2022-1학기 스터디 #9

나정휘
https://justiceHui.github.io/

목차

- Binary Search Tree
- Heap
- Union-Find (Disjoint Set)

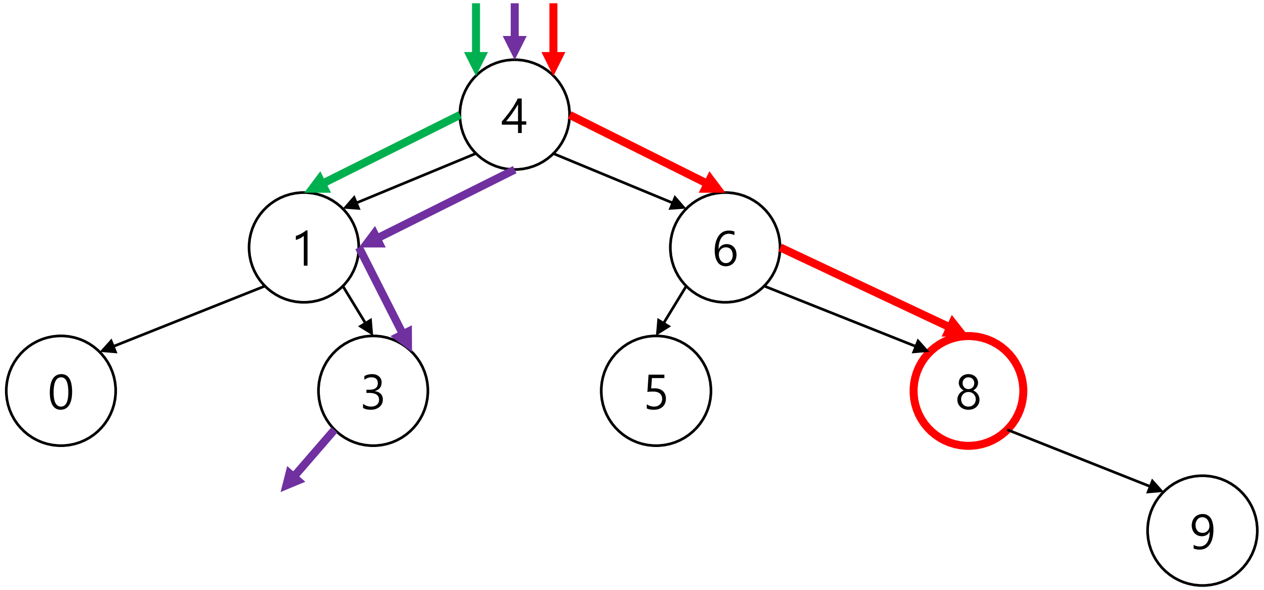
Binary Search Tree

Binary Search Tree

- 이진 탐색
  - 정렬된 구간 [l, r]에서 어떤 값 x가 있는지 확인하는 알고리즘
    - 중간 지점 m = (l+r)/2를 잡고
    - A[m] = x이면 탐색 완료
    - A[m] < x이면 x는 m보다 오른쪽에 있음 : l = m + 1
    - A[m] > x이면 x는 m보다 왼쪽에 있음 : r = m – 1
  - A = {0, 1, 3, 4, 5, 6, 8, 9}, x = 8
    - l = 0, r = 7, m = 3, A[m] = 4
    - l = 4, r = 7, m = 5, A[m] = 6
    - l = 6, r = 7, m = 6, A[m] = 8, 종료
  - x = 1
    - l = 0, r = 7, m = 3, A[m] = 4
    - l = 0, r = 2, m = 1, A[m] = 1, 종료

Binary Search Tree

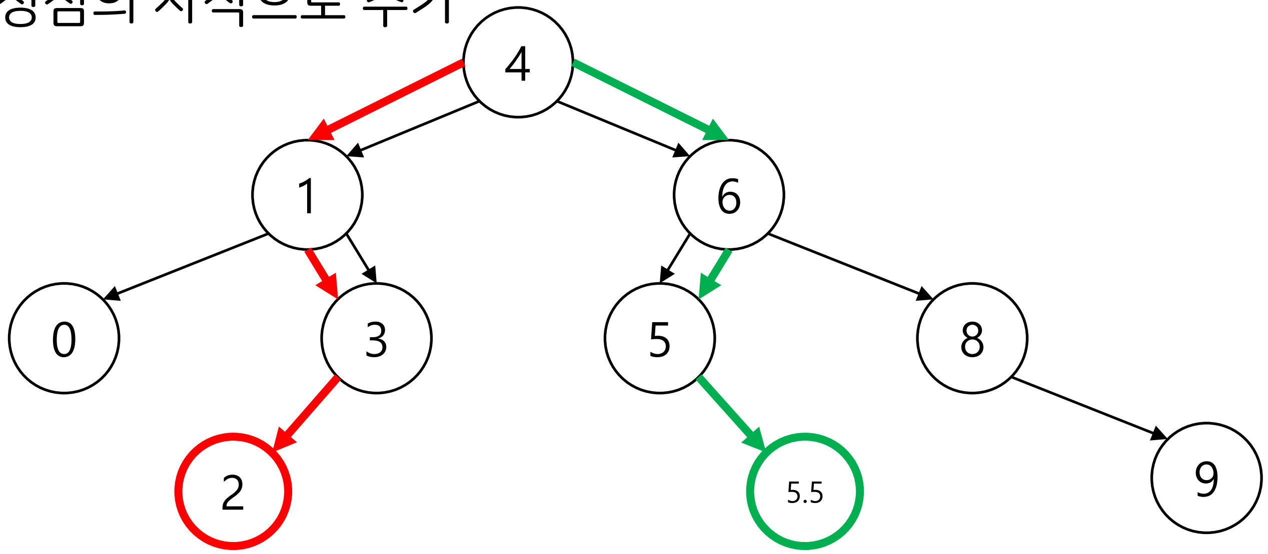
- 이진 탐색 트리
  - 이진 탐색 과정을 이진 트리로 나타낸 것
    - x = 8
    - x = 1
    - x = 2
  - 할 수 있는 연산
    - 트리에 원소 삽입/삭제
    - 어떤 원소가 있는지 확인
    - 가장 큰/작은 원소
    - x 이상/초과인 가장 작은 원소
    - x 이하/미만인 가장 큰 원소
    - 모든 원소를 오름차순으로 순회



Binary Search Tree

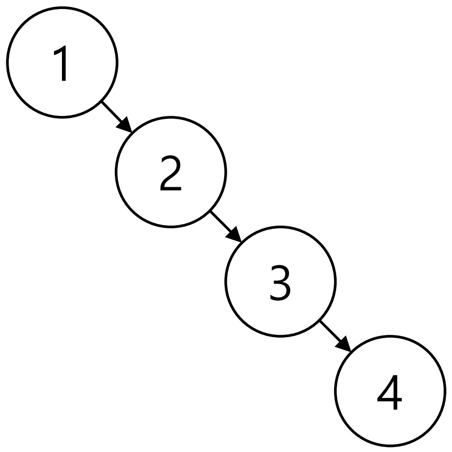
- 원소 삽입
  - 루트부터 시작
  - 현재 정점의 원소보다 크면 오른쪽 / 작으면 왼쪽으로 이동
  - 더 이상 내려갈 수 없을 때까지 이동
  - 마지막으로 방문한 정점의 자식으로 추가

- INSERT 2
- INSERT 5.5



Binary Search Tree

- 원소 삽입
  - 시간 복잡도: 트리의 높이를 h라고 하면 O(h)
  - 최악의 경우 O(N)
    - INSERT 1
    - INSERT 2
    - INSERT 3
    - …
  - 최선의 경우 O(log N)



Binary Search Tree

- 원소 삭제
  - 삭제해야 할 정점 x를 찾음
  - 리프 정점이면 그냥 삭제
  - 자식 정점이 1개면 x의 부모와 자식을 연결하고 x 삭제
  - 자식 정점이 2개면
    - 오른쪽 서브 트리에서 가장 작은 원소 m을 찾음 (x보다 크면서 가장 작은 원소)
    - m을 x 자리로 옮기고 m을 삭제 (m은 자식 정점이 1개 이하)

  - 시간 복잡도: O(h)

질문?

Binary Search Tree

- 가장 큰/작은 원소
  - 가장 큰 원소: 루트에서 시작해서 오른쪽 자식으로 계속 내려가면 됨
  - 가장 작은 원소: 루트에서 시작해서 왼쪽 자식으로 계속 내려가면 됨

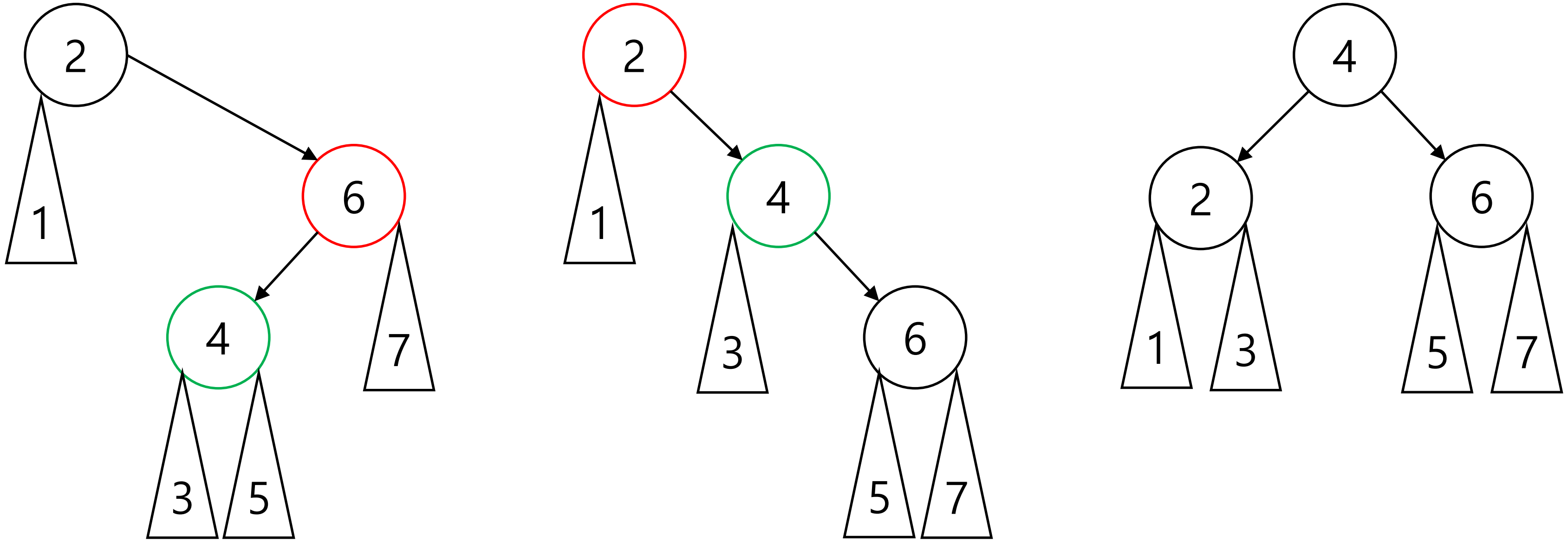
  - x보다 큰 원소 중 가장 작은 원소
    - x의 오른쪽 자식이 존재하면 오른쪽 서브 트리에서 가장 작은 원소
    - x의 오른쪽 자식이 없으면 x의 조상을 따라가면서 처음으로 만나는 x보다 큰 원소
      - 현재 정점이 왼쪽 자식일 때까지 부모 정점을 따라가면 됨
  - x보다 작은 원소 중 가장 큰 원소
    - 반대로 하면 됨

Binary Search Tree

- 중위 순회 (Inorder Traversal)
  - 이진 탐색 트리의 중위 순회는 원소를 오름차순으로 순회함
    - 왼쪽 서브 트리(자신보다 작은 원소)를 모두 순회한 뒤
    - 루트(자신)을 보고
    - 오른쪽 서브 트리(자신보다 큰 원소)를 모두 순회함

Binary Search Tree

- Balanced Binary Search Tree
  - 단순하게 구현하면 최악의 경우 O(N)을 피할 수 없음
  - BBST : 높이를 O(log N)으로 유지하는 BST
    - 대충 이런 느낌으로 BST 성질을 만족시키면서 트리를 바꿈
    - AVL Tree, Red Black Tree, Treap, Splay Tree, …



질문?

Binary Search Tree

- std::set
  - #include <set>
  - 집합을 관리하는 컨테이너
    - 원소의 중복을 허용하지 않음 (std::multiset은 중복 허용)
    - 원소를 오름차순으로 관리
    - 원소 삽입/삭제/검색
    - 가장 큰/작은 원소
    - x 이상인 가장 작은 원소 / x 초과인 가장 작은 원소
    - 오름차순/내림차순 순회
  - 보통 Red Black Tree로 구현되어 있음
    - 모든 연산은 O(log N)에 동작
    - 오름차순/내림차순 순회는 O(N)

Binary Search Tree

- std::map
  - #include <map>
  - key-value 쌍을 관리하는 컨테이너
    - key의 중복을 허용하지 않음
    - value 변경 가능
    - key에 대해서 std::set의 모든 연산 수행 가능

Binary Search Tree – 예시 1

- BOJ 1822 차집합
  - 집합 A, B가 주어지면 A – B의 원소를 오름차순으로 출력하는 문제

  - std::set 사용하면 됨
    - s.count(t) 대신 s.find(t) != s.end() 가 더 좋은 듯
    - count는 multiset에서 시간 복잡도가 원소 개수에 비례함

```cpp
#include <bits/stdc++.h>
using namespace std;

int main(){
    ios_base::sync_with_stdio(0); cin.tie(0);
    int a, b; cin >> a >> b;
    set<int> s;
    for(int i=0; i<a; i++){
        int t; cin >> t; s.insert(t);
    }
    for(int i=0; i<b; i++){
        int t; cin >> t;
        if(s.count(t)) s.erase(t);
    }
    cout << s.size() << "\n";
    for(auto i : s) cout << i << " ";
}
```

Binary Search Tree – 예시 2

- BOJ 20920 영단어 암기는 괴로워
  - N개의 단어가 주어지면 특정 기준으로 정렬해야 함
    - 많이 등장한 문자열을 앞에 배치
    - 길이가 긴 단어를 앞에 배치
    - 사전 순으로 앞에 있는 단어를 앞에 배치
  - std::map 사용해서 단어의 등장 횟수를 셀 수 있음
    - 중복 제거도 map으로 할 수 있음

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M;
vector<string> V;
map<string, int> C;

bool Compare(const string &a, const string &b){
    if(C[a] != C[b]) return C[a] > C[b];
    if(a.size() != b.size()) return a.size() > b.size();
    return a < b;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M; V.resize(N);
    for(auto &i : V) cin >> i, C[i]++;
    sort(V.begin(), V.end());
    V.erase(unique(V.begin(), V.end()), V.end());
    sort(V.begin(), V.end(), Compare);
    for(const auto &i : V) if(i.size() >= M) cout << i << "\n";
}
```

질문?

Heap

Heap

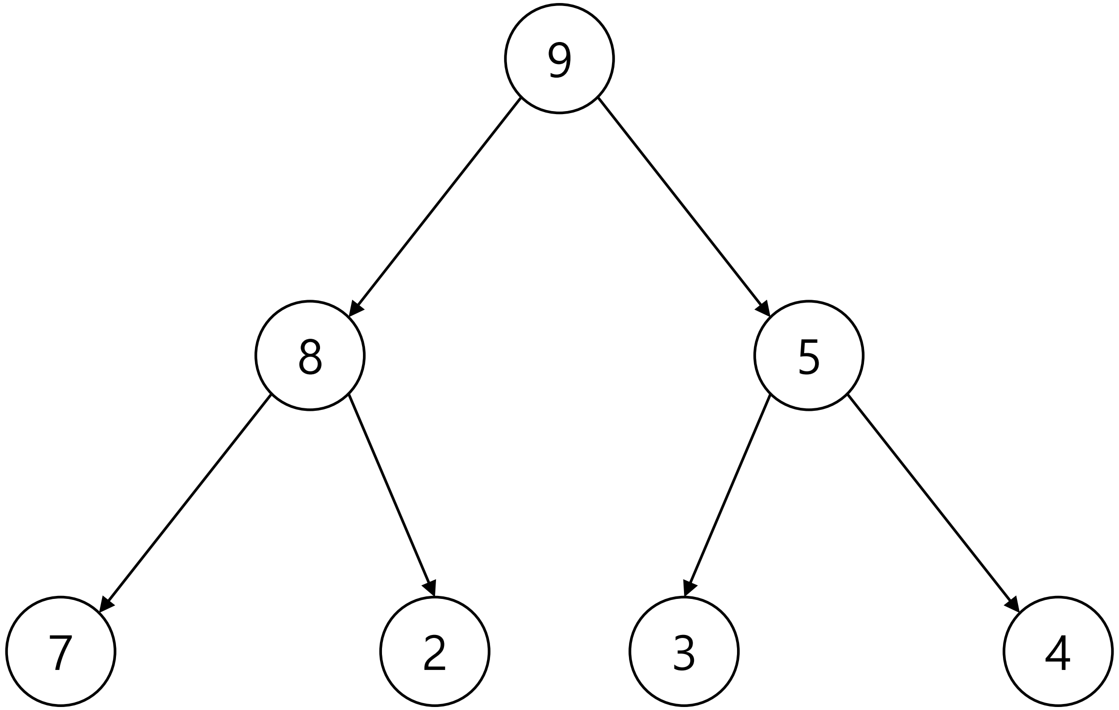
- Priority Queue
  - Queue: 먼저 들어온 데이터가 먼저 나가는 자료구조
  - Priority Queue: 우선순위가 높은 데이터가 먼저 나가는 자료구조
    - 원소 삽입
    - 가장 큰 원소 검색
    - 가장 큰 원소 제거

  - 우선순위로 BST를 만들고 가장 오른쪽 자손을 반환?
    - BBST는 O(log N)이지만 생각보다 느림
    - log N vs 10 log N 느낌
  - 더 좋은 방법은 없을까?

Heap

- Heap
  - 각 정점의 값이 자식 정점보다 큰 트리 구조
    - BST보다 제약 조건이 적음
    - 보통 제약 조건이 약하면 더 효율적으로 구현 가능
  - 주로 사용하는 건 이진 힙 (Binary Heap)
    - 완전 이진 트리(Complete Binary Tree) 형태
    - 루트 : 1번 인덱스
    - x의 부모 : x / 2
    - x의 왼쪽/오른쪽 자식 : 2x, 2x+1

Heap



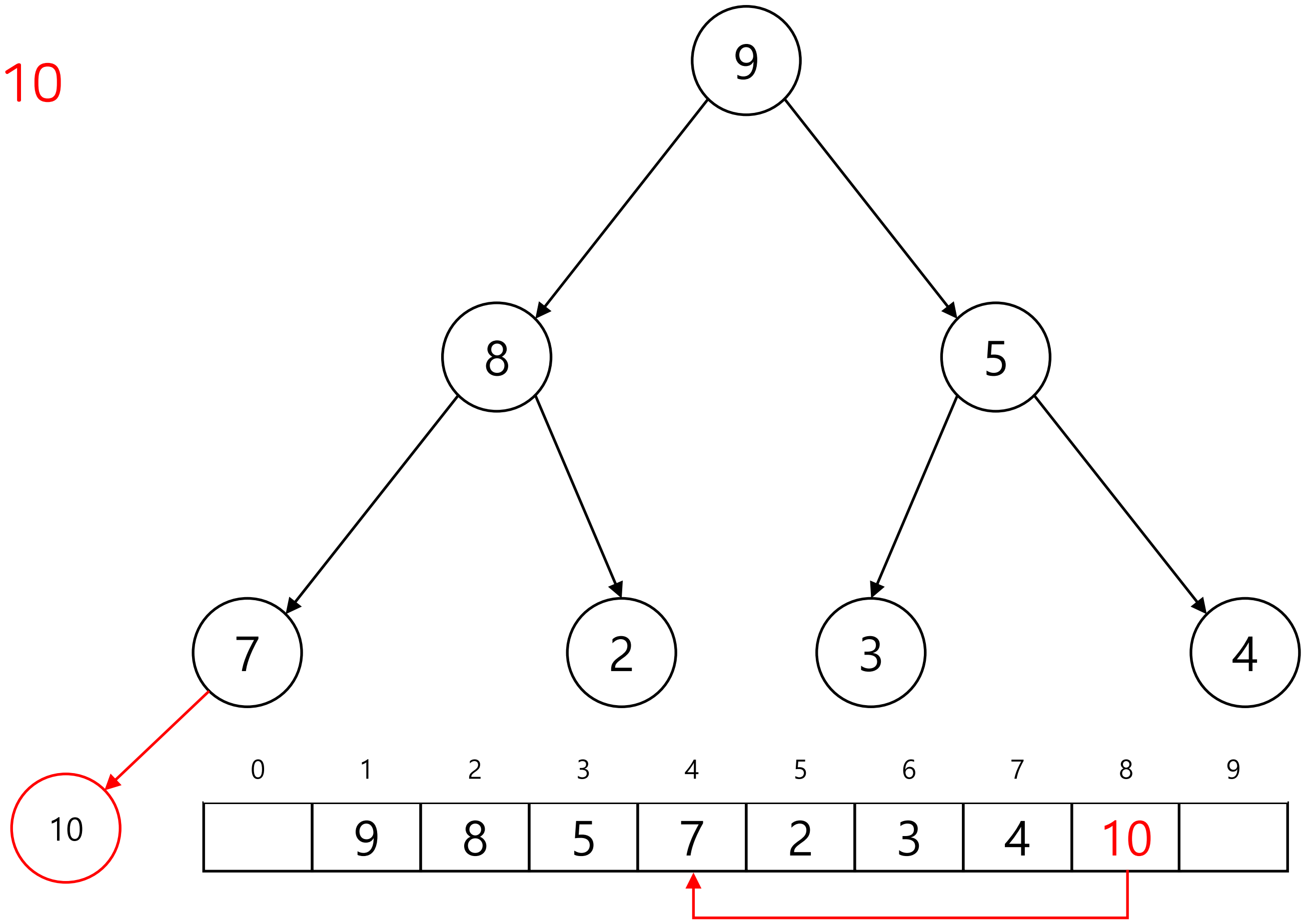
0	1	2	3	4	5	6	7	8	9
	9	8	5	7	2	3	4		

Heap

- 원소 삽입
  - 배열의 맨 뒤에 원소 추가
  - 만약 부모가 더 작으면 부모와 교환

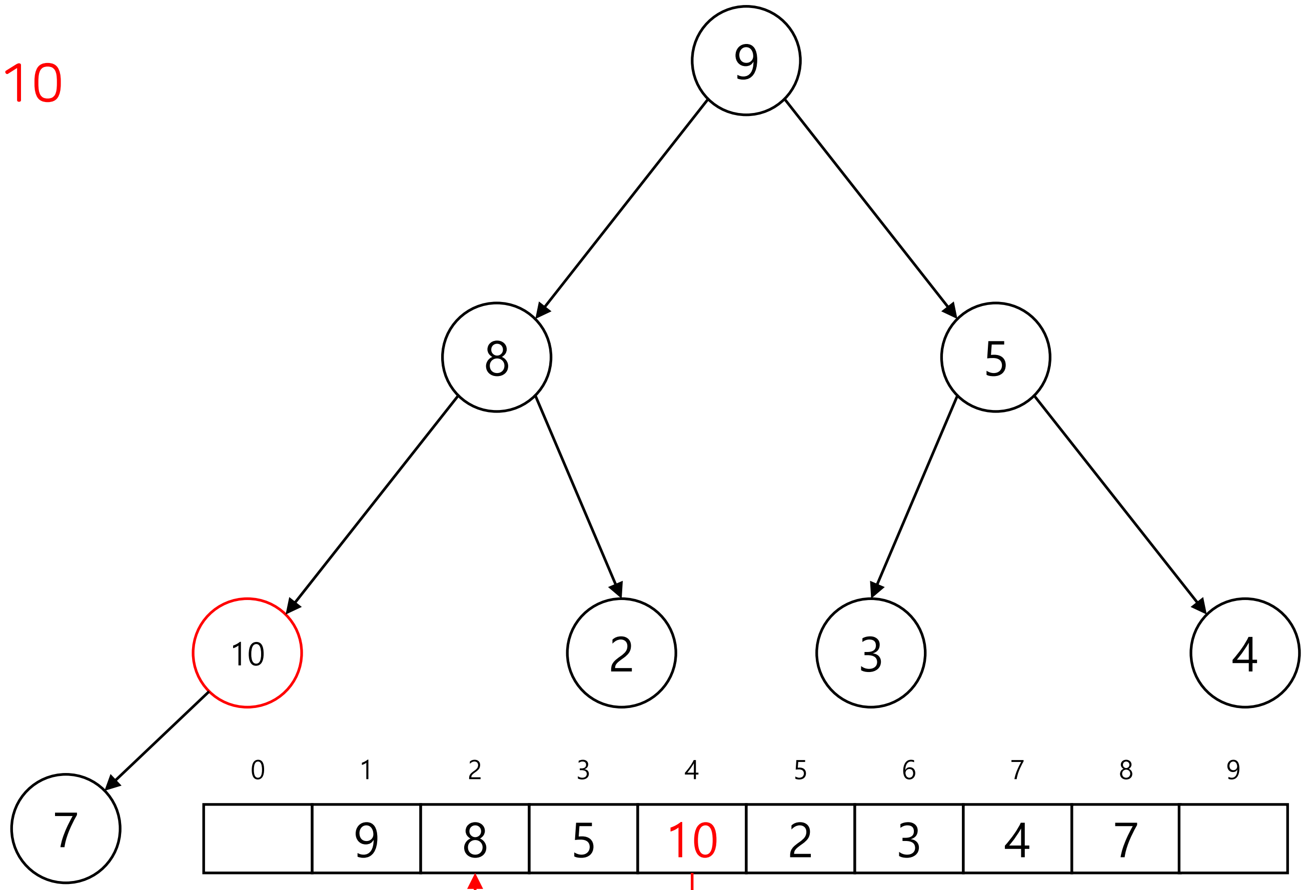
Heap

- 원소 삽입
  - INSERT 10



Heap

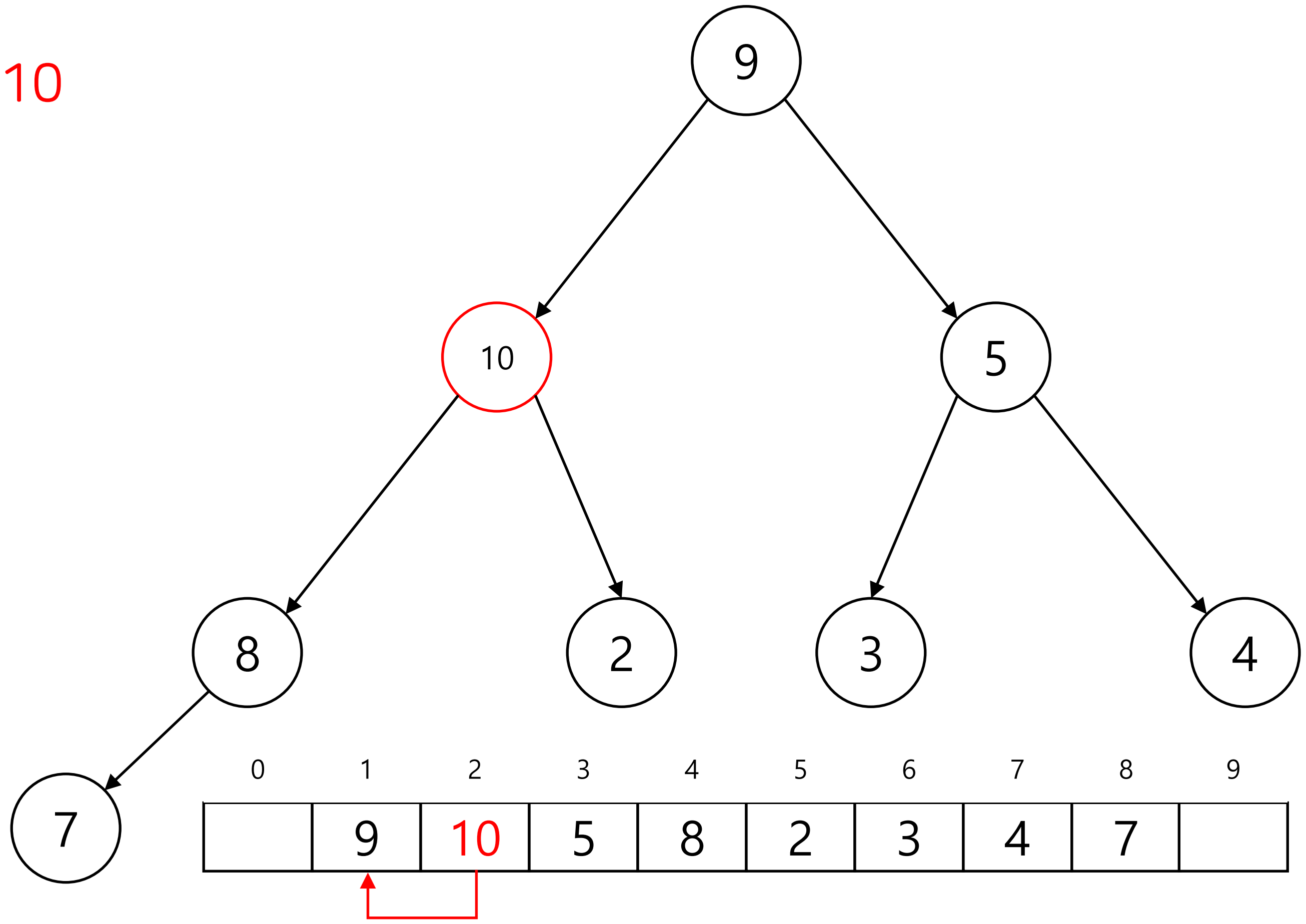
- 원소 삽입
  - INSERT 10



0	1	2	3	4	5	6	7	8	9
	9	8	5	10	2	3	4	7	

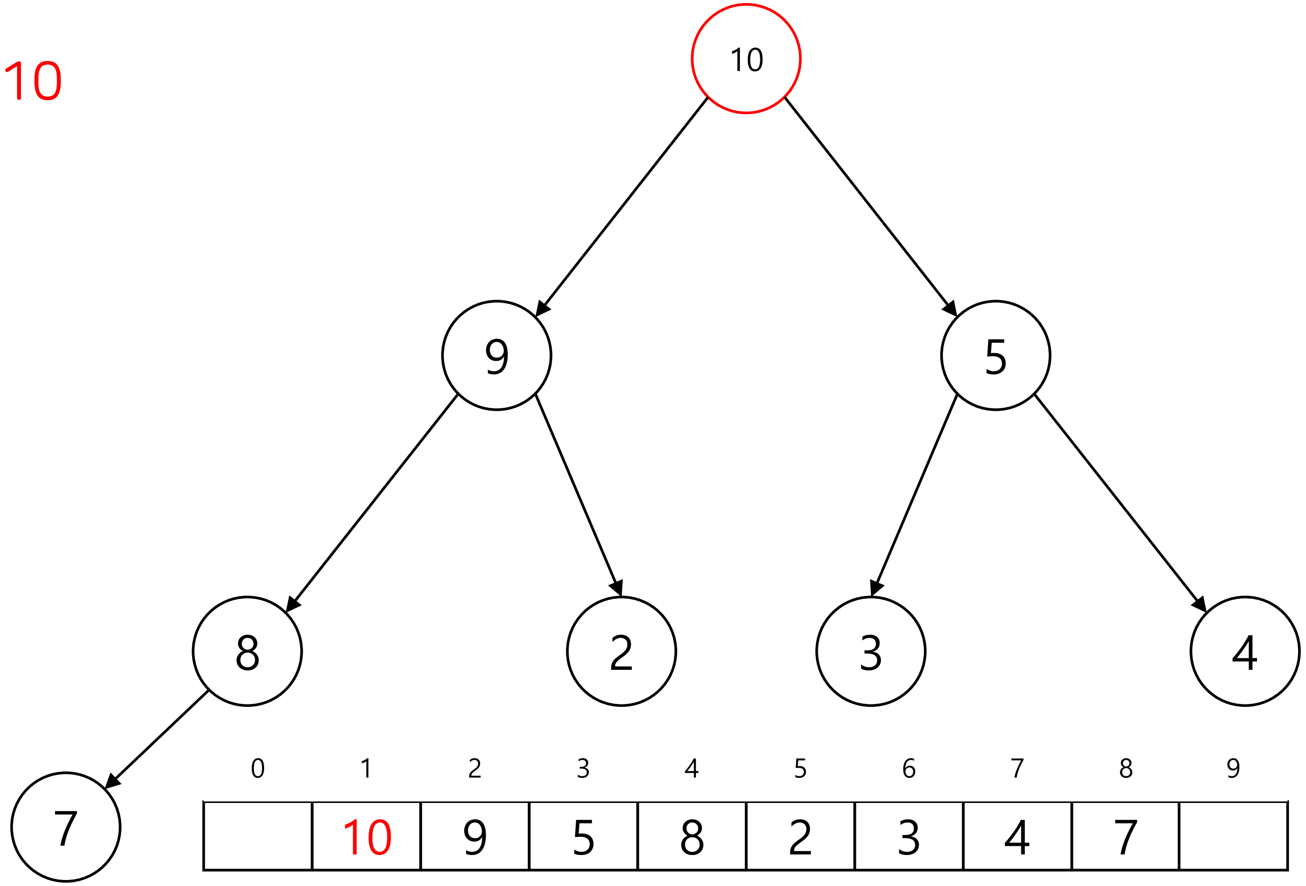
Heap

- 원소 삽입
  - INSERT 10



Heap

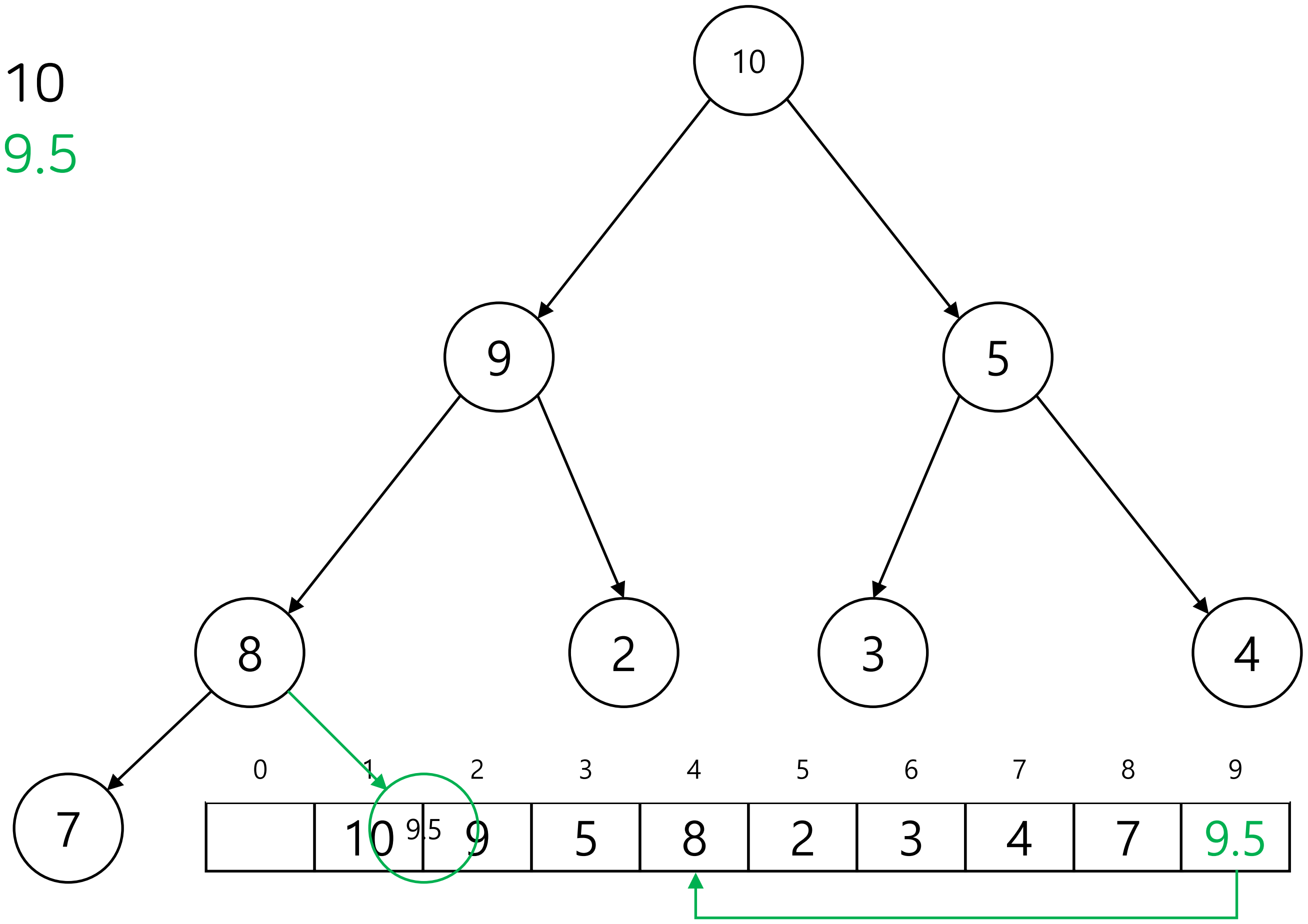
- 원소 삽입
  - INSERT 10



0	1	2	3	4	5	6	7	8	9
	10	9	5	8	2	3	4	7	

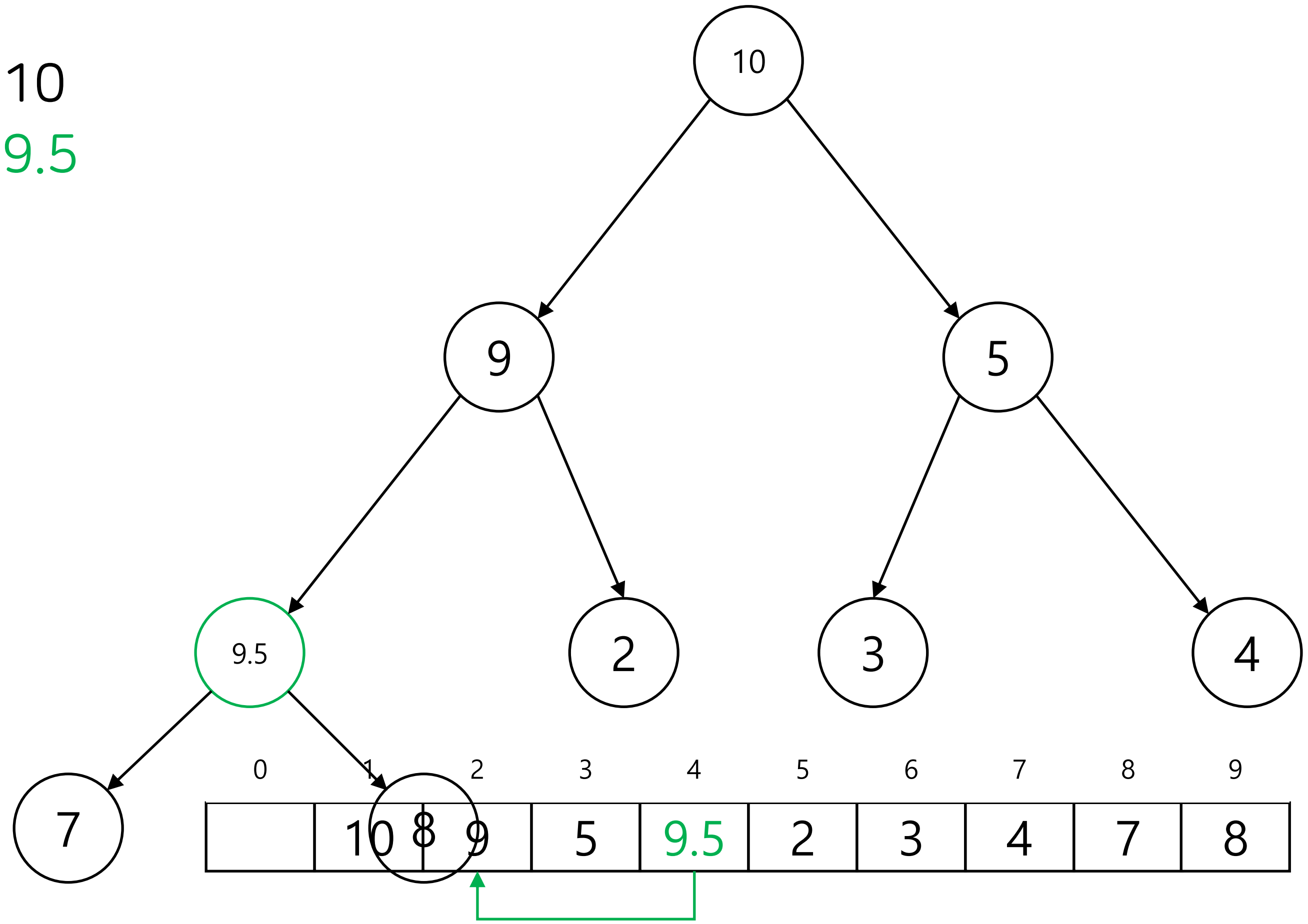
Heap

- 원소 삽입
  - INSERT 10
  - INSERT 9.5



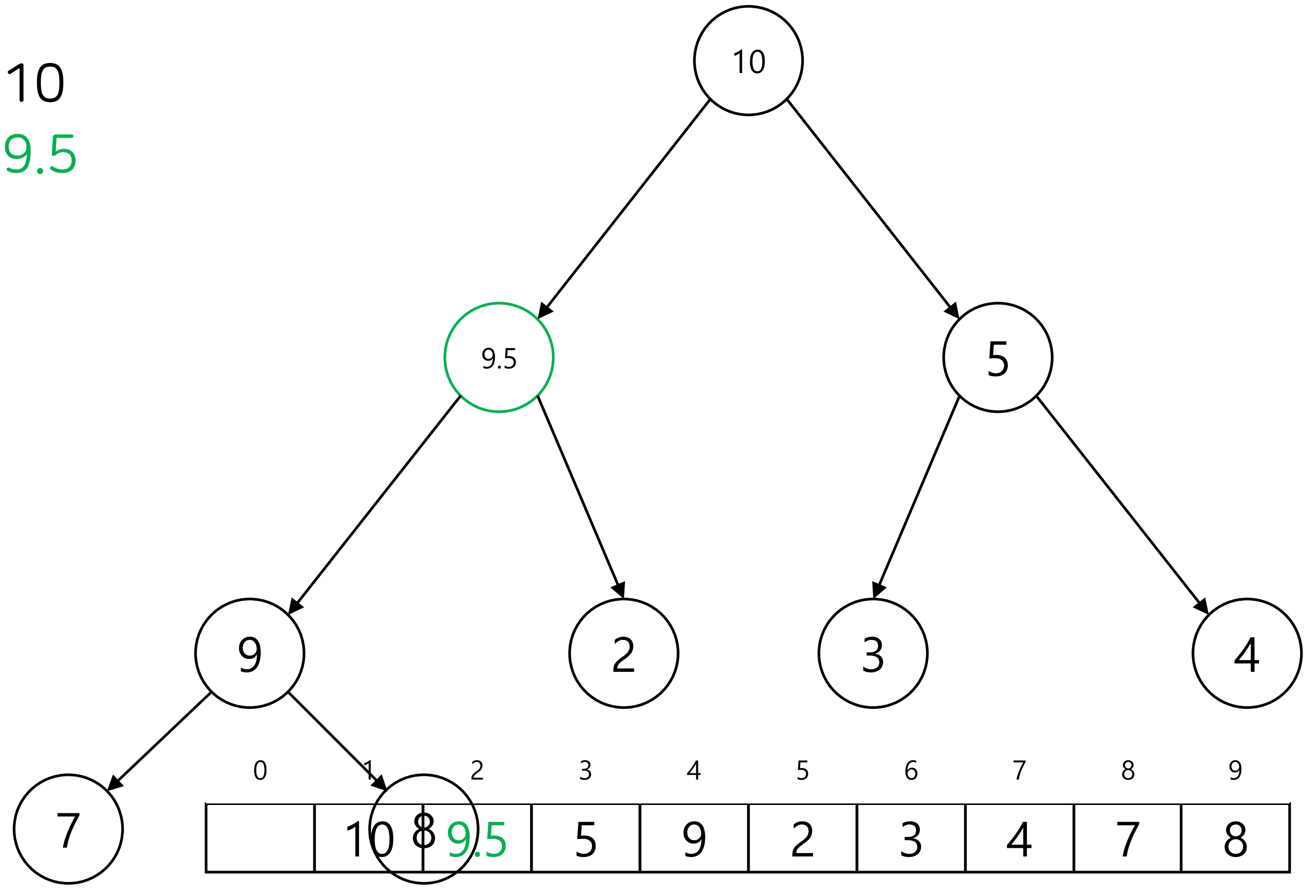
Heap

- 원소 삽입
  - INSERT 10
  - INSERT 9.5



Heap

- 원소 삽입
  - INSERT 10
  - INSERT 9.5



Heap

- 원소 삽입
  - 시간 복잡도 : O(h) = O(log N)

Heap

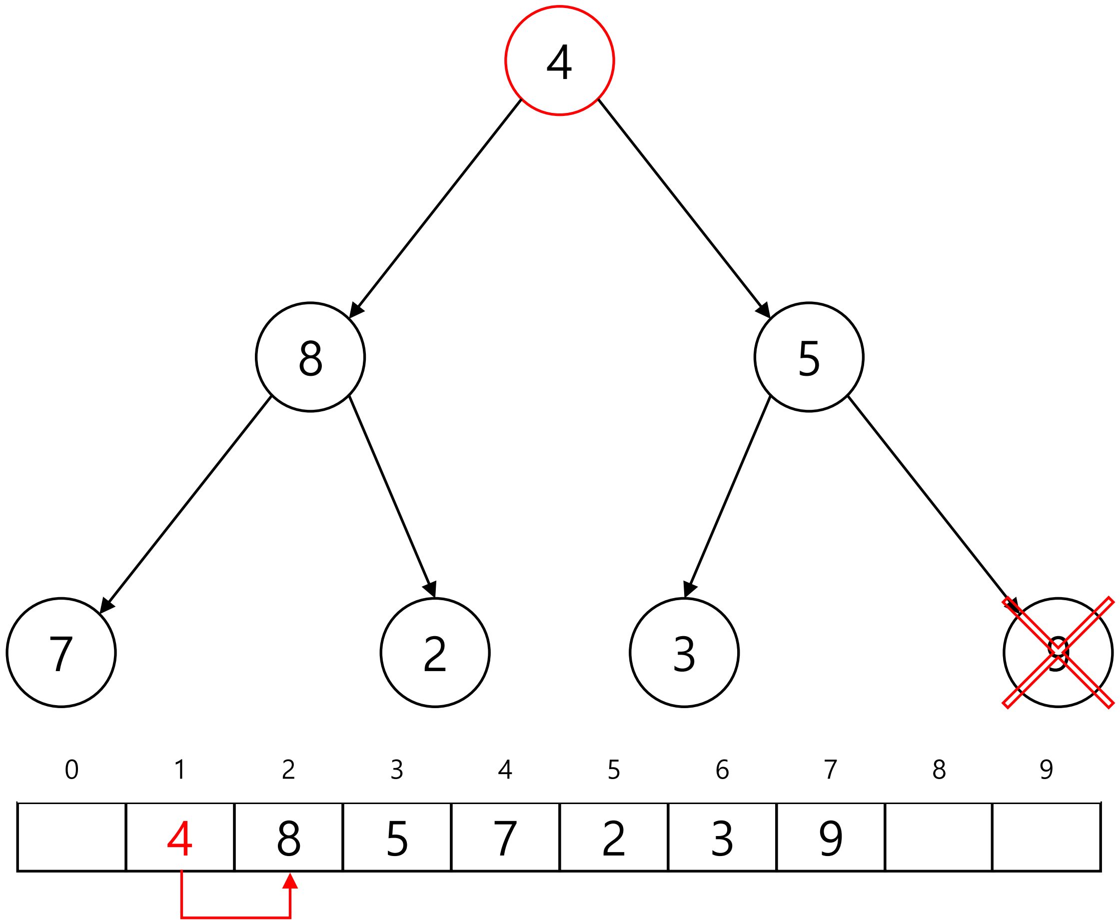
- 가장 큰 원소 탐색
  - return heap[1];
  - 시간 복잡도 : O(1)

Heap

- 가장 큰 원소 제거
  - 루트(가장 큰 원소)와 마지막 정점의 값을 바꿈
  - 마지막 정점 제거
  - 현재 루트에 있는 값이 자식보다 작으면 밑으로 내림
    - 두 자식 모두 현재 정점보다 크면 더 큰 방향으로 이동

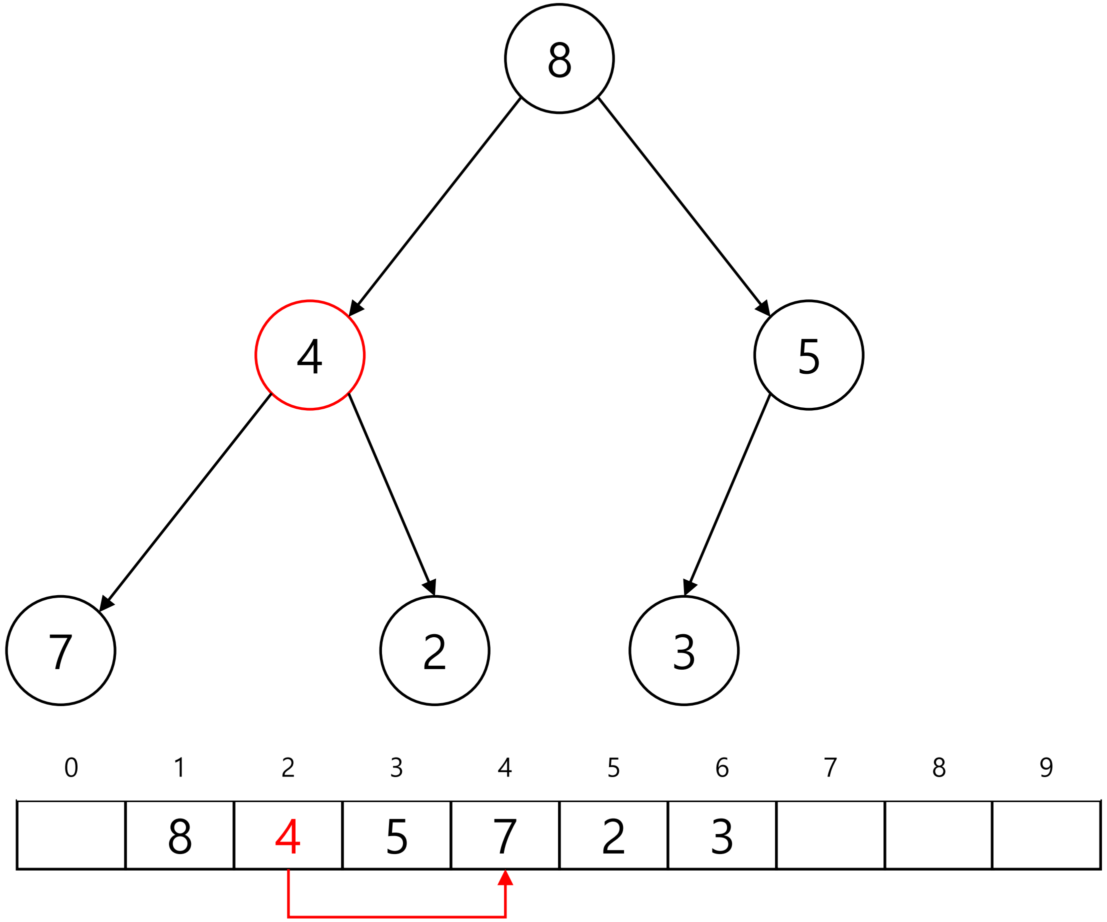
Heap

- 가장 큰 원소 제거
  - POP 9



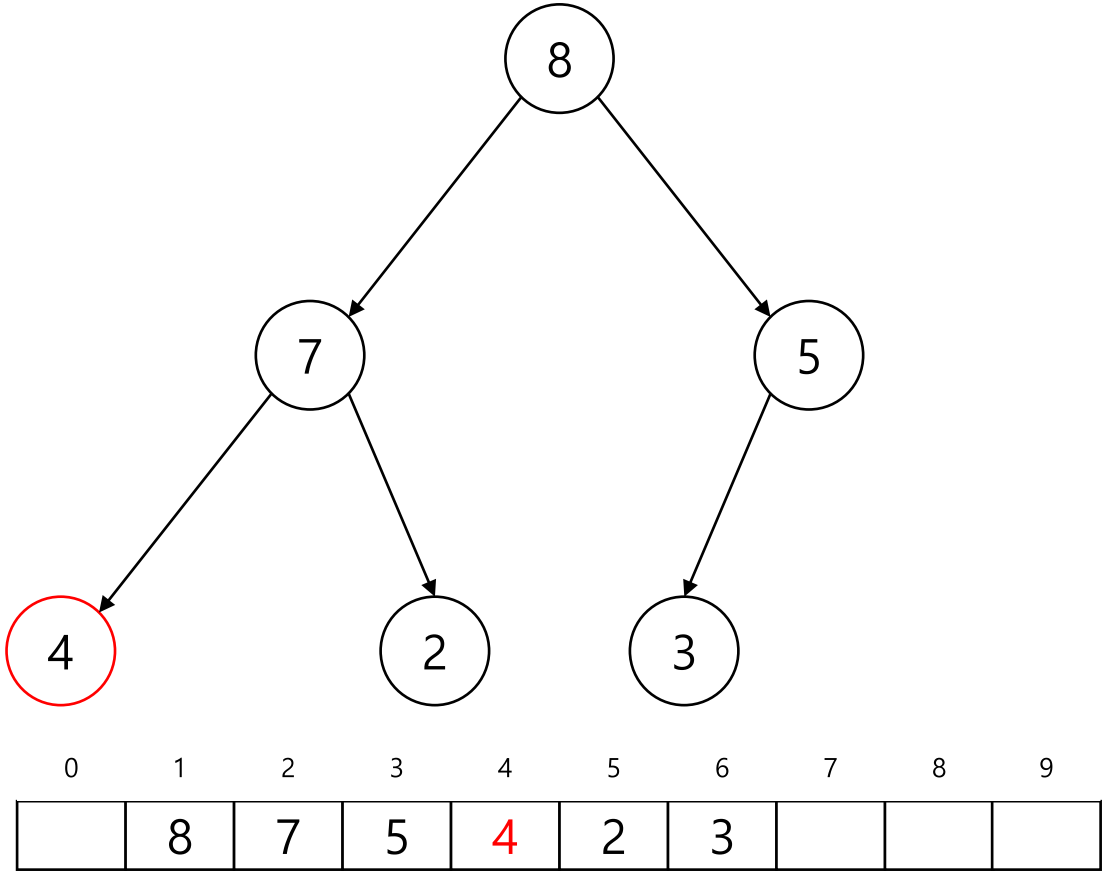
Heap

- 가장 큰 원소 제거
  - POP 9



Heap

- 가장 큰 원소 제거
  - POP 9



Heap

- 가장 큰 원소 제거
  - 시간 복잡도 : O(h) = O(log N)

Heap

- 구현

```
int Heap[101010], sz = 0;

void push(int v){
    Heap[++sz] = v;
    for(int i=sz; i>1; i/=2){
        if(Heap[i] > Heap[i/2]) swap(Heap[i/2], Heap[i]);
        else break;
    }
}

void pop(){
    Heap[1] = Heap[sz--];
    for(int i=1; i*2<=sz; ){
        int ch = i*2;
        if(ch+1 <= sz && Heap[ch+1] > Heap[ch]) ch += 1;
        if(Heap[ch] > Heap[i]) swap(Heap[ch], Heap[i]), i = ch;
        else break;
    }
}
```

Heap

- std::priority_queue
  - #include <queue>
  - 이진 힙으로 구현되어 있음

질문?

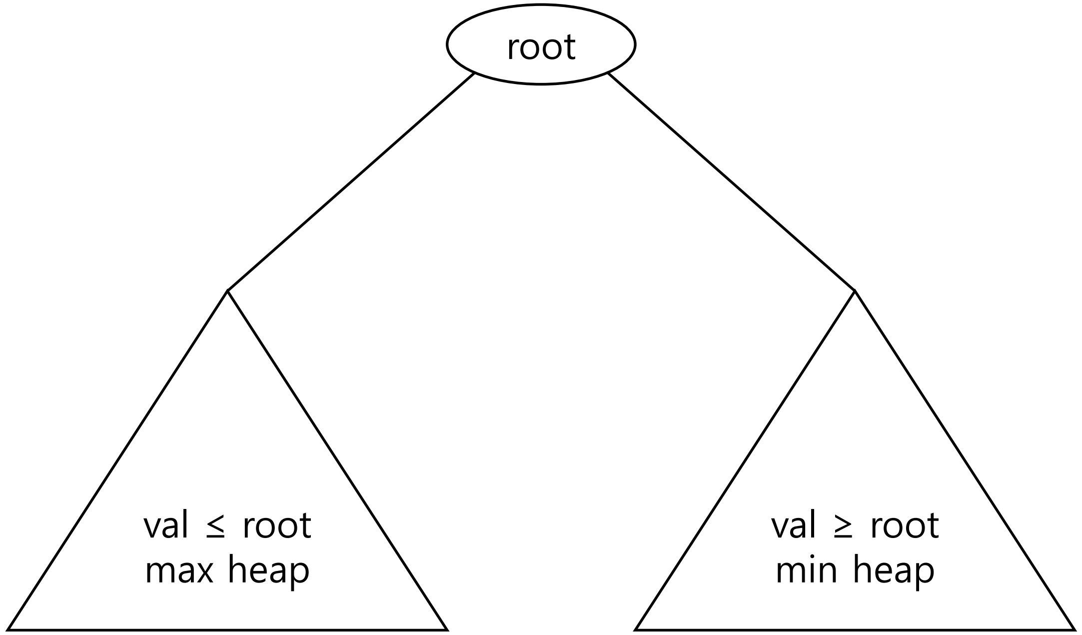
Heap – 예시 1

- BOJ 2696 중앙값 구하기
  - 모든 홀수 i에 대해 A[1..i]의 중앙값을 구하는 문제

  - 원소가 2개 추가될 때마다 중앙값을 구하는 문제
    - 현재의 중앙값을 x라고 하자.
      - 새로 추가된 값이 모두 x 이하 : x 이하인 가장 큰 값이 새로운 중앙값
      - 새로 추가된 값이 모두 x 이상 : x 이상인 가장 작은 값이 새로운 중앙값
      - 하나는 x 이하, 하나는 x 이상 : x가 중앙값
    - x 이하인 원소를 관리하는 max heap과 x 이상인 원소를 관리하는 min heap

Heap – 예시 1

- BOJ 2696 중앙값 구하기
  - x 이하인 원소를 관리하는 max heap과 x 이상인 원소를 관리하는 min heap




```cpp
void Solve(){
    int N; cin >> N;
    priority_queue<int, vector<int>, less<>> max_heap;
    priority_queue<int, vector<int>, greater<>> min_heap;
    int root; cin >> root;

    cout << (N + 1) / 2 << "\n";
    cout << root << " ";
    for(int i=0; i<N/2; i++){
        int a, b; cin >> a >> b;
        if(a > b) swap(a, b);
        if(a <= root && root <= b){
            max_heap.push(a); min_heap.push(b);
        }
        else if(b <= root){
            max_heap.push(a); max_heap.push(b);
            min_heap.push(root);
            root = max_heap.top(); max_heap.pop();
        }
        else{
            min_heap.push(a); min_heap.push(b);
            max_heap.push(root);
            root = min_heap.top(); min_heap.pop();
        }
        cout << root << " ";
    }
    cout << "\n";
}
```

질문?

Union Find

Union Find

- 서로소 집합족
  - 교집합이 공집합인 집합으로 구성된 집합족
    - 두 집합 A, B가 A = B이거나 $A \cap B = \emptyset$
    - 서로소인 두 정수 A, B의 소인수분해를 생각해 보자.
  - 각 원소가 속한 집합을 유일하게 특정할 수 있음

Union Find

- Union Find (또는 Disjoint Set)
  - 서로소 집합을 관리하는 자료구조
    - Init : 모든 원소가 자기 자신만을 원소로 하는 집합에 속하도록 초기화
    - Union(u, v) : u가 속한 집합과 v가 속한 집합을 병합
    - Find(v) : v가 속한 집합을 반환

  - 위 3가지 연산을 빠르게 구현하는 것이 목표

Union Find

- Union Find
  - 각 집합을 Rooted Tree로 표현한다고 생각해 보자.
    - Init은 포레스트에서 모든 간선을 제거하는 것과 동일
    - Find(v)는 v가 속한 트리의 루트 정점을 반환하면 됨
    - Union(u, v)는 u가 속한 트리의 루트를 v가 속한 트리의 루트의 자식으로 넣어주면 됨

    - 시간 복잡도는?

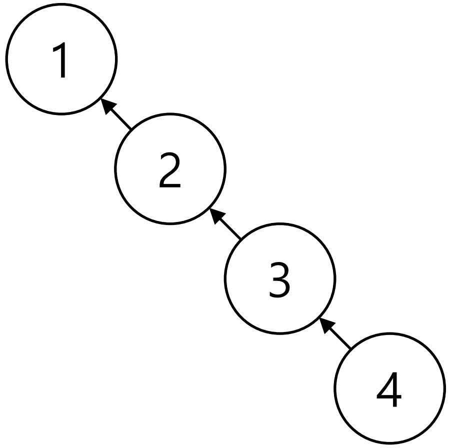
Union Find

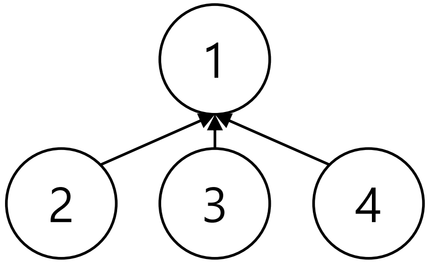
```
int Parent[101010];

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

int Find(int v){
    if(v == Parent[v]) return v;
    return Find(Parent[v]);
}

void Union(int u, int v){
    int u_root = Find(u), v_root = Find(v);
    if(u_root != v_root) Parent[u_root] = v_root;
}
```

Union Find

- Union Find
  - Init : O(N)
  - Find : 트리의 높이를 h라고 하면 O(h)
  - Union : Find와 동일

  - 최악의 경우 O(N)
  - 최선의 경우 O(1)





질문?

Union Find

- Union Find 최적화
  - 트리의 높이를 줄여야 함
    - Union by Rank
    - Union by Size
    - Path Compression

  - 3개 중 하나만 사용해도 M번 연산했을 때 O(M log N)이 보장됨
    - Union by Rank, 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 : N을 1 이하로 만들기 위해서 필요한 log의 횟수
    - log* N = 1 + log* (log N)

Union Find

- Union by Rank
  - 높이가 낮은 트리를 높은 트리 아래로 넣는 방법
    - Rank[i] : i를 루트로 하는 트리 높이의 상한
    - 만약 두 트리의 높이가 동일하면 Union 후 Rank 1 증가

```
int Parent[101010], Rank[101010];

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;
    return Find(Parent[v]);
}

void Union(int u, int v){
    int u_root = Find(u), v_root = Find(v);
    if(u_root == v_root) return;
    if(Rank[u_root] > Rank[v_root]) swap(u_root, v_root);
    Parent[u_root] = v_root;
    if(Rank[u_root] == Rank[v_root]) Rank[v_root] += 1;
}
```

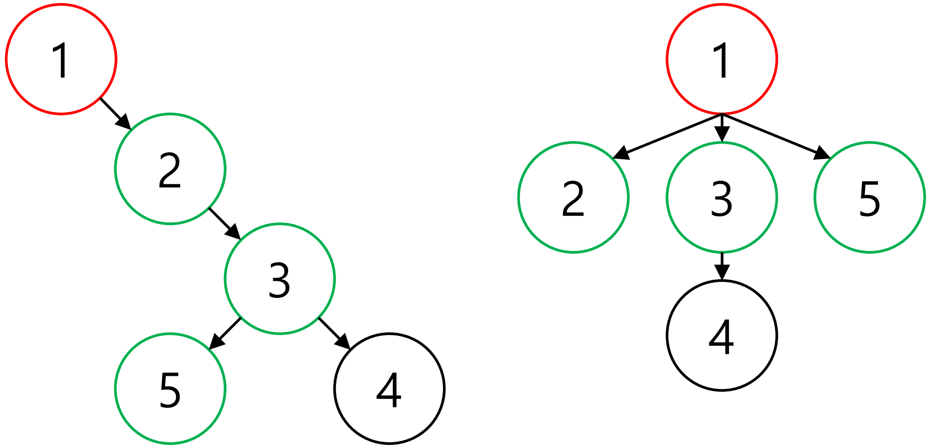
Union Find

- Union by Size
  - 정점이 적은 트리를 많은 트리 아래로 넣는 방법
    - Size[i] = i를 루트로 하는 트리의 정점 개수
    - u를 v 밑으로 넣으면 Size[v]는 Size[u] 만큼 증가

```
int Parent[101010], Size[101010];

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;
    return Find(Parent[v]);
}

void Union(int u, int v){
    int u_root = Find(u), v_root = Find(v);
    if(u_root == v_root) return;
    if(Size[u_root] > Size[v_root]) swap(u_root, v_root);
    Parent[u_root] = v_root;
    Size[v_root] += Size[u_root];
}
```

Union Find

- Path Compression
  - 경로 압축
    - Find를 통해 루트를 찾았다면
    - 루트까지의 경로 상에 있는 정점을 모두 루트의 바로 밑(자식)으로 붙임




```
int Parent[101010];

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

int Find(int v){
    if(v == Parent[v]) return v;
    return Parent[v] = Find(Parent[v]);
}

void Union(int u, int v){
    int u_root = Find(u), v_root = Find(v);
    if(u_root != v_root) Parent[u_root] = v_root;
}
```

질문?

Union Find – 예시 1

- BOJ 1976 여행 가자
  - 그래프 G와 수열 A가 주어짐
  - 수열 A에서 인접한 정점 u, v가 G에서 연결되어 있는지 확인하는 문제

  - 그래프의 연결 요소를 Union Find로 관리하면 됨

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M, A[1010], P[222];
int Find(int v){ return v == P[v] ? v : P[v] = Find(P[v]); }
void Merge(int u, int v){ if(Find(u) != Find(v)) P[Find(u)] = Find(v); }

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    iota(P+1, P+N+1, 1);
    for(int i=1; i<=N; i++){
        for(int j=1; j<=N; j++){
            int t; cin >> t;
            if(t) Merge(i, j);
        }
    }
    for(int i=1; i<=M; i++) cin >> A[i];

    bool res = true;
    for(int i=2; i<=M; i++) res &= Find(A[i-1]) == Find(A[i]);
    cout << (res ? "YES" : "NO");
}
```

Union Find – 예시 2

- BOJ 13306 트리
  1. 트리의 간선을 제거하는 쿼리
  2. 두 정점 u, v가 같은 컴포넌트에 속하는지 확인하는 쿼리

  - Union Find는 간선을 추가하는 쿼리만 처리할 수 있음
    - 일단 쿼리를 모두 입력받고 뒤에서부터 처리하면
    - 1번 쿼리를 간선 추가 쿼리로 생각할 수 있음
    - 2번 쿼리는 u, v가 같은 집합에 속하는지 확인하면 됨

```cpp
#include <bits/stdc++.h>
using namespace std;

int P[202020];
int Find(int v){ return v == P[v] ? v : P[v] = Find(P[v]); }
void Merge(int u, int v){ if(Find(u) != Find(v)) P[Find(u)] = Find(v); }

int N, Q, G[202020];
int A[404040], B[404040], C[404040];
vector<int> R;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> Q; Q += N-1;
    for(int i=2; i<=N; i++) cin >> G[i];
    for(int i=1; i<=Q; i++){
        cin >> A[i];
        if(A[i] == 0) cin >> B[i];
        else cin >> B[i] >> C[i];
    }

    iota(P+1, P+N+1, 1);
    for(int i=Q; i>=1; i--){
        if(A[i] == 0) Merge(B[i], G[B[i]]);
        else R.push_back(Find(B[i]) == Find(C[i]));
    }

    reverse(R.begin(), R.end());
    for(auto i : R) cout << (i ? "YES" : "NO") << "\n";
}
```

질문?

과제

- 필수
  - 1822 차집합
  - 20920 영단어 암기는 괴로워
  - 2696 중앙값 구하기
  - 1976 여행 가자
  - 13306 트리
- 심화
  - 1351 무한 수열
  - 2957 이진 탐색 트리
  - 15942 Thinking Heap
  - 17469 트리의 색깔과 쿼리