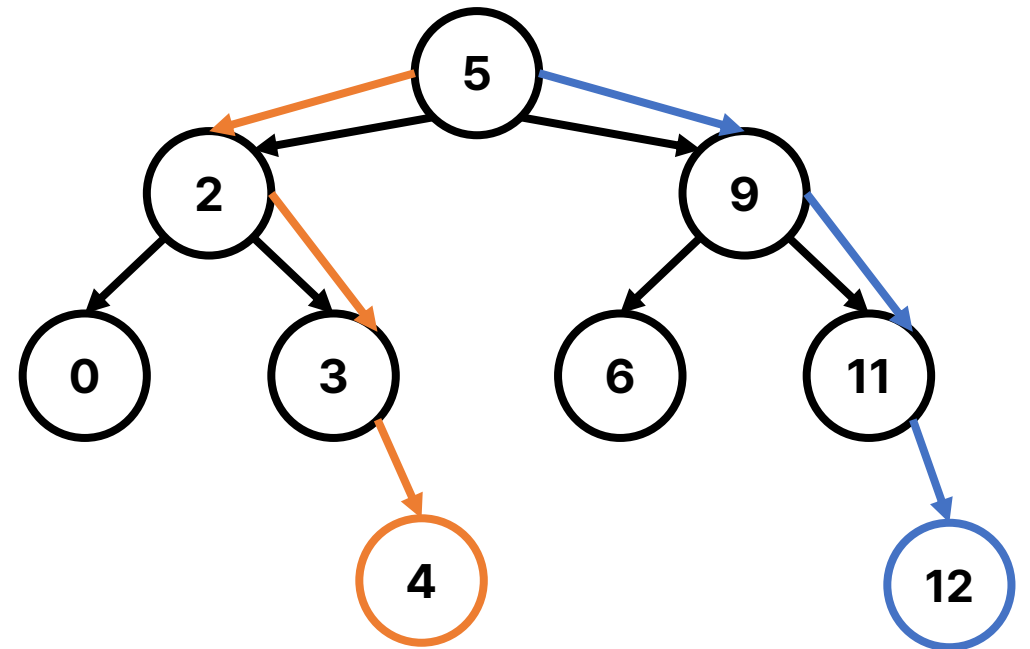
Binary Search Tree

- 서브트리의 원소와의 관계?



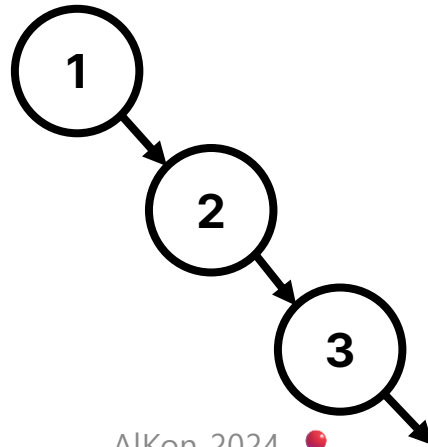
Binary Search Tree

- 원소 삽입
 - 루트부터 시작
 - 삽입하려고 하는 원소가 현재 정점의 원소보다 크면 오른쪽, 작으면 왼쪽으로 이동
 - 더 내려갈 수 없다면 그 자리에 삽입
 - 12를 넣으려면...
 - 4를 넣으려면...



Binary Search Tree

- 원소 삽입
 - 시간 복잡도: 트리의 높이를 h 라고 하면 $O(h)$
 - 최악의 경우 $h = N$ 일 수도 있다
 - Balanced Binary Search Tree(BBST, 균형 이진 탐색 트리)
 - 왼쪽 서브트리와 오른쪽 서브트리의 높이 차이가 1 이하인 이진 탐색 트리
 - Red-Black
 - AVL
 - Splay



Heap

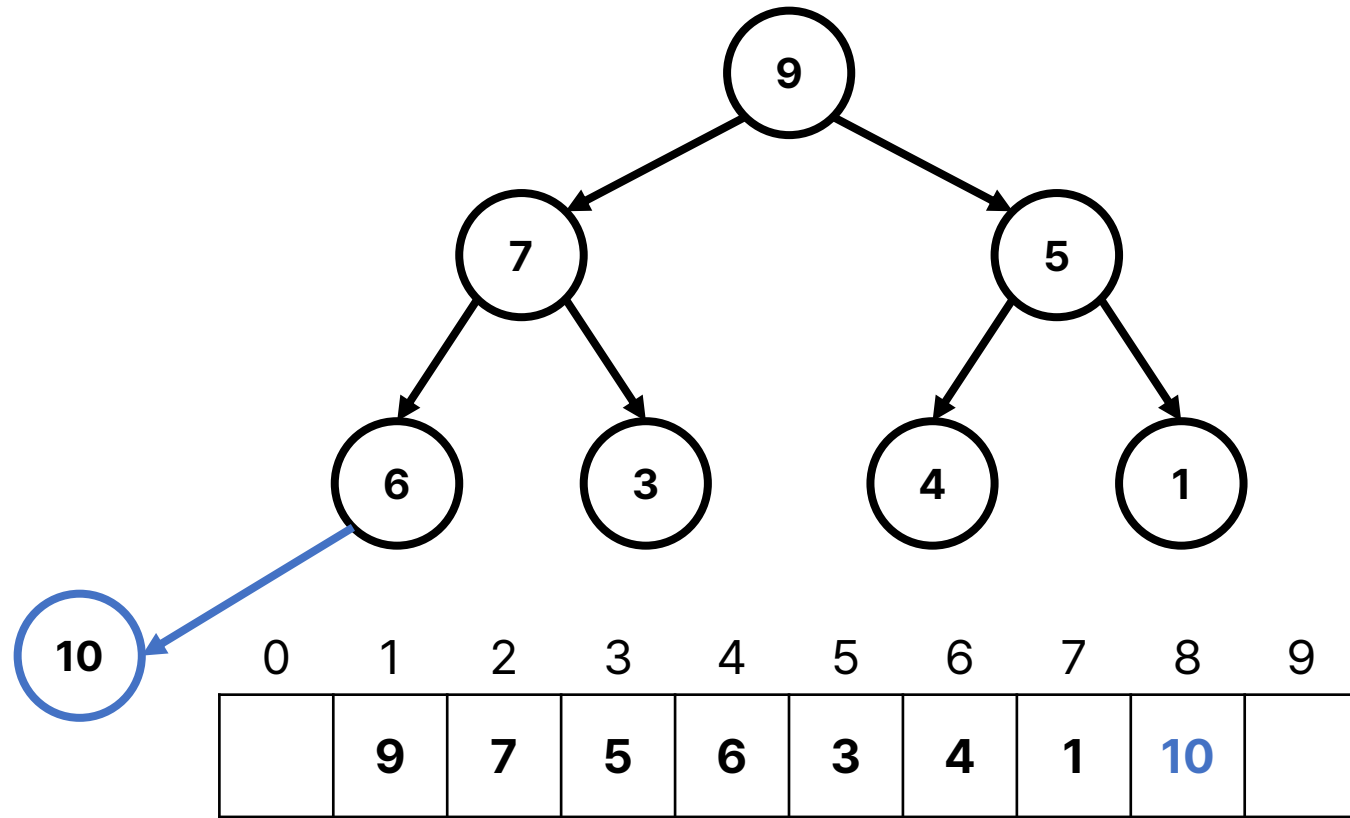
- Priority Queue
 - Queue: **먼저 들어온** 데이터가 먼저 나가는 자료구조
 - Priority Queue: **우선순위가 높은** 데이터가 먼저 나가는 자료구조
- 우선순위를 기준으로 BST를 만들고 가장 오른쪽 자손을 반환?
 - BBST는 $O(\log N)$ 이지만, 앞의 계수를 생각하면 약간 느림

Heap

- 각 정점의 값이 자식 정점보다 큰 트리 구조
 - 주로 사용되는 구조는 이진 힙 (Binary Heap)으로 완전 이진 트리(Complete Binary Tree) 형태
 - 완전 이진 트리: 마지막 레벨을 제외하고 모든 레벨이 완전히 채워져 있으며, 마지막 레벨의 모든 노드가 가능한 한 가장 왼쪽에 있는 형태
 - x 의 부모: $\frac{x}{2}$
 - x 의 왼쪽 / 오른쪽 자식: $2x, 2x + 1$

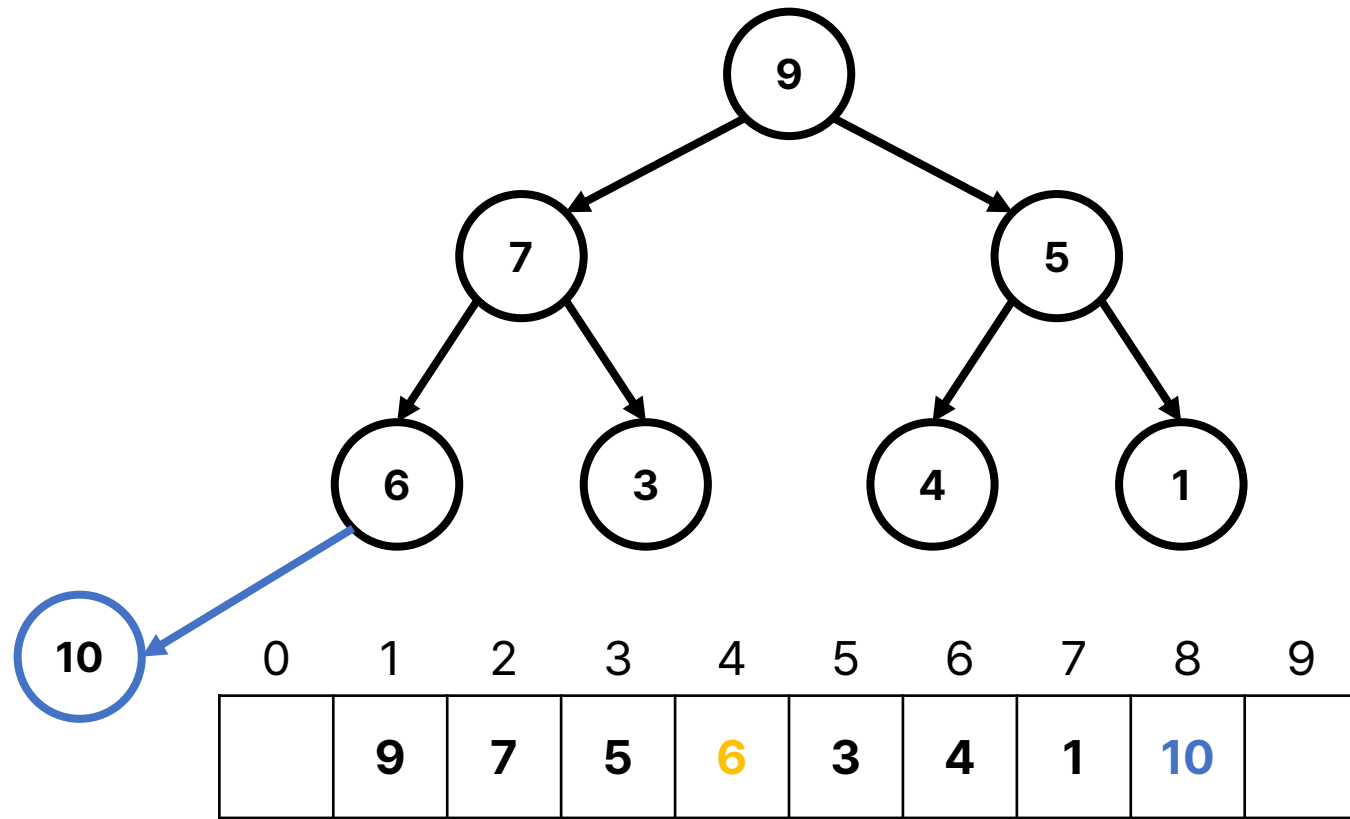
Heap

- 10 삽입



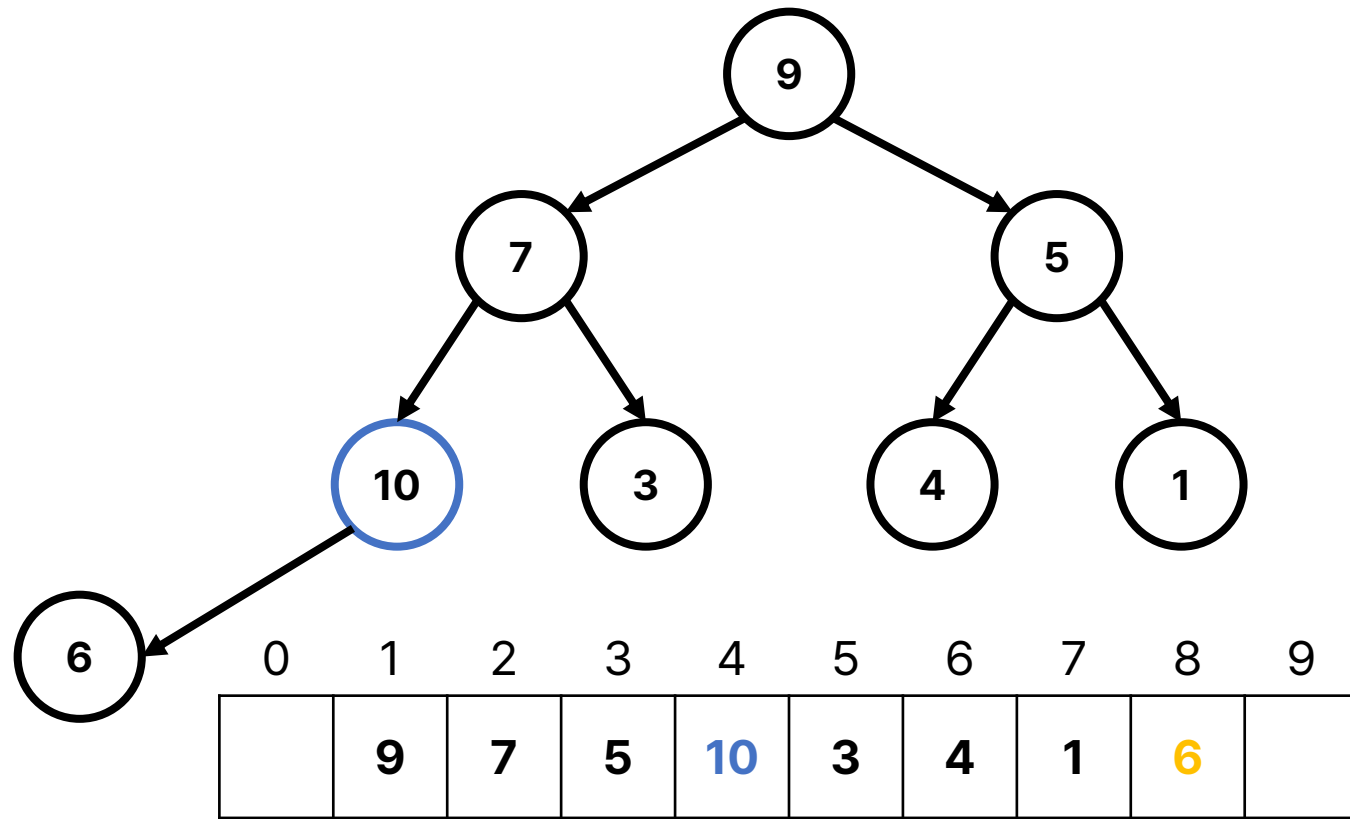
Heap

- 10 삽입



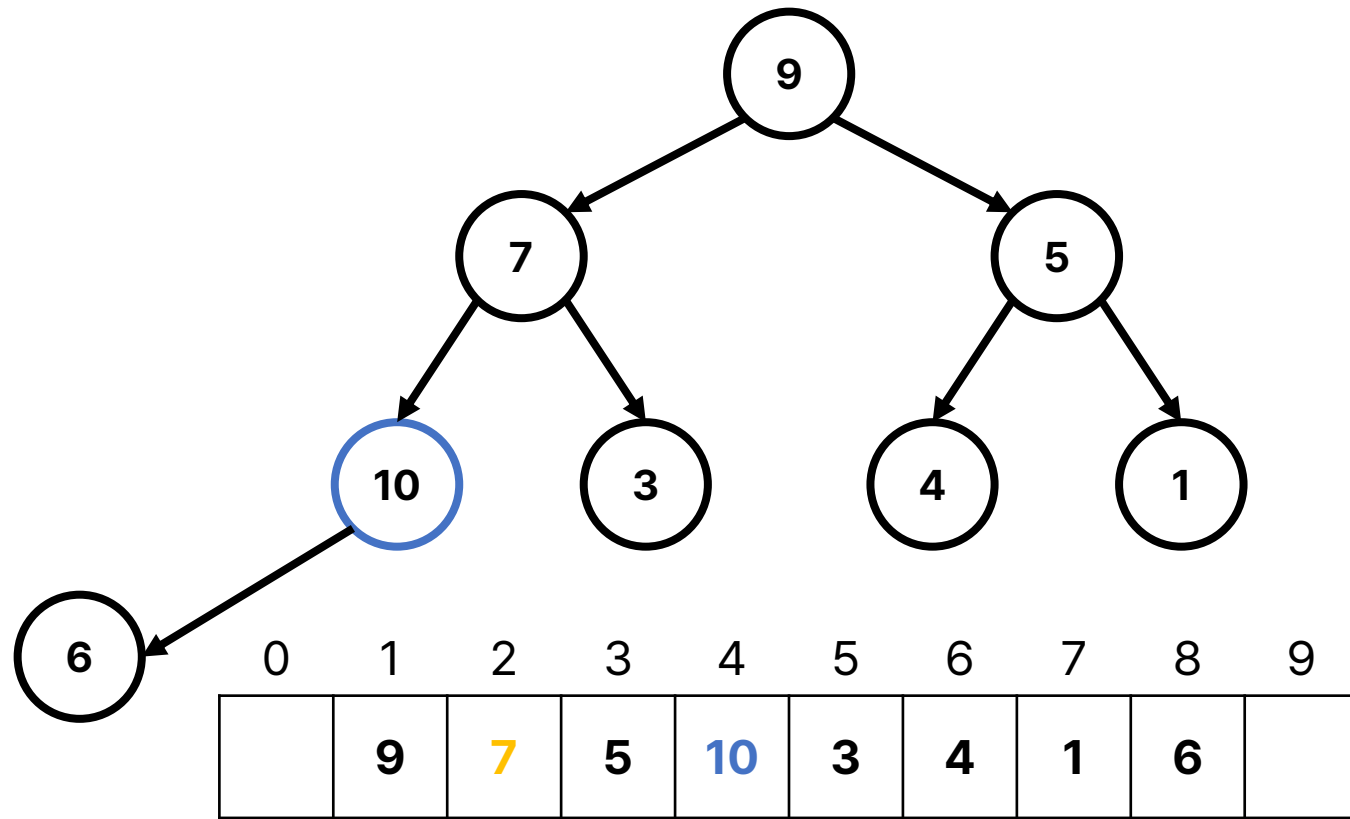
Heap

- 10 삽입



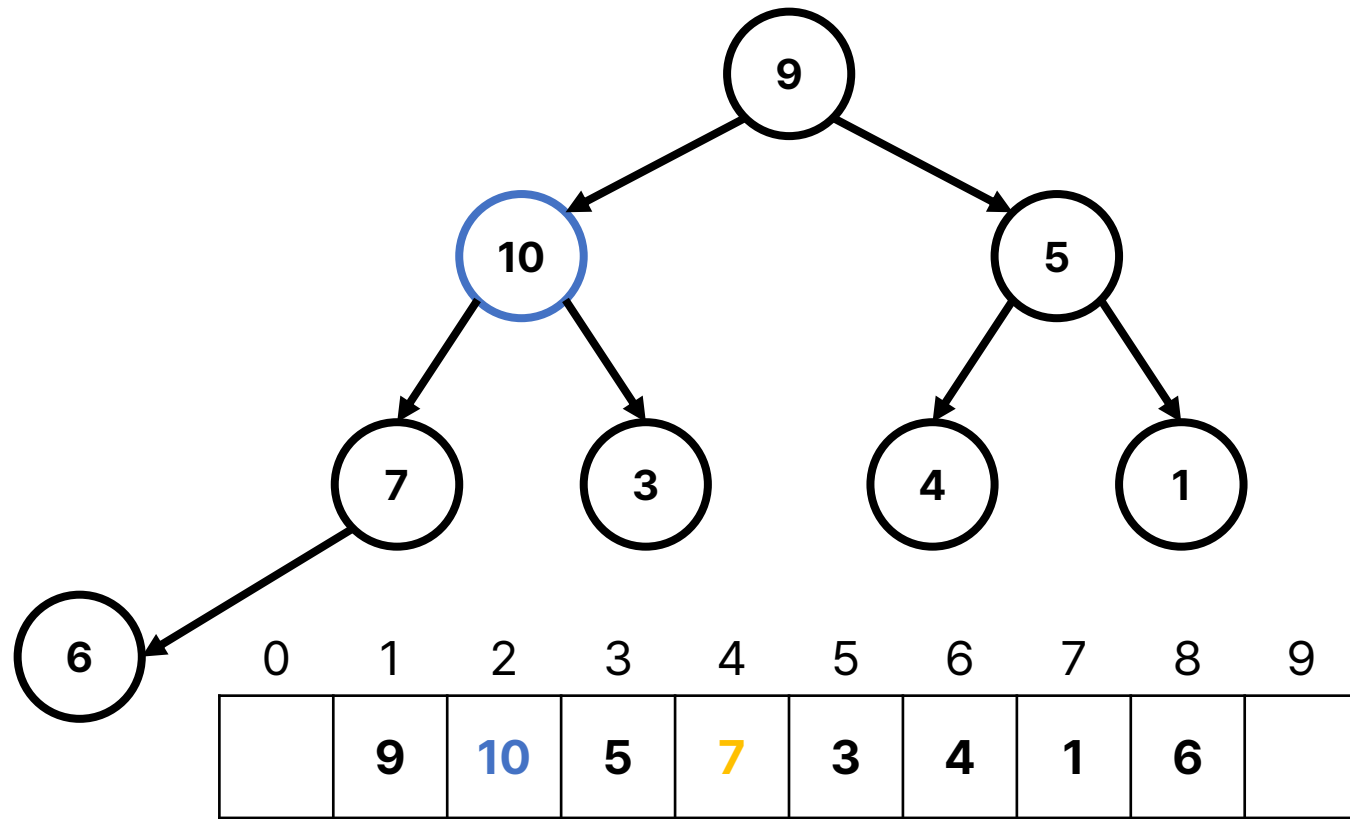
Heap

- 10 삽입



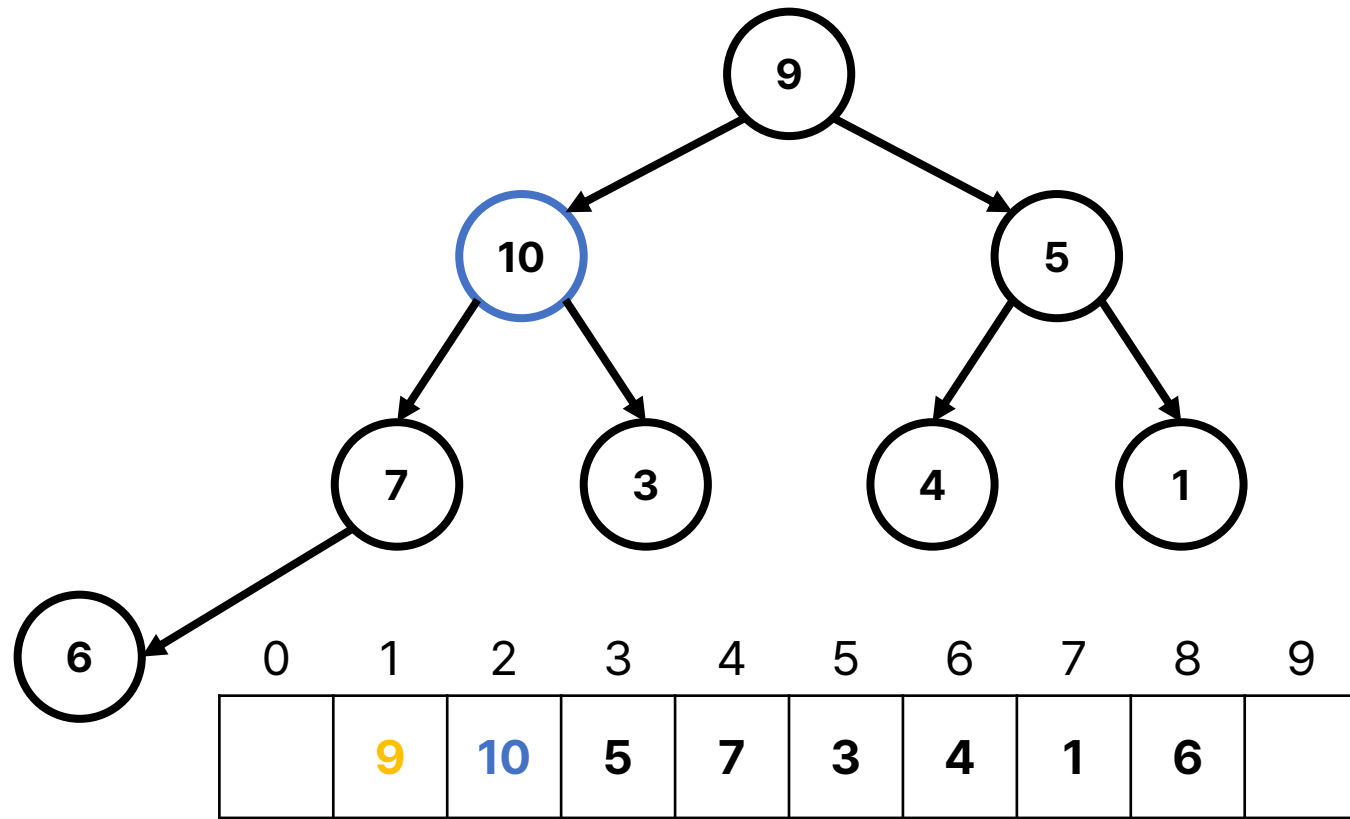
Heap

- 10 삽입



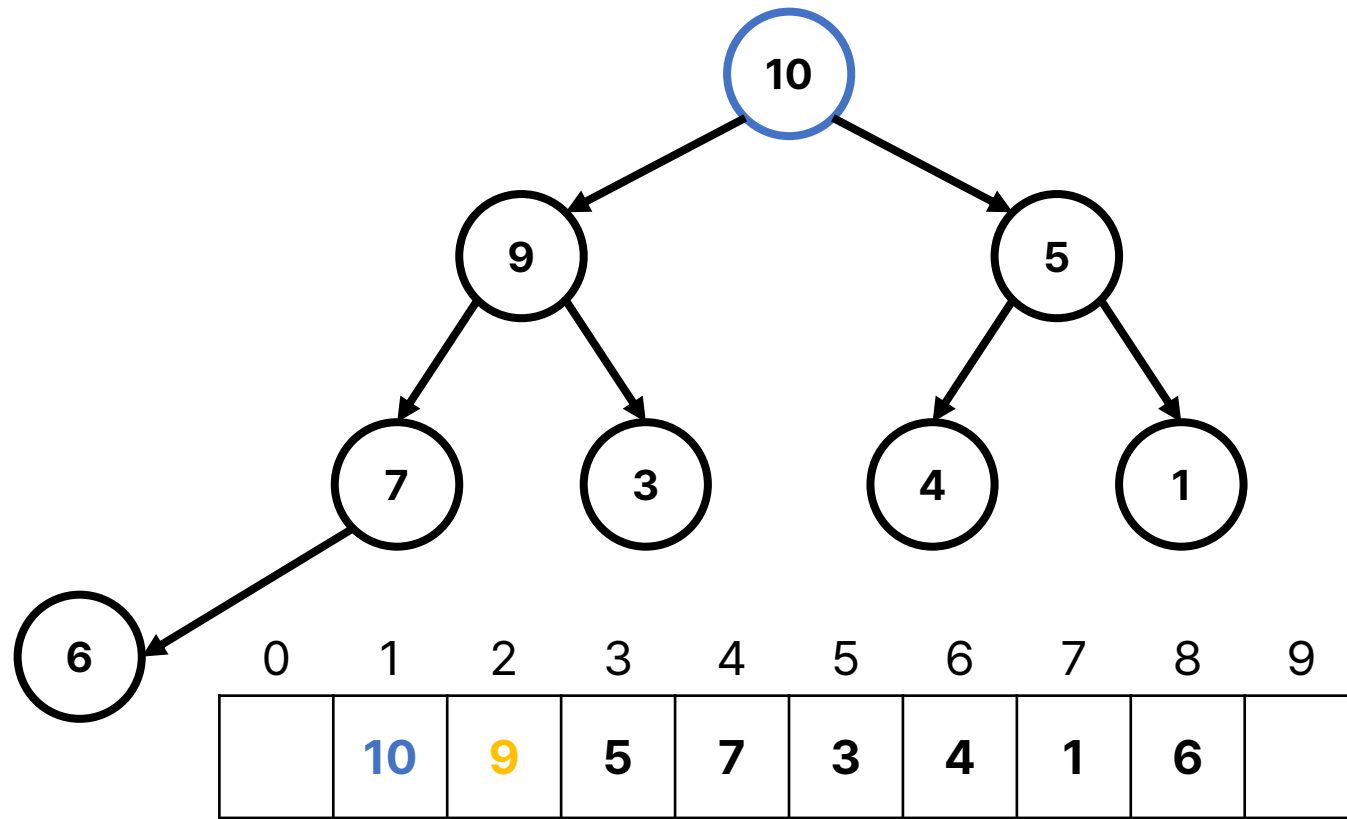
Heap

- 10 삽입



Heap

- 10 삽입



Heap

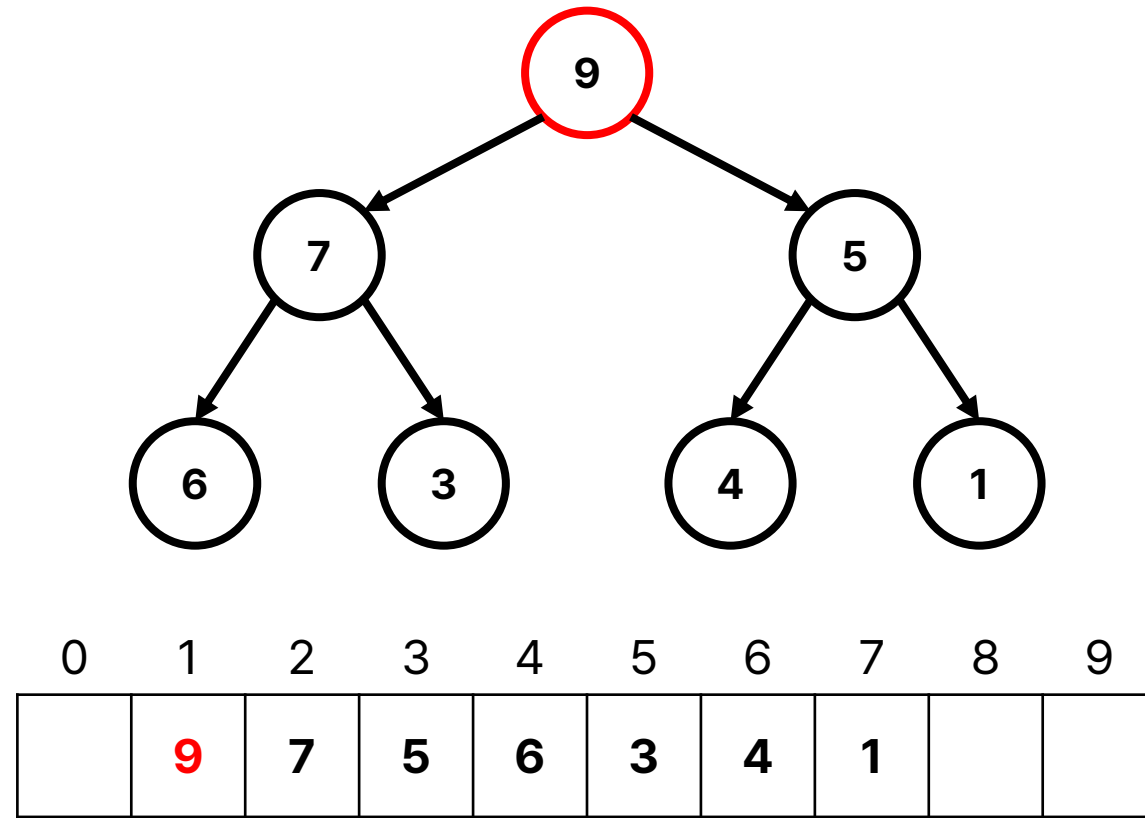
- 삽입의 시간 복잡도는 $O(\log N)$
- 가장 큰 원소 탐색은 heap의 첫 번째 원소를 반환하면 되기에 $O(1)$
- 가장 큰 원소를 제거하려면?

Heap

- 삽입의 시간 복잡도는 $O(\log N)$
- 가장 큰 원소 탐색은 heap의 첫 번째 원소를 반환하면 되기에 $O(1)$
- 가장 큰 원소를 제거하려면?
 - 루트 노드(가장 큰 원소)와 마지막 노드의 값을 바꿈
 - 마지막 노드(가장 큰 원소)를 제거
 - 루트 노드(원래 마지막 노드에 있던 것)에 있는 값이 자식보다 작으면 밑으로 내리는 것을 반복
 - 두 자식 모두 해당 노드보다 크면 더 큰 자식으로 이동

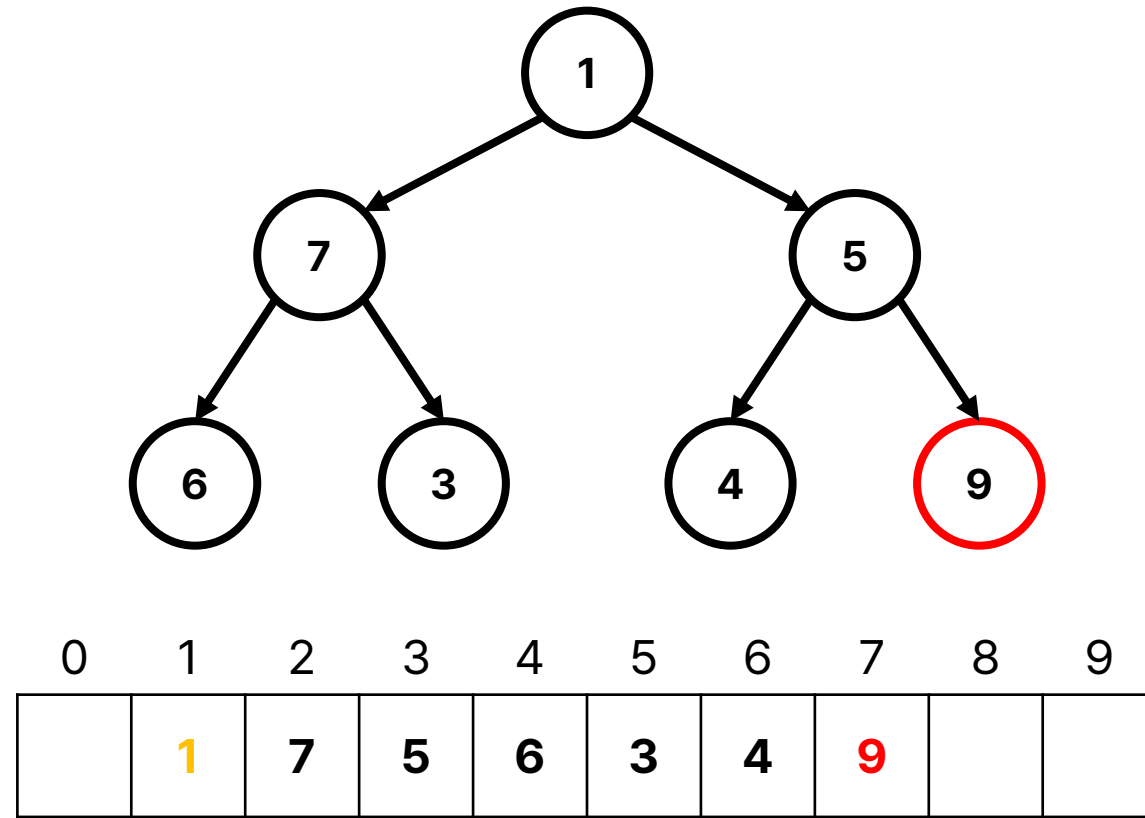
Heap

- 가장 큰 원소 제거



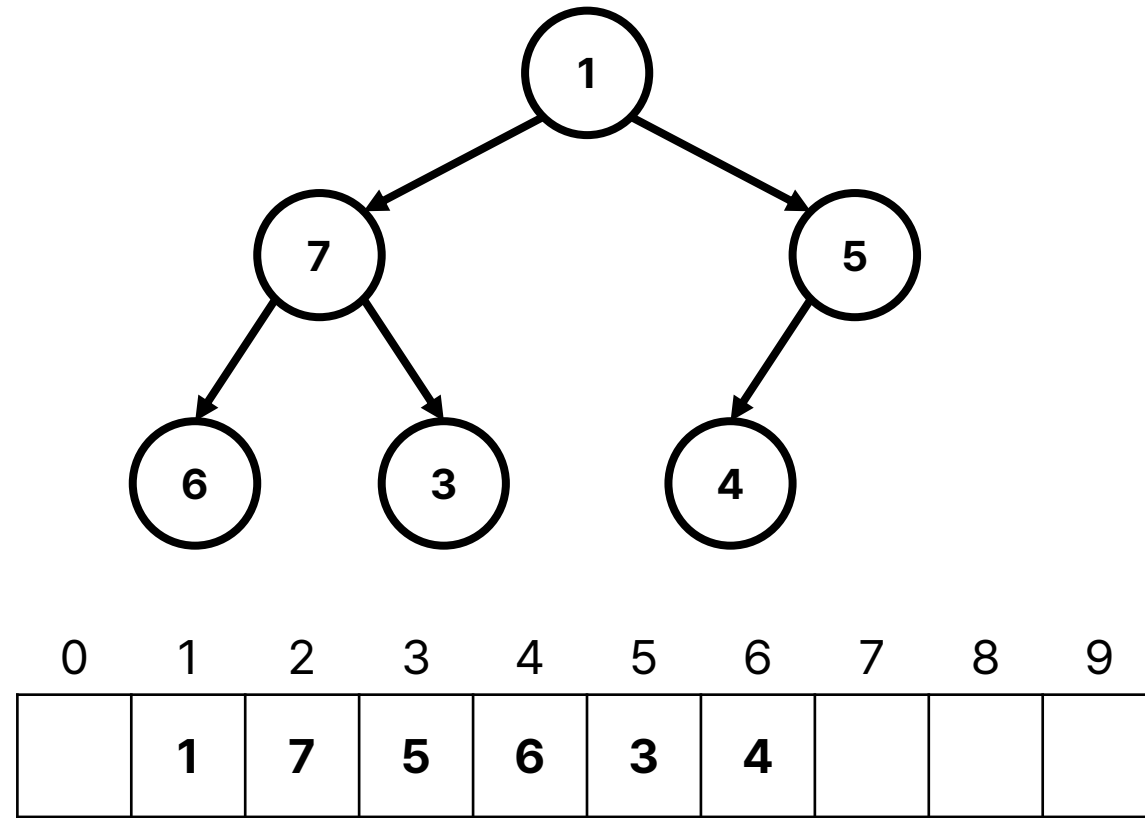
Heap

- 가장 큰 원소 제거



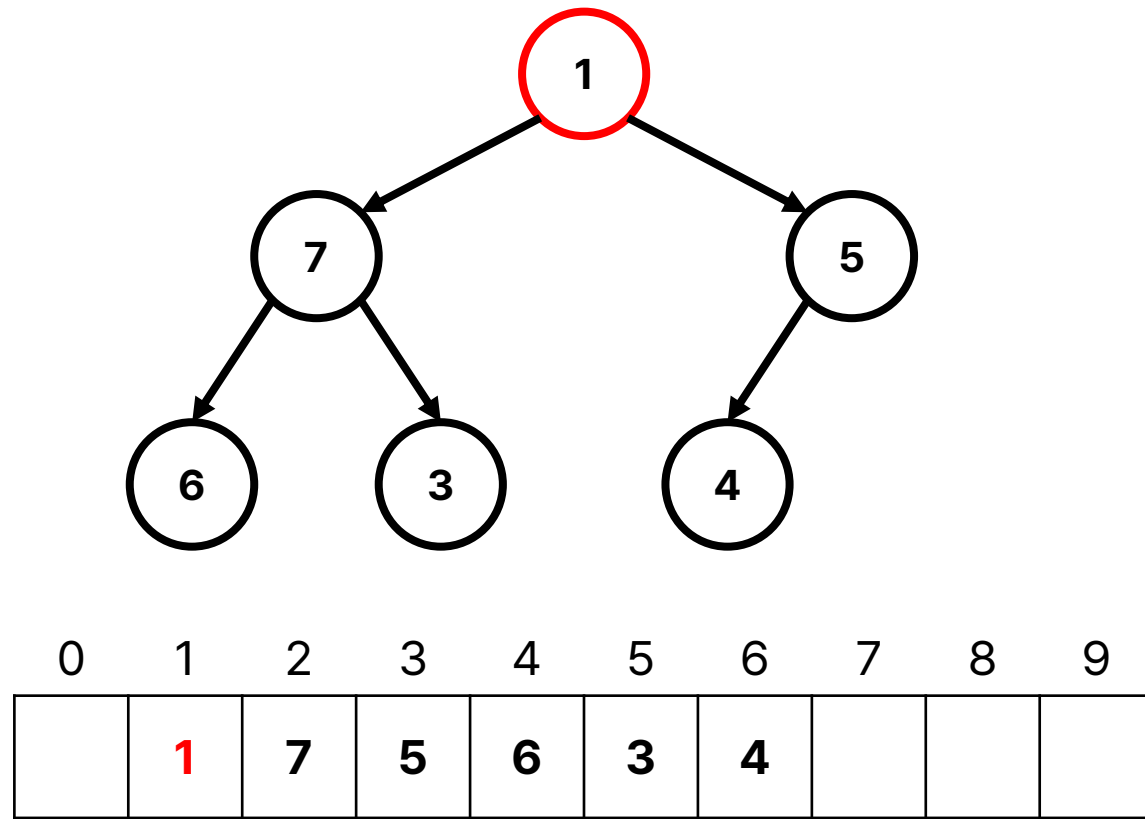
Heap

- 가장 큰 원소 제거



Heap

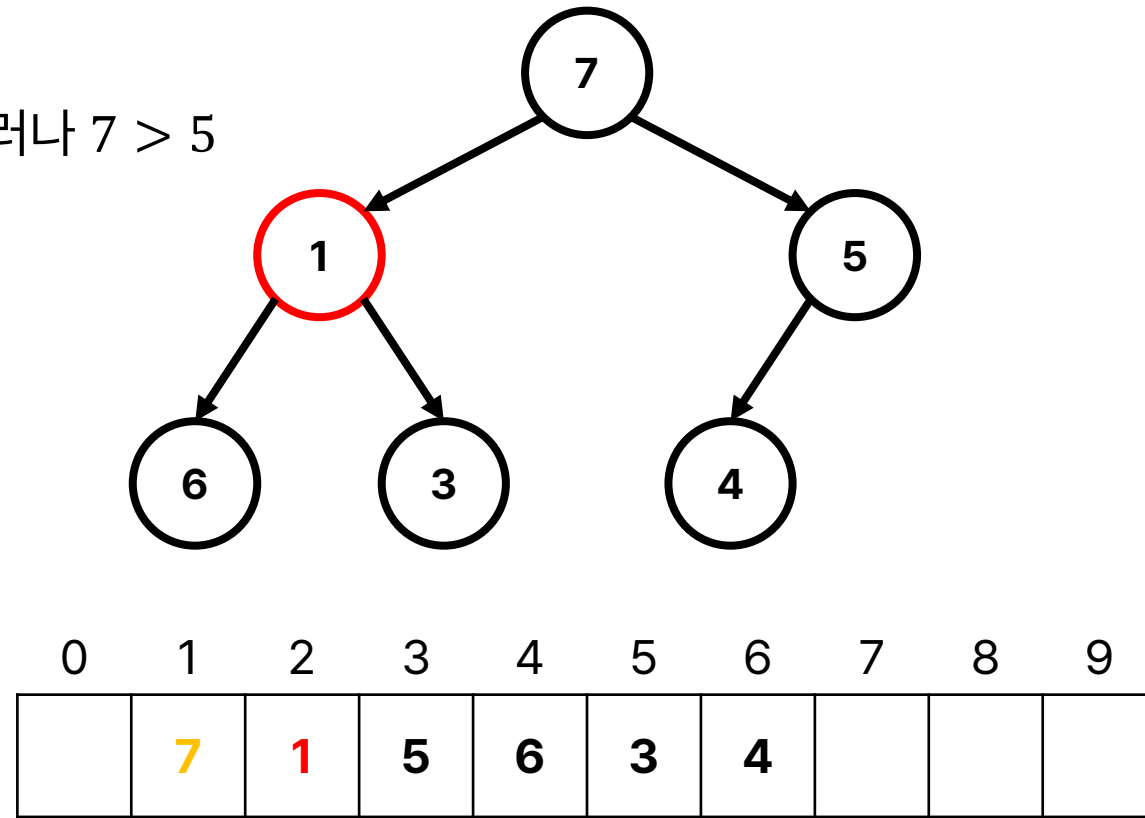
- 가장 큰 원소 제거



Heap

- 가장 큰 원소 제거

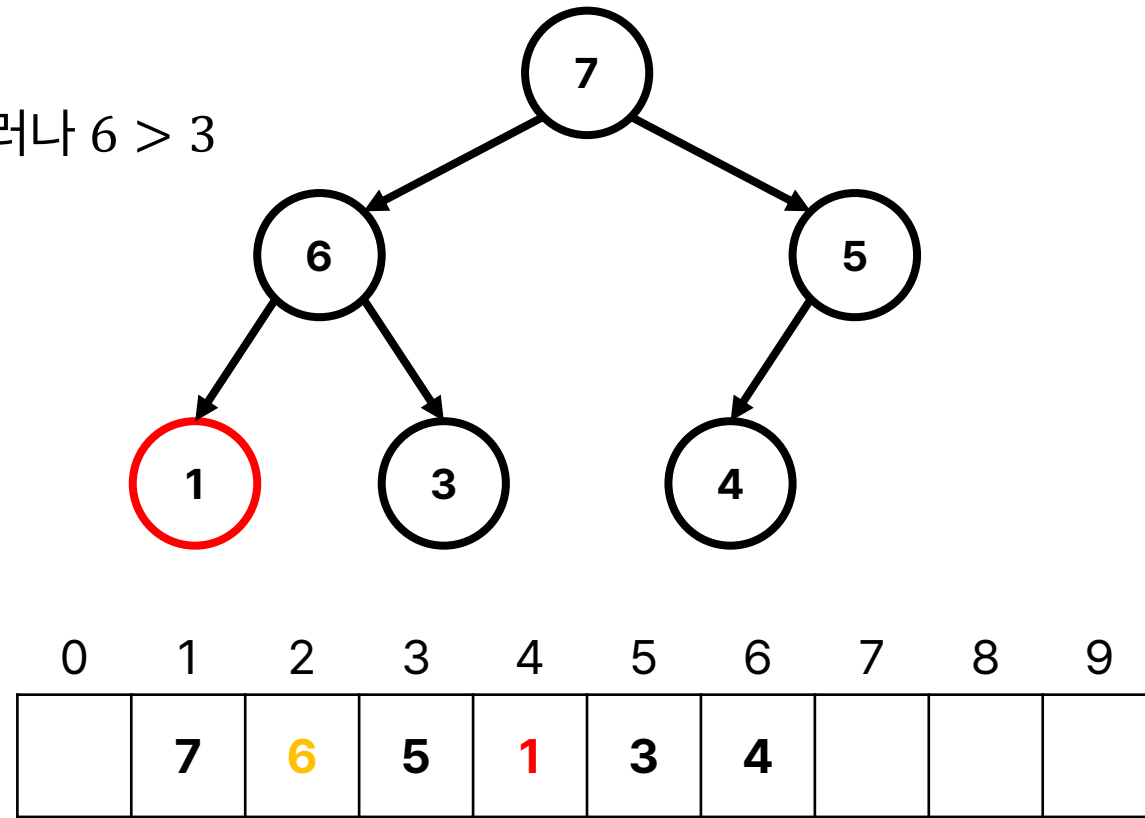
- $7 > 1, 5 > 1$ 그러나 $7 > 5$



Heap

- 가장 큰 원소 제거

- $6 > 1, 3 > 1$ 그러나 $6 > 3$



Heap

- 삽입의 시간 복잡도는 $O(\log N)$
- 가장 큰 원소 탐색은 heap의 첫 번째 원소를 반환하면 되기에 $O(1)$
- 제거의 시간 복잡도는 $O(\log N)$

Heap

- `std::priority_queue`
 - `#include <queue>`

카드 정렬하기 BOJ 1715

- 각 묶음의 카드의 수를 A, B라 하면 두 묶음을 합쳐서 하나로 만드는 데에는 $A + B$ 번의 비교를 해야 함
- 여러 묶음이 있을 때 최소 몇 번의 비교로 하나의 묶음으로 만들 수 있는지 구하여라
- (3묶음) 10, 20, 30이 있는 경우
 - 10, 20 합치고 그것과 30 합치기 $\rightarrow (10 + 20) + (30 + 30) = 90$ 번의 비교 필요
 - 20, 30 합치고 그것과 10 합치기 $\rightarrow (20 + 30) + (50 + 10) = 110$ 번의 비교 필요

카드 정렬하기 BOJ 1715

- 가장 작은 것끼리 합치는 것을 반복!
 - priority_queue에서 2개를 빼고 합친 것을 다시 넣어주자
 - 큐 안에 1개만 남을 때까지 반복하자
- 2개를 빼서 합칠 때마다 큐의 원소가 1개씩 줄어들기에 시간복잡도는 $O(N)$

Union Find

- 서로소 집합
 - 교집합이 공집합인 집합으로 구성된 집합
 - 두 집합 A, B 에 대해 $A = B$ 이거나 $A \cap B = \emptyset$
 - 각 원소가 속한 집합을 유일하게 특정할 수 있음
 - 1학년이면서 2학년일 수는 없음


Union Find

- Union Find (Disjoint set)
 - 서로소 집합을 관리하는 자료구조
 - init: 모든 원소가 자기 자신만을 원소로 하는 집합에 속하도록 초기화
 - union(u, v): u 가 속한 집합과 v 가 속한 집합을 합침
 - find(v): v 가 속한 집합을 반환
 - 위 3가지 연산을 빠르게 구현하는 것이 목표

Union Find

- Union Find
 - 트리의 형태로 구성하면?
 - init: 모든 원소가 각각 루트가 되는 트리 N 개가 생김
 - union(u, v): u 가 속한 트리의 루트를 v 가 속한 트리의 루트의 자식으로 넣어 줌
 - find(v): v 가 속한 트리의 루트 정점을 반환

Union Find



```
int parent[100'000];

void init(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i;
    }
}

int find(int v) {
    if (v == parent[v]) return v;
    else return find(parent[v]);
}

void merge(int u, int v) {
    u = find(u), v = find(v);
    if (u != v) parent[u] = v;
}
```

Union Find

- Union Find
 - 트리의 형태로 구성하면?
 - init: 모든 원소가 각각 루트가 되는 트리 N 개가 생김 $O(N)$
 - union(u, v): u 가 속한 트리의 루트를 v 가 속한 트리의 루트의 자식으로 넣어 줌 $O(h)$
 - find(v): v 가 속한 트리의 루트 정점을 반환 $O(h)$
 - 즉, 최악의 경우 union과 find연산이 각각 $O(N)$ 일 수 있다

Union Find 최적화

- union과 find의 연산이 $O(h)$ 의 시간복잡도를 가짐
 - 트리의 높이를 줄일 수 있다면 시간복잡도가 같이 줄어듦
 - union by rank
 - union by size
 - path compression
 - 3개 중 하나만 사용해도 M 번 연산했을 때 $O(M \log N)$ 이 보장됨
 - union by rank과 union by size는 $O(\log N)$
 - path compression은 amortized $O(\log N)$
 - union by rank와 path compression을 함께 사용하면 amortized $O(\log^* N)$
 - $\log^* N$: 로그 함수를 반복적으로 적용시켜서 결과 값이 1보다 같거나 작아질 때까지 걸리는 횟수
 - $\log^* N = 1 + \log^*(\log N)$

union by rank

- 높이가 낮은 트리를 높은 트리의 루트 아래로 넣는 방법
 - $\text{rank}[i]$: i 를 루트로 하는 트리 높이의 상한
 - 만약 두 트리의 높이가 동일하면 union을 하고 rank 1 증가

```
int parent[100'000];
int rank[100'000];

void init(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i;
        rank[i] = 1;
    }
}

int find(int v) {
    if (v == parent[v]) return v;
    else return find(parent[v]);
}

void merge(int u, int v) {
    u = find(u), v = find(v);
    if (u == v) return;
    if (rank[u] > rank[v])
        swap(u, v); // 랭크가 낮은 것을 u로
    parent[u] = v; // 랭크가 낮은 것을 높은 트리의 아래 넣음
    if (rank[u] == rank[v])
        rank[v]++;
}
```

union by size

- 정점이 적은 트리를 많은 트리의 루트 아래로 넣는 방법
 - $\text{size}[i]$: i 를 루트로 하는 트리의 정점 개수
 - u 를 v 밑으로 넣으면 $\text{size}[v]$ 는 $\text{size}[u]$ 만큼 증가

```
int parent[100'000];
int size[100'000];

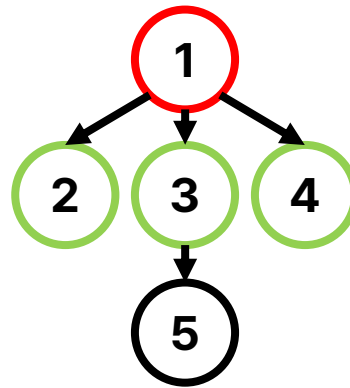
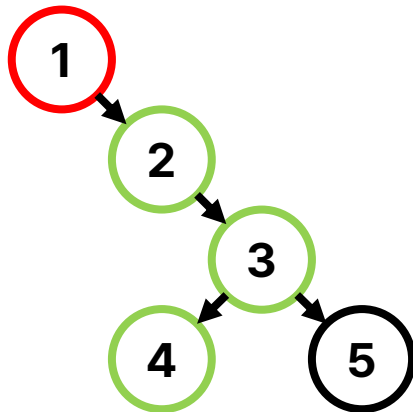
void init(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i;
        size[i] = 1;
    }
}

int find(int v) {
    if (v == parent[v]) return v;
    else return find(parent[v]);
}

void merge(int u, int v) {
    u = find(u), v = find(v);
    if (u == v) return;
    if (size[u] > size[v])
        swap(u, v); // 정점이 적은 것을 u로
    parent[u] = v; // 정점이 적은 것을 높은 트리의 아래 넣음
    size[v] += size[u];
}
```

path compression

- 경로 압축
 - find를 통해 루트를 찾았다면
 - 루트까지의 경로 상에 있는 모든 정점들을 루트의 바로 밑으로 붙임



```
int parent[100'000];

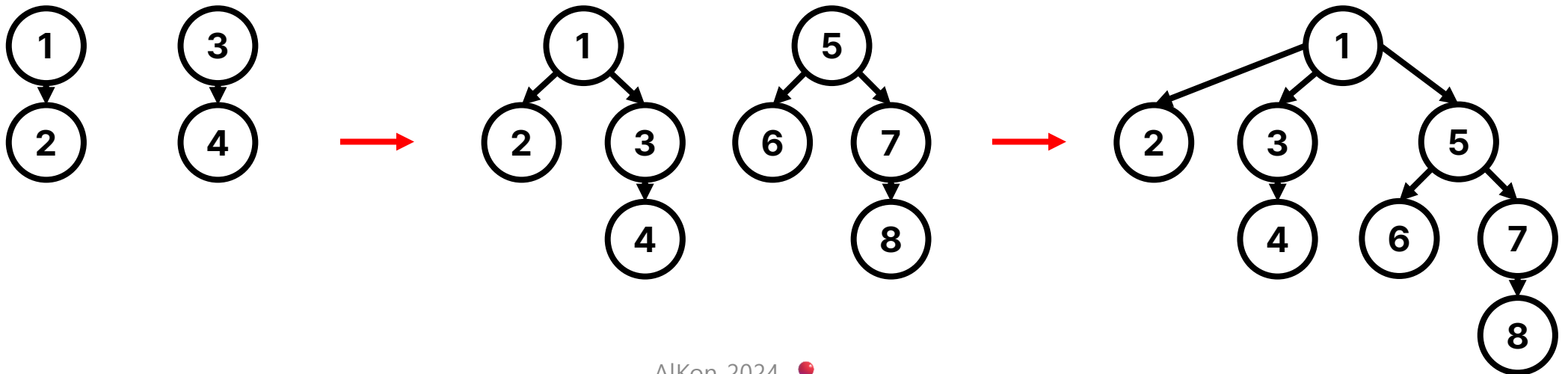
void init(int n) {
    for (int i = 1; i <= n; i++) {
        parent[i] = i;
    }
}

int find(int v) {
    if (v == parent[v]) return v;
    else return parent[v] = find(parent[v]);
}

void merge(int u, int v) {
    u = find(u), v = find(v);
    if (u != v) parent[u] = v;
}
```

union by rank & size

- 이게 왜 $O(\log N)$ 이 되는 걸까?
 - union by rank부터 생각해보자
 - 하나의 트리를 다른 트리에 합칠 때, 항상 최적화된(연산 횟수가 최대한 늘어나지 않는) 형태로 합쳐진다
 - 아래의 경우가 가장 적은 노드로 가장 높은 트리를 만들 수 있는 형태: 노드의 개수 = 2^{rank-1}



union by rank & size

- 이게 왜 $O(\log N)$ 이 되는 걸까?
 - 즉, 모든 rank는 $\lceil \log 2N \rceil$ 이하이다
 - 그렇기에 union by rank의 find의 시간복잡도가 $O(\log N)$ 이 된다.
 - union은 find의 시간복잡도에 종속됨 (find(u), find(v)를 하고 $O(1)$ 의 연산을 진행)

union by rank & size

- 이게 왜 $O(\log N)$ 이 되는 걸까?
 - union by size는 이를 반대로만 해주면 됨
 - size별로 최대 rank가 결정되고 그 결론은 union by rank와 동일함

union by rank & size + path compression

- find 함수의 시간복잡도가 $O(\log^* N)$ 이 됨
 - 이는 아커만(Ackermann) 함수와 관련 있음
 - $N = 2^{65536}$ 에서 아커만 함수의 값이 5이므로 사실상 상수 시간복잡도를 가진다고 봐도 된다
 - 자세한 증명은 <https://codeforces.com/blog/entry/98275> 여기서 확인

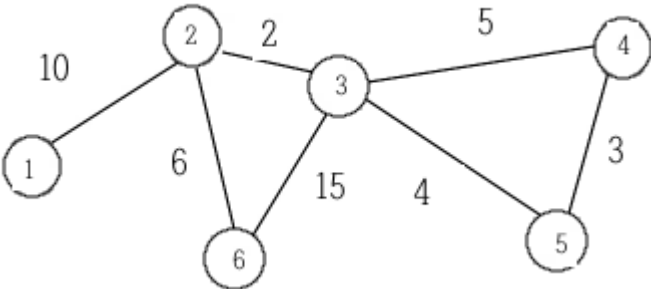
사이클 게임 BOJ 20040

- 평면 상의 점 N 개가 있다 (세 점이 일직선 위에 놓이지 않는다)
- 매 차례에 플레이어는 i 번 점과 j 번 점을 연결하는 선분을 긋는다
 - 이전에 그린 선분을 다시 그을 수 없다
 - 다른 선분과 교차는 가능하다
- 사이클이 몇 번째 차례에 생기는지 구하여라

사이클 게임 BOJ 20040

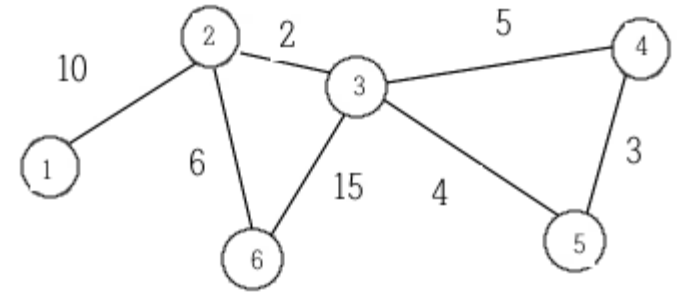
- 평면 상의 점 N 개가 있다 (세 점이 일직선 위에 놓이지 않는다)
- 매 차례에 플레이어는 i 번 점과 j 번 점을 연결하는 선분을 긋는다
 - 이전에 그린 선분을 다시 그을 수 없다
 - 다른 선분과 교차는 가능하다
- 사이클이 몇 번째 차례에 생기는지 구하여라
 - 사이클이 생겼다는 것은 열려 있던 것을 닫았다는 것이다
 - 즉, 같은 집합에 있던 점을 다시 연결하였을 때, 사이클이 생성된다

비용 BOJ 2463

- 가중치 그래프가 주어졌을 때, 두 정점 사이의 경로가 있으면 그래프에서 가중치가 가장 작은 간선부터 제거해 나가는 것을 두 정점 사이의 경로가 없을 때까지 반복한다
 - 이때 드는 비용은 제거한 간선들의 가중치 합이다
 - $u < v$ 인 모든 두 정점에 대한 비용의 합을 구하여라
- 
- 오른쪽 그림에서 $\text{cost}(2, 6) = 2 + 3 + 4 + 5 + 6 = 20$ 이다

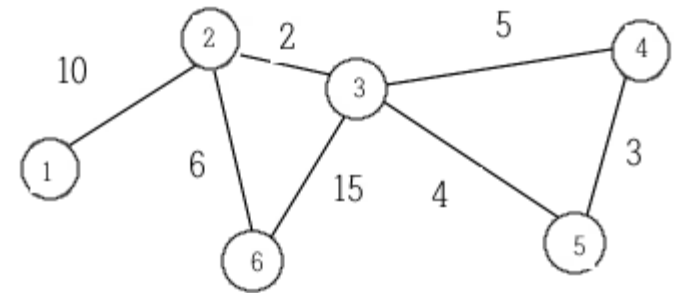
비용 BOJ 2463

- Union Find에서 제거하는 연산이 있었는가?
- 합치는 것이 편함
 - 반대로 생각해보자
 - 모든 것이 연결된 상태로 제거를 해가는 것이 아닌 합쳐간다면?
 - 이렇게 해도 답에는 변함이 없을까?
 - 크기가 큰 것부터 합쳐가면 크기가 작은 것부터 제거하는 것의 반대 순서와 동일함!



비용 BOJ 2463

- 만약 1, 2번 정점을 잇는 간선이 제거되었다고 하자
 - 1번과 연결된 모든 정점들의 경로도 사라진다
 - 여태까지 제거한 간선들의 가중치의 합과 두 component의 크기를 곱한 만큼을 답에 더하면 된다
 - 지나다니는 골목이 끊어졌으니 해당 길을 지나는 경우의 수를 곱해주는 것이다
 - component의 크기를 같이 관리하기 위해서 union by size로 합쳐주자



Offline Query

- 방금 문제를 푼 방식이다!
 - 쿼리 문제를 푸는 방식은 **Online Query**와 **Offline Query**가 있다
 - Online Query의 경우는 쿼리가 들어오면 **바로** 처리해서 결과를 반환해준다 (평소에 풀던 방식일 것이다)
 - Offline Query의 경우는 쿼리를 한 번에 받은 다음, **원하는 순서**로 처리하여 답을 구하고 쿼리가 들어온 순서대로 답을 출력해주는 방식이다
 - 보통 트리 혹은 그래프에서 간선을 끊어가면서 진행하는 문제는 이 Offline Query 방식을 사용하여 합치는 연산으로 진행하는 것이 편한 경우가 많다 (Union Find와 함께)
 - Offline Query는 나중에 sqrt decomposition, Mo's Algorithm 등의 알고리즘에서도 사용한다

연습 문제

1715 : 카드 정렬하기

20040 : 사이클 게임

2463 : 비용

References

- <https://github.com/justiceHui/SSU-SCCC-Study>
- <https://codeforces.com/blog/entry/98275>