# Segment Tree

나정휘
https://justiceHui.github.io/

# 목차

- AM-GM Inequality
- Sqrt Decomposition
- Segment Tree
- Segment Tree + Binary Search
- Applications

# AM-GM Inequality

# AM-GM Inequality

- AM-GM Inequality
  - Arithmetic Mean 산술 평균
  - Geometric Mean 기하 평균
  - Inequality 부등식

# AM-GM Inequality

- 음이 아닌 실수 $x_1, x_2, \cdots, x_n$ 에 대해 아래 식이 성립함
  - $\frac{x_1+x_2\cdots+x_n}{n} \geq \sqrt[n]{x_1 x_2 \cdots x_n}$
  - 등호가 성립할 조건은 $x_1 = x_2 = \cdots = x_n$

- $n = 2$ 인 경우
  - $a + b \geq 2\sqrt{ab}$
  - $a = b$ 일 때 등호 성립
  - 두 식의 곱이 일정할 때 합의 최솟값을 찾을 수 있음

# AM-GM Inequality

- 간단한 정렬 알고리즘
  - 길이가 N인 배열을 정렬하는 알고리즘
    - 배열을 크기가 B인 N/B개의 덩어리로 분할
    - 각 덩어리를 $O(N^2)$ 알고리즘으로 정렬
    - N/B개의 덩어리는 각각 정렬되어 있는 상태

    - 각 덩어리의 맨 앞에 있는 원소들의 최솟값을 정답에 추가
    - N번 반복하면 전체 배열이 정렬됨

  - 시간 복잡도는?

# AM-GM Inequality

- 간단한 정렬 알고리즘 – 시간 복잡도
  - 길이가 N인 배열을 정렬하는 알고리즘
    - 배열을 크기가 B인 N/B개의 덩어리로 분할
    - 각 덩어리를 $O(N^2)$ 알고리즘으로 정렬 $B^2 * N/B = NB$
    - N/B개의 덩어리는 각각 정렬되어 있는 상태

    - 각 덩어리의 맨 앞에 있는 원소들의 최솟값을 정답에 추가 $N/B * N = N^2/B$
    - N번 반복하면 전체 배열이 정렬됨

  - 시간 복잡도는? $NB + N^2/B$

# AM-GM Inequality

- 간단한 정렬 알고리즘 – 시간 복잡도
  - 길이가 N인 배열을 정렬하는 알고리즘
    - 배열을 크기가 B인 N/B개의 덩어리로 분할
    - 각 덩어리를 $O(N^2)$ 알고리즘으로 정렬 — $B^2 * N/B = NB$
    - N/B개의 덩어리는 각각 정렬되어 있는 상태

    - 각 덩어리의 맨 앞에 있는 원소들의 최솟값을 정답에 추가 — $N/B * N = N^2/B$
    - N번 반복하면 전체 배열이 정렬됨

  - 시간 복잡도는? — $NB + N^2/B$
    - 곱이 일정하므로 $NB = N^2/B$일 때 최소가 됨
    - $B = \sqrt{N}$ 이면 전체 시간 복잡도는 $O(N\sqrt{N})$

질문?

# Sqrt Decomposition

# Sqrt Decomposition

- BOJ 14438 수열과 쿼리 17
  - 1: $A_i$를 v로 바꾼다.
  - 2: $A_i$, $A_{i+1}$, … , $A_j$에서 크기가 가장 작은 값을 출력한다.

  - 단순하게 구현하면 1번 쿼리 O(1), 2번 쿼리 O(N)

# Sqrt Decomposition

- BOJ 14438 수열과 쿼리 17
  - 정렬 알고리즘에서 본 방식을 사용해 보자.
    - 배열을 크기가 B인 N/B개의 버킷으로 분할
    - 각 버킷의 최솟값을 전처리
      - O(N)에 가능
    - 1: 원소의 값을 바꾼 뒤, 그 원소가 속한 버킷의 최솟값을 갱신
      - 버킷의 크기는 B 이므로 O(B)에 가능
    - 2: 구간에 완전히 포함된 버킷은 버킷의 최솟값을 취하고, 일부만 겹친 버킷은 모든 원소를 확인
      - 구간에 완전히 포함된 버킷은 최대 N/B개
      - 일부만 겹친 버킷은 최대 2개이므로 최대 2B개의 버킷의 원소를 확인
      - O(N/B + 2B)
  - B = √N 이면 전체 시간 복잡도는 O(N + Q√N)

# Sqrt Decomposition

- A = {1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 2, 6}, B = 3

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

- Update(5, 3)

| 1 | | | 3 | | | 2 | | | 2 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 7 | 9 | 3 | 2 | 4 | 6 | 8 | 2 | 6 |

- Update(6, 5)

| 1 | | | 3 | | | 4 | | | 2 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 7 | 9 | 3 | 5 | 4 | 6 | 8 | 2 | 6 |

- Query(1, 9)

| 1 | | | 3 | | | 4 | | | 2 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 7 | 9 | 3 | 5 | 4 | 6 | 8 | 2 | 6 |

# Sqrt Decomposition

```cpp
constexpr int B = 400;
int N, Q, A[101010], M[101010/B];

void Calc(int id){
    int s = id * B, e = min(N, (id + 1) * B);
    M[id] = 1e9;
    for(int i=s; i<e; i++) M[id] = min(M[id], A[i]);
}
void Init(){
    for(int i=0; i*B<N; i++) Calc(i);
}
void Update(int x, int v){
    A[x] = v; Calc(x / B);
}
int Query(int l, int r){
    int res = 1e9;
    while(l <= r && l % B != 0) res = min(res, A[l++]);
    while(l <= r && r % B + 1 != B) res = min(res, A[r--]);
    if(l <= r) for(int i=l/B; i<=r/B; i++) res = min(res, M[i]);
    return res;
}
```

질문?

# Segment Tree

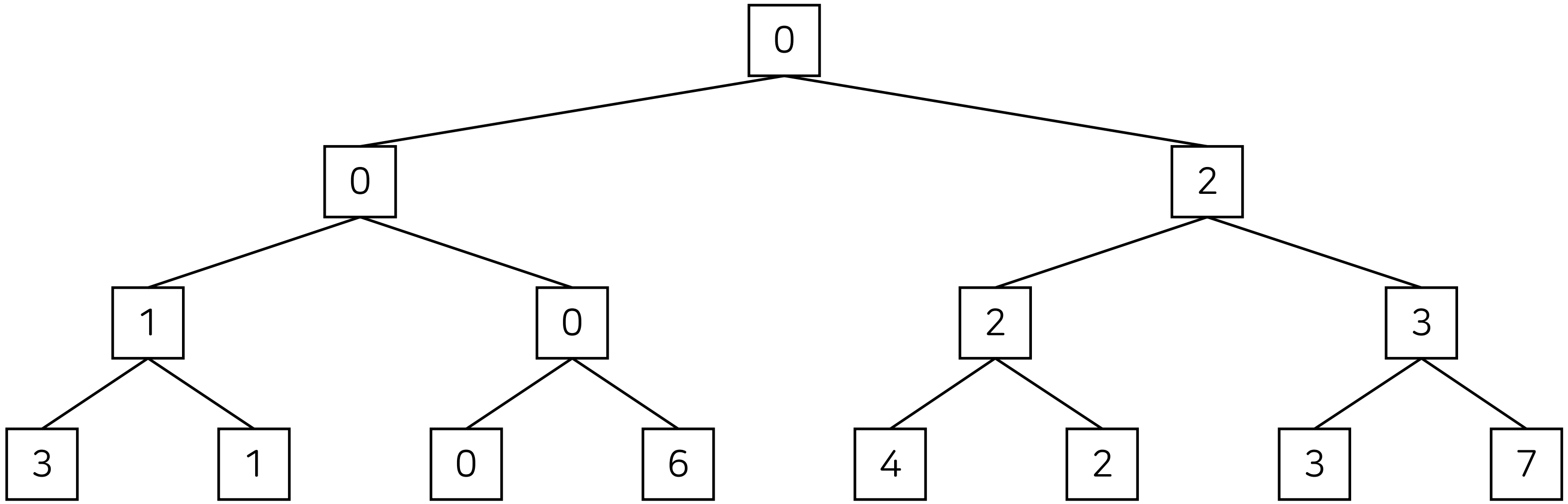
# Segment Tree

- 굳이 √N 크기로 한 번만 분할할 이유가 있을까?
  - 크기가 B=√N인 버킷을 다시 √B인 버킷으로 분할한다면?
  - 크기가 제곱근인 버킷으로 총 O(log log N)번 분할할 수 있고, 딱히 분할하지 않을 이유가 없음

<table>
<tr><td colspan="6">1</td><td colspan="6">0</td><td colspan="6">2</td><td colspan="6">1</td></tr>
<tr><td colspan="2">3</td><td colspan="2">5</td><td colspan="2">1</td><td colspan="2">0</td><td colspan="2">1</td><td colspan="2">3</td><td colspan="2">2</td><td colspan="2">5</td><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">1</td><td colspan="2">6</td></tr>
<tr><td>8</td><td>3</td><td>5</td><td>8</td><td>1</td><td>7</td><td>4</td><td>0</td><td>9</td><td>1</td><td>8</td><td>3</td><td>2</td><td>4</td><td>5</td><td>6</td><td>2</td><td>7</td><td>9</td><td>3</td><td>1</td><td>3</td><td>7</td><td>6</td></tr>
</table>

# Segment Tree

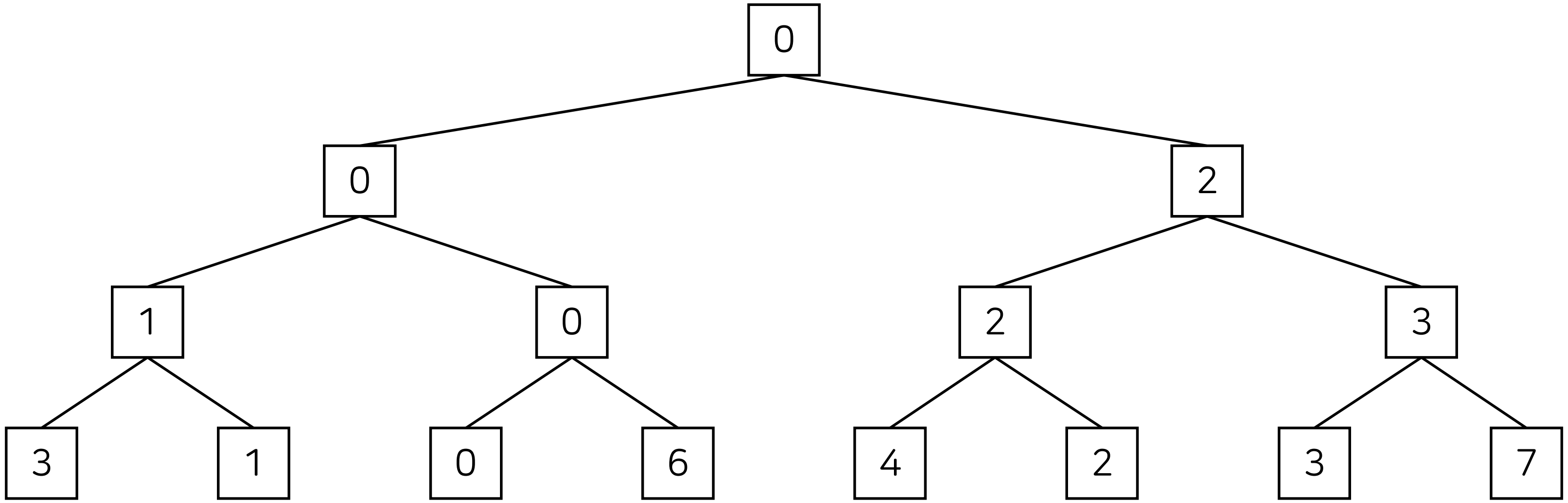
- Segment Tree
  - 분할하는 김에 계산하기 깔끔하게 절반씩 분할하자!
    - 매번 크기가 절반이 되므로 최대 O(log N)번 분할됨
  - 포화 이진 트리
    - 배열로 저장 가능
    - 왼쪽 자식 2x, 오른쪽 자식 2x+1, 부모 정점 x/2
    - N = $2^k$꼴이면 A[i]를 담당하는 리프 정점은 $2^k$+i번 정점



# Segment Tree

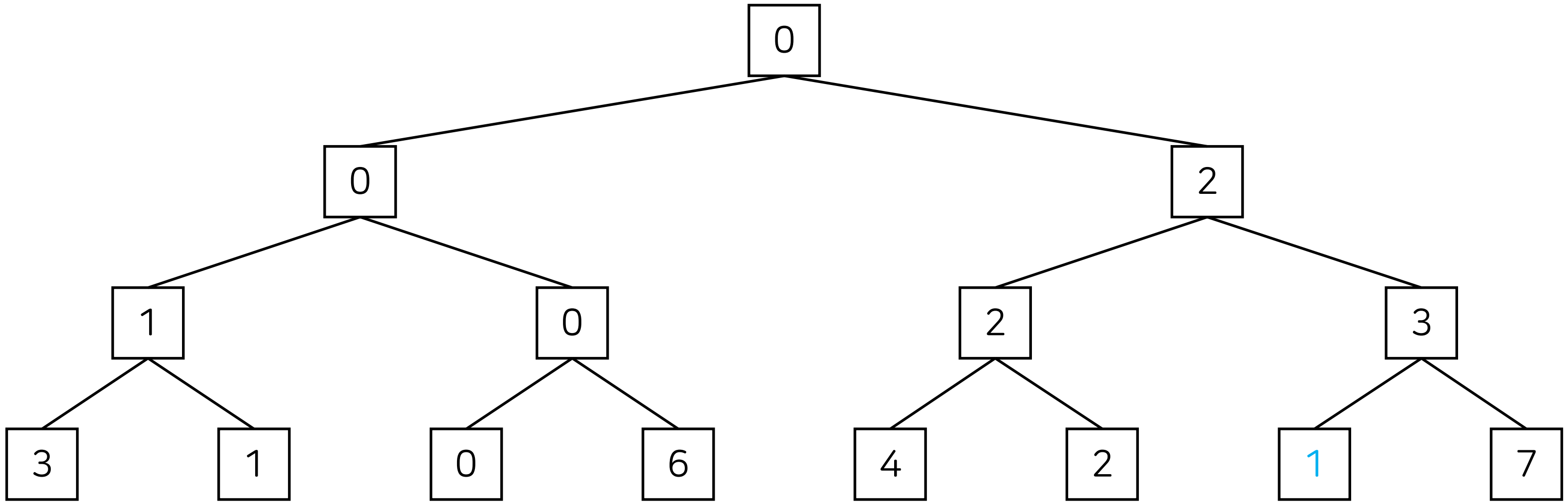
- Segment Tree
  - 원소 수정: 리프 정점 수정하고, 부모 정점 타고 가면서 갱신
    - O(log N)개의 정점만 갱신하면 됨

  - Update(6, 1)



# Segment Tree

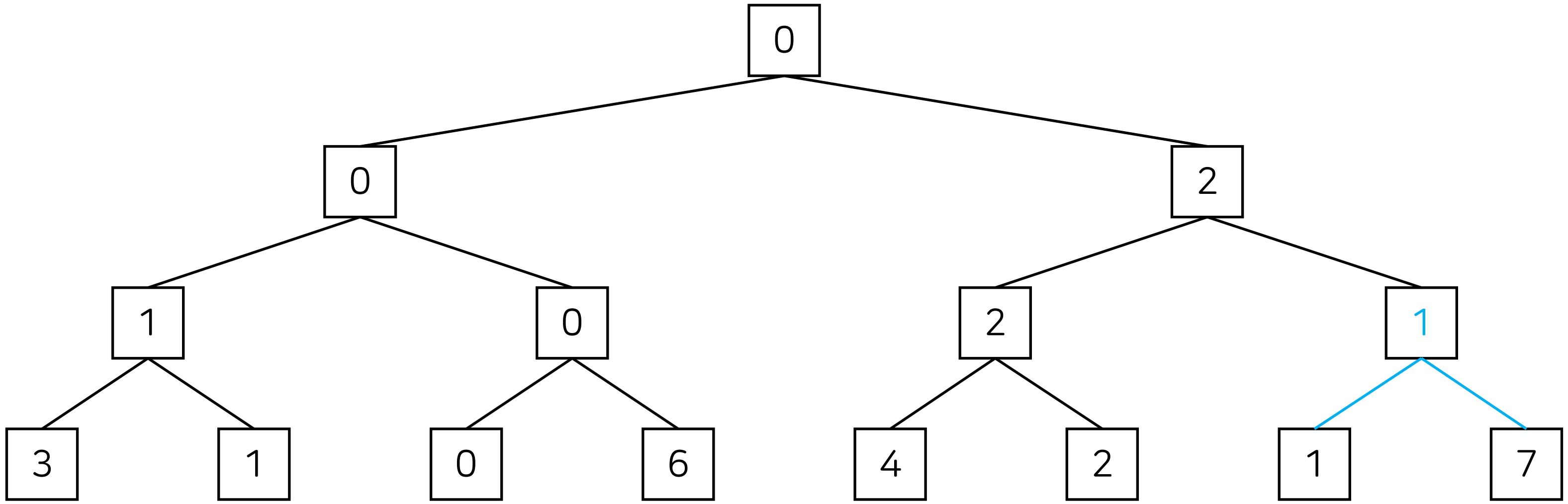
- Segment Tree
  - 원소 수정: 리프 정점 수정하고, 부모 정점 타고 가면서 갱신
    - O(log N)개의 정점만 갱신하면 됨
  - Update(6, 1)



# Segment Tree

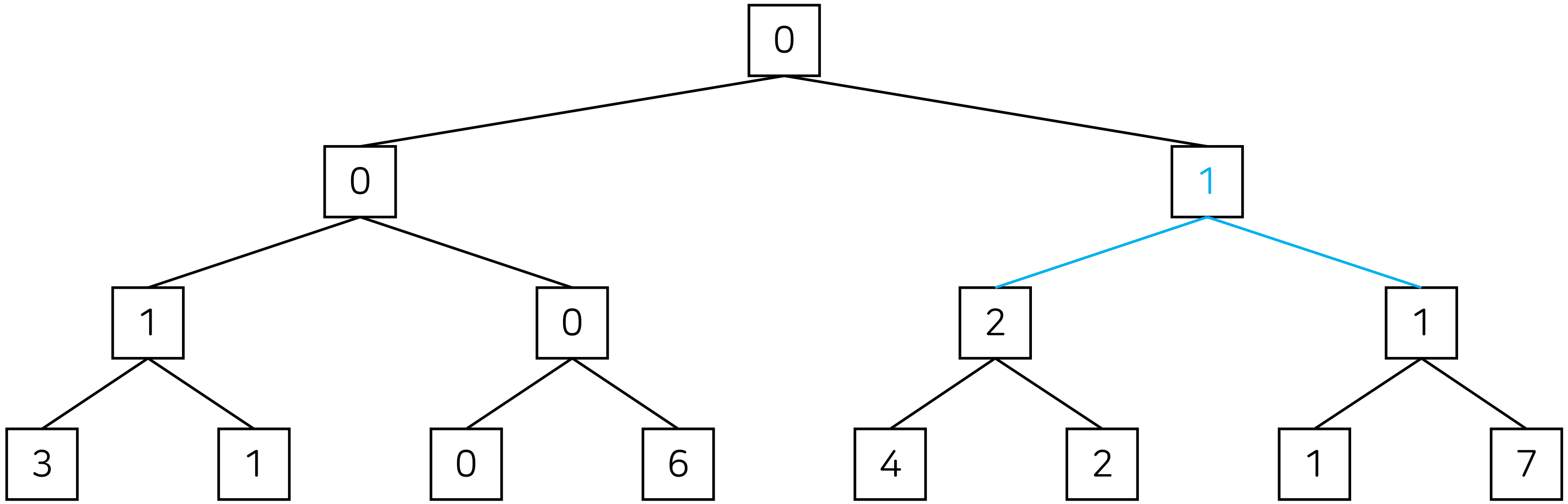
- Segment Tree
  - 원소 수정: 리프 정점 수정하고, 부모 정점 타고 가면서 갱신
    - O(log N)개의 정점만 갱신하면 됨

  - Update(6, 1)



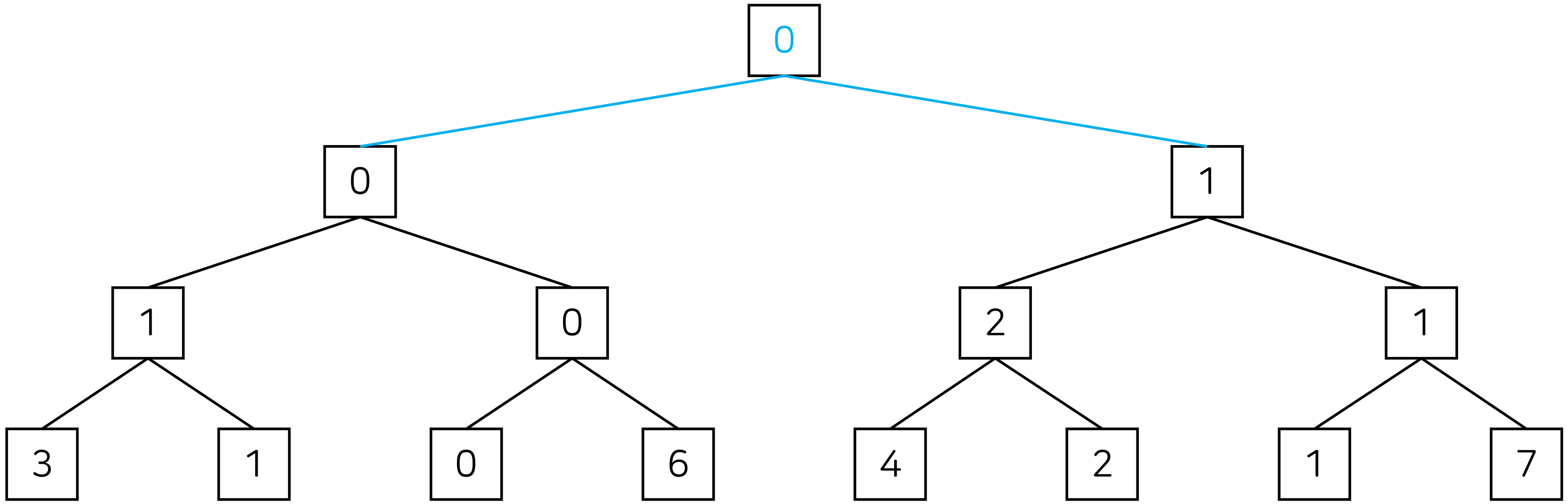
# Segment Tree

- Segment Tree
  - 원소 수정: 리프 정점 수정하고, 부모 정점 타고 가면서 갱신
    - O(log N)개의 정점만 갱신하면 됨
  - Update(6, 1)



# Segment Tree

- Segment Tree
  - 원소 수정: 리프 정점 수정하고, 부모 정점 타고 가면서 갱신
    - O(log N)개의 정점만 갱신하면 됨
  - Update(6, 1)

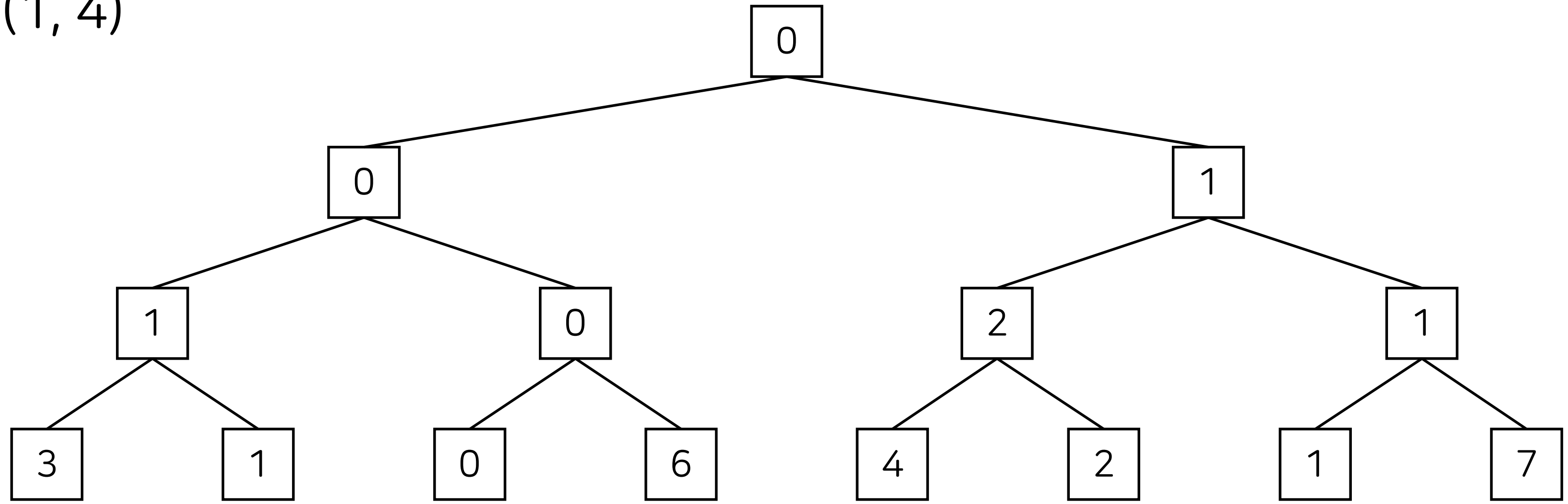


질문?

# Segment Tree

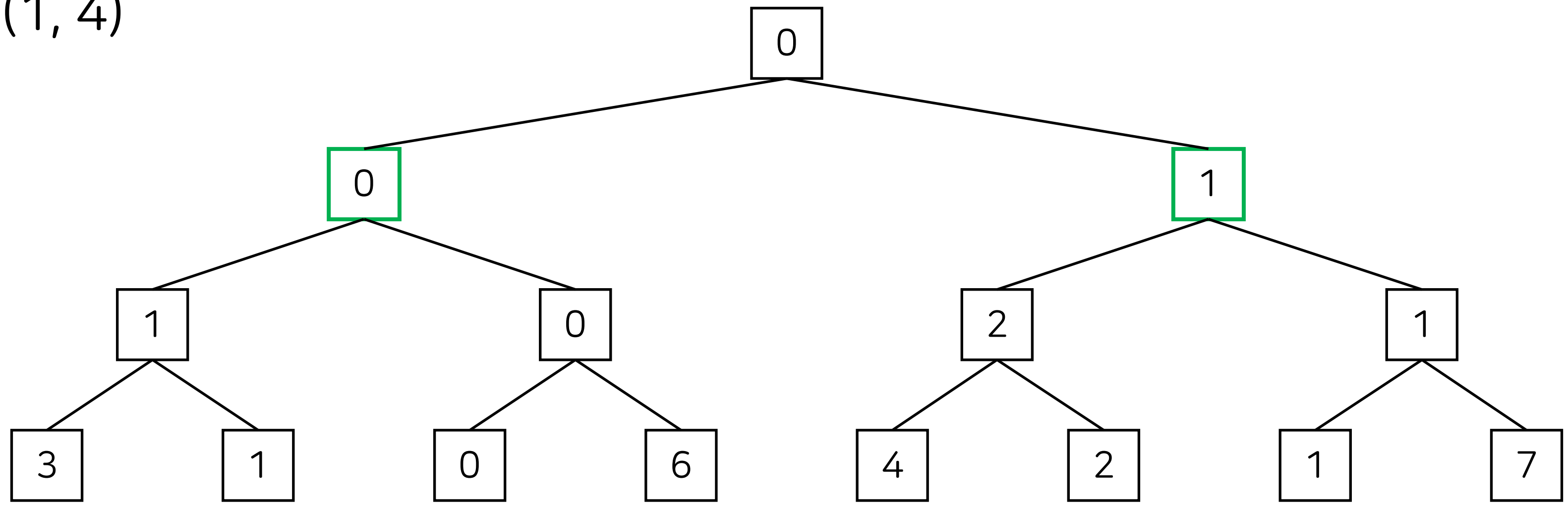
- Segment Tree
  - 구간 쿼리는 조금 복잡함
    - 구하고자 하는 구간을 [l, r], x번 정점이 관리하는 구간을 [$s_x$, $e_x$]라고 하자.
    - 만약 $r < s_x \mid\mid e_x < l$ 이면 [$s_x$, $e_x$]와 [l, r]의 교집합이 존재하지 않음 → 더 보지 않아도 됨
    - 만약 $l \le s_x$ && $e_x \le r$ 이면 정점이 관리하는 구간이 [l, r]에 완전히 포함됨 → x번 정점의 값 반영
    - 두 경우 모두 해당하지 않으면 구간의 일부만 겹치는 경우 → 자식 정점에 대해 재귀 호출

  - Query(1, 4)



# Segment Tree

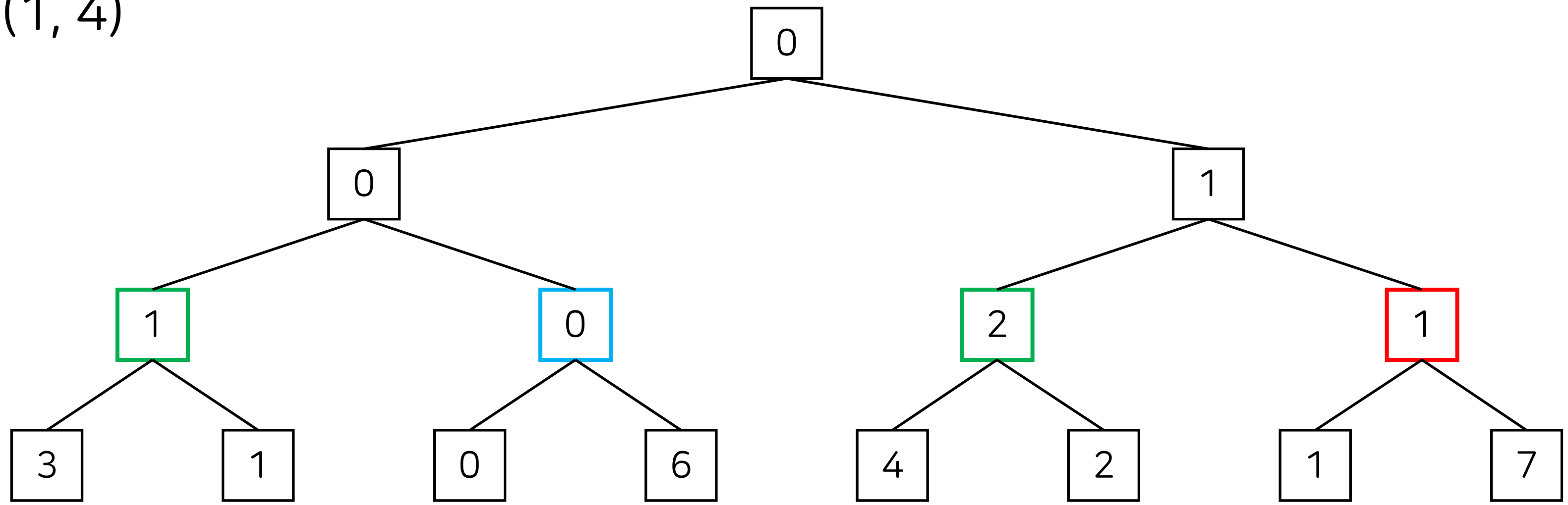
- Segment Tree
  - 구간 쿼리는 조금 복잡함
    - 구하고자 하는 구간을 [l, r], x번 정점이 관리하는 구간을 [$s_x$, $e_x$]라고 하자.
    - 만약 $r < s_x \,||\, e_x < l$ 이면 [$s_x$, $e_x$]와 [l, r]의 교집합이 존재하지 않음 → 더 보지 않아도 됨
    - 만약 $l \le s_x \,\&\&\, e_x \le r$ 이면 정점이 관리하는 구간이 [l, r]에 완전히 포함됨 → x번 정점의 값 반영
    - 두 경우 모두 해당하지 않으면 구간의 일부만 겹치는 경우 → 자식 정점에 대해 재귀 호출
  - Query(1, 4)



# Segment Tree

- Segment Tree
  - 구간 쿼리는 조금 복잡함
    - 구하고자 하는 구간을 [l, r], x번 정점이 관리하는 구간을 $[s_x, e_x]$라고 하자.
    - 만약 $r < s_x \,||\, e_x < l$ 이면 $[s_x, e_x]$와 [l, r]의 교집합이 존재하지 않음 → 더 보지 않아도 됨
    - 만약 $l \le s_x \,\&\&\, e_x \le r$ 이면 정점이 관리하는 구간이 [l, r]에 완전히 포함됨 → x번 정점의 값 반영
    - 두 경우 모두 해당하지 않으면 구간의 일부만 겹치는 경우 → 자식 정점에 대해 재귀 호출

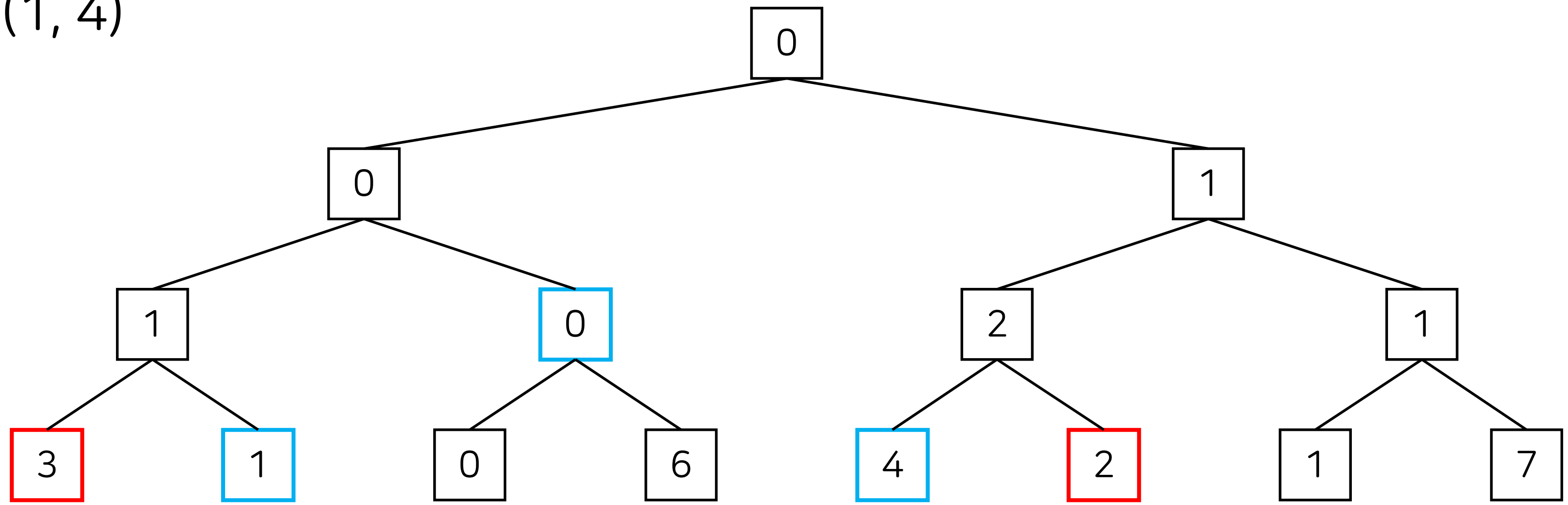
  - Query(1, 4)



# Segment Tree

- Segment Tree
  - 구간 쿼리는 조금 복잡함
    - 구하고자 하는 구간을 [l, r], x번 정점이 관리하는 구간을 [$s_x$, $e_x$]라고 하자.
    - 만약 $r < s_x \,||\, e_x < l$ 이면 [$s_x$, $e_x$]와 [l, r]의 교집합이 존재하지 않음 → 더 보지 않아도 됨
    - 만약 $l \le s_x \,\&\&\, e_x \le r$ 이면 정점이 관리하는 구간이 [l, r]에 완전히 포함됨 → x번 정점의 값 반영
    - 두 경우 모두 해당하지 않으면 구간의 일부만 겹치는 경우 → 자식 정점에 대해 재귀 호출

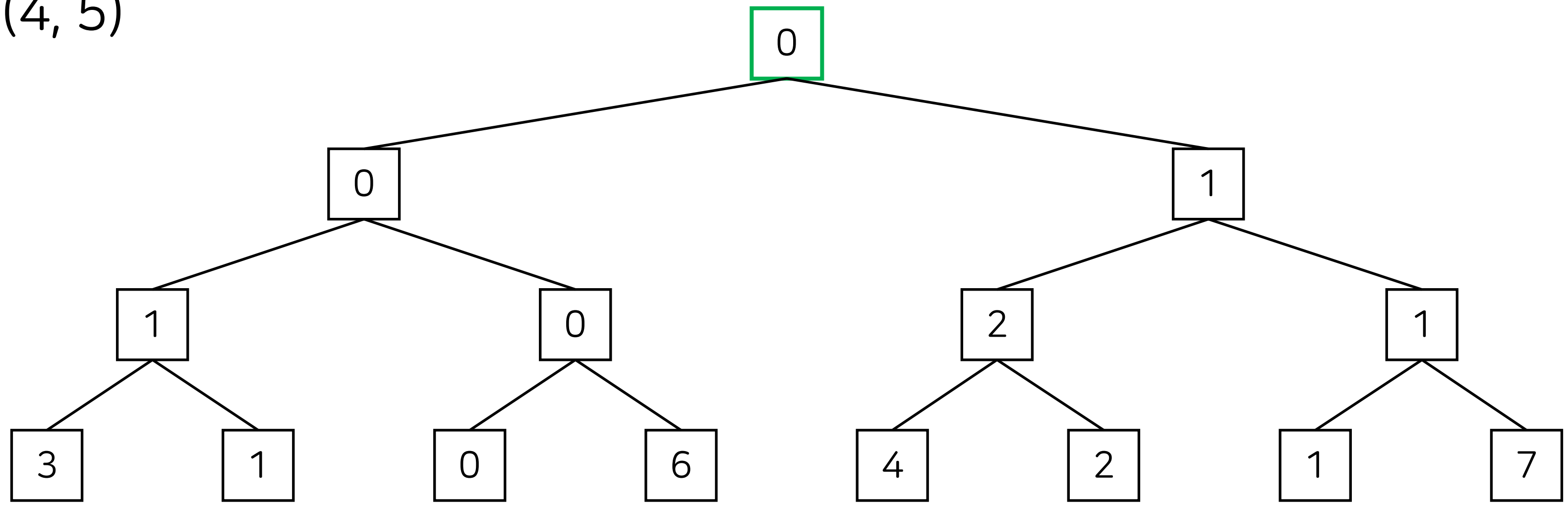
  - Query(1, 4)



# Segment Tree

- Segment Tree
  - 구간 쿼리는 조금 복잡함
    - 구하고자 하는 구간을 [l, r], x번 정점이 관리하는 구간을 [$s_x$, $e_x$]라고 하자.
    - 만약 $r < s_x \,||\, e_x < l$ 이면 [$s_x$, $e_x$]와 [l, r]의 교집합이 존재하지 않음 → 더 보지 않아도 됨
    - 만약 $l \le s_x \,\&\&\, e_x \le r$ 이면 정점이 관리하는 구간이 [l, r]에 완전히 포함됨 → x번 정점의 값 반영
    - 두 경우 모두 해당하지 않으면 구간의 일부만 겹치는 경우 → 자식 정점에 대해 재귀 호출

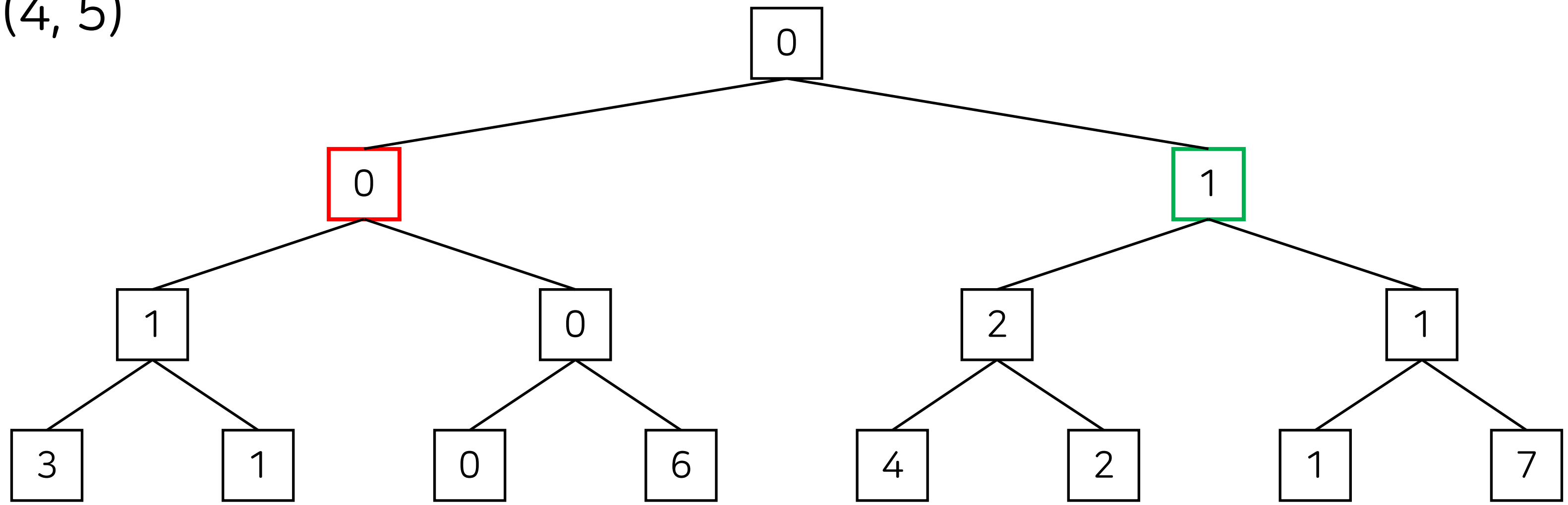
- Query(4, 5)



# Segment Tree

- Segment Tree
  - 구간 쿼리는 조금 복잡함
    - 구하고자 하는 구간을 [l, r], x번 정점이 관리하는 구간을 [$s_x$, $e_x$]라고 하자.
    - 만약 $r < s_x \,||\, e_x < l$ 이면 [$s_x$, $e_x$]와 [l, r]의 교집합이 존재하지 않음 → 더 보지 않아도 됨
    - 만약 $l \le s_x$ && $e_x \le r$ 이면 정점이 관리하는 구간이 [l, r]에 완전히 포함됨 → x번 정점의 값 반영
    - 두 경우 모두 해당하지 않으면 구간의 일부만 겹치는 경우 → 자식 정점에 대해 재귀 호출

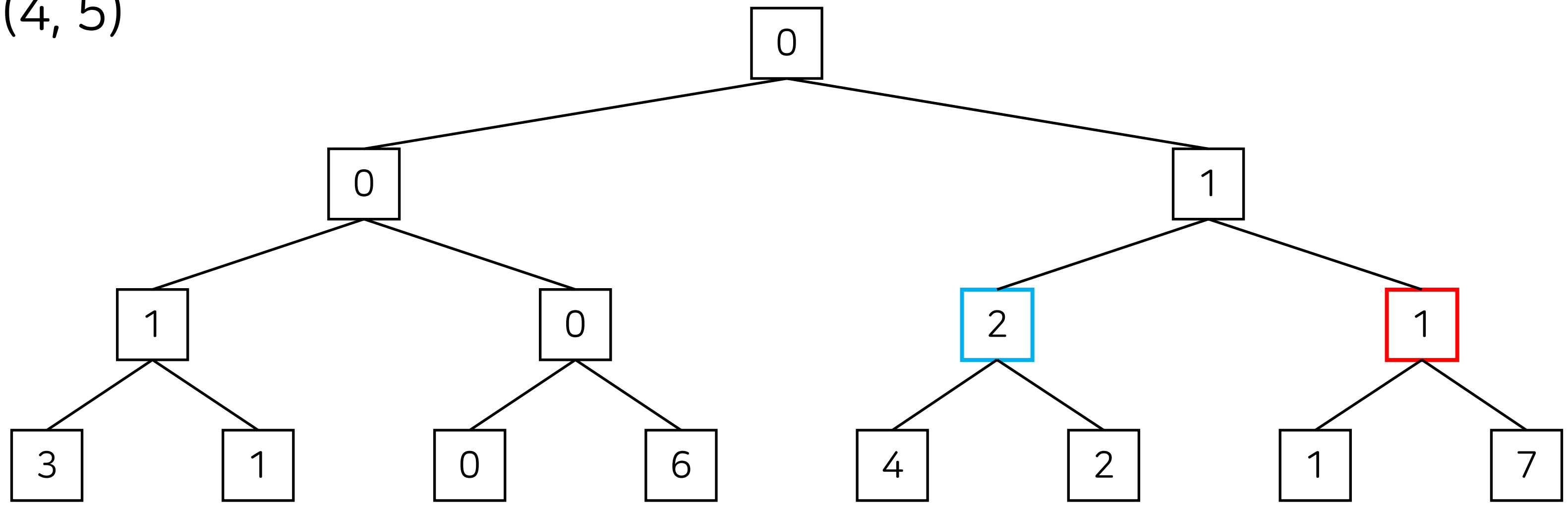
  - Query(4, 5)



# Segment Tree

- Segment Tree
  - 구간 쿼리는 조금 복잡함
    - 구하고자 하는 구간을 [l, r], x번 정점이 관리하는 구간을 [$s_x$, $e_x$]라고 하자.
    - 만약 $r < s_x \| e_x < l$ 이면 [$s_x$, $e_x$]와 [l, r]의 교집합이 존재하지 않음 → 더 보지 않아도 됨
    - 만약 $l \le s_x$ && $e_x \le r$ 이면 정점이 관리하는 구간이 [l, r]에 완전히 포함됨 → x번 정점의 값 반영
    - 두 경우 모두 해당하지 않으면 구간의 일부만 겹치는 경우 → 자식 정점에 대해 재귀 호출

  - Query(4, 5)



# Segment Tree

```
constexpr int SZ = 1 << 17;
int N, Q, A[SZ], T[SZ<<1];

void Init(int node=1, int s=1, int e=N){
    if(s == e){ T[node] = A[s]; return; }
    int m = (s + e) / 2;
    Init(node*2, s, m);
    Init(node*2+1, m+1, e);
    T[node] = min(T[node*2], T[node*2+1]);
}

void Update(int x, int v, int node=1, int s=1, int e=N){
    if(s == e){ T[node] = v; return; }
    int m = (s + e) / 2;
    if(x <= m) Update(x, v, node*2, s, m);
    else Update(x, v, node*2+1, m+1, e);
    T[node] = min(T[node*2], T[node*2+1]);
}

int Query(int l, int r, int node=1, int s=1, int e=N){
    if(r < s || e < l) return 1e9;
    if(l <= s && e <= r) return T[node];
    int m = (s + e) / 2;
    return min(Query(l, r, node*2, s, m), Query(l, r, node*2+1, m+1, e));
}
```

질문?

# Segment Tree

- Query의 시간 복잡도는?
  - case 1. 겹치지 않는 경우: return 1e9;
    - 추가로 다른 정점을 방문하지 않음
  - case 2. 정점이 완전히 구간에 포함되는 경우: return T[node];
    - 추가로 다른 정점을 방문하지 않음
  - case 3. 구간이 일부 겹치는 경우: 자식 정점에 대해 재귀 호출
    - case 3-1. [s, m]과 [l, r]만 겹치는 경우 왼쪽 자식만 방문
    - case 3-2. [m+1, e]와 [l, r]만 겹치는 경우 오른쪽 자식만 방문
    - case 3-3. 둘 다 겹치는 경우 양쪽 자식 모두 방문

# Segment Tree

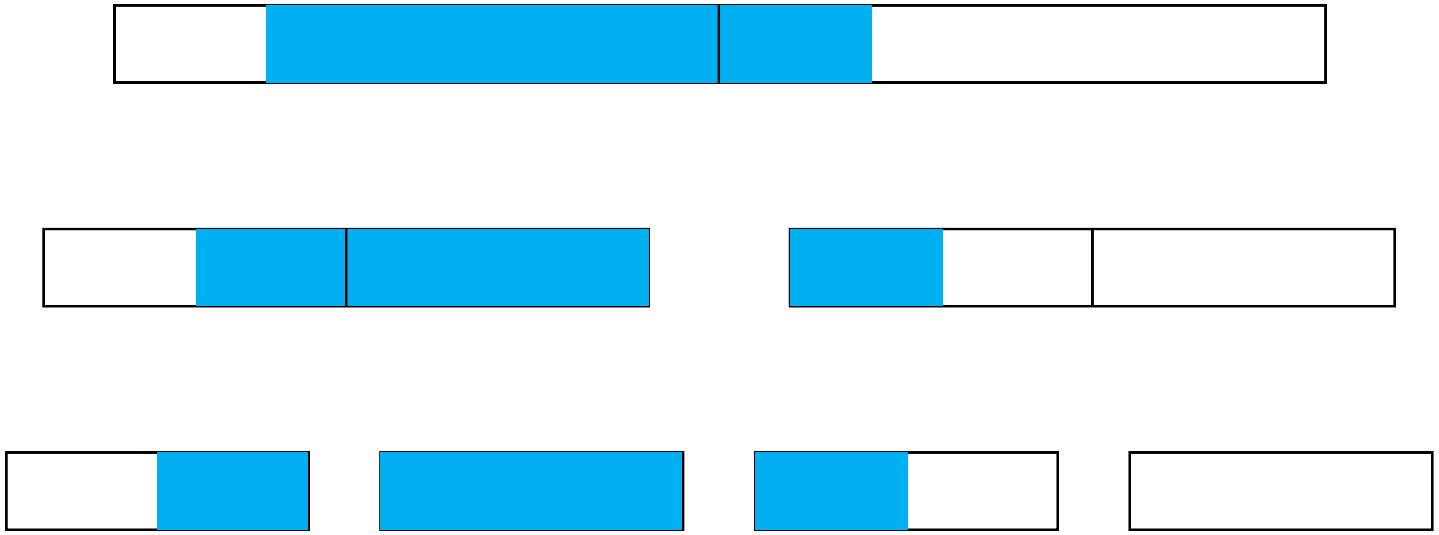
- Query의 시간 복잡도는?
  - case 1, 2: 방문하는 정점의 개수를 증가시키지 않음
  - case 3
    - case 3-1, 3-2: 다음 깊이에서 방문하는 정점 개수 1 증가
    - case 3-3: 다음 깊이에서 방문하는 정점 개수 2 증가

    - case 3-3을 조상으로 갖는 정점은 한 개 이상의 자식 정점이 case 1 또는 2에 해당함
      - $s_x$ 또는 $e_x$가 항상 [l, r]에 포함된다는 점을 생각해 보자.
    - 따라서 각 깊이마다 최대 2개의 case 3 정점이 존재

  - 방문하는 정점은 최대 O(log N)개

# Segment Tree

- Query의 시간 복잡도는?



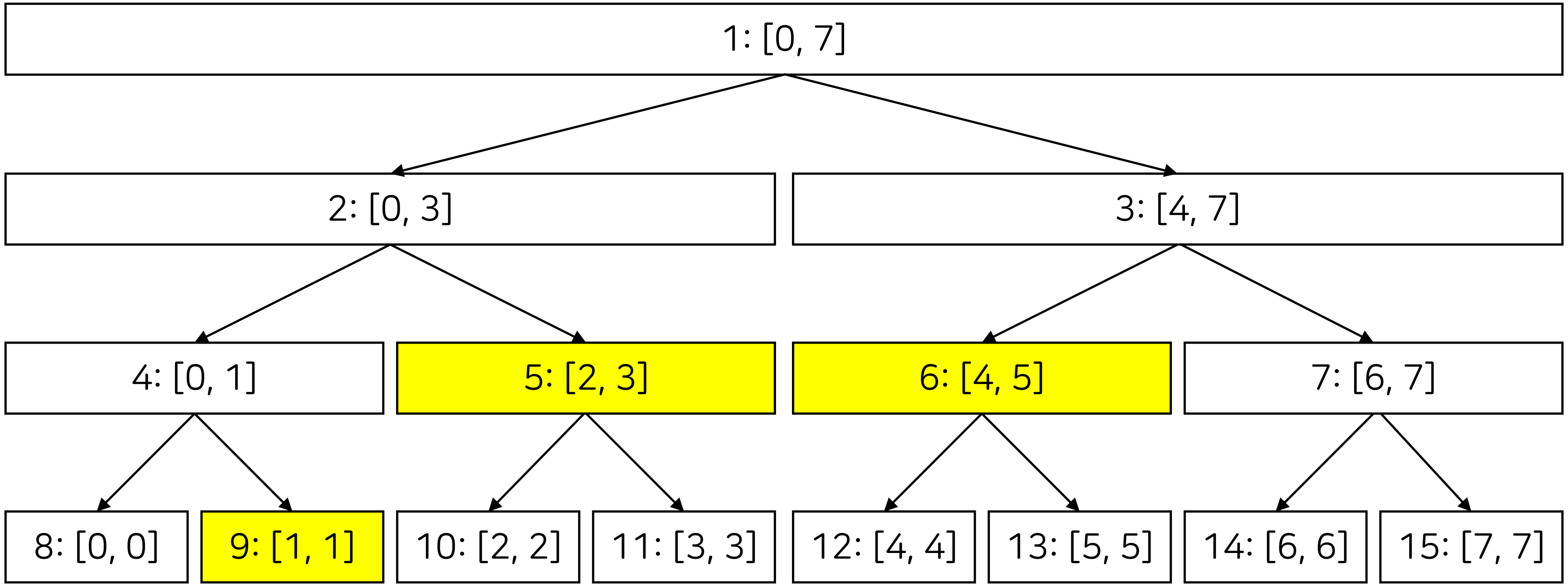
질문?

# Segment Tree

- Segment Tree의 비재귀 구현
  - Update(x, v)
    - x번째 리프 정점의 값을 v로 수정하고, 부모 정점 따라가면서 갱신
    - x += $2^k$; T[x] = v;
    - while(x /= 2) T[x] = min(T[x*2], T[x*2+1]);
  - Query(l, r)
    - 각각의 깊이에서 case 3인 정점은 최대 2개
    - case 3인 정점은 case 2인 정점보다 바깥쪽에 위치
    - case 2인 정점들의 위치를 트래킹하는 방식으로 구현

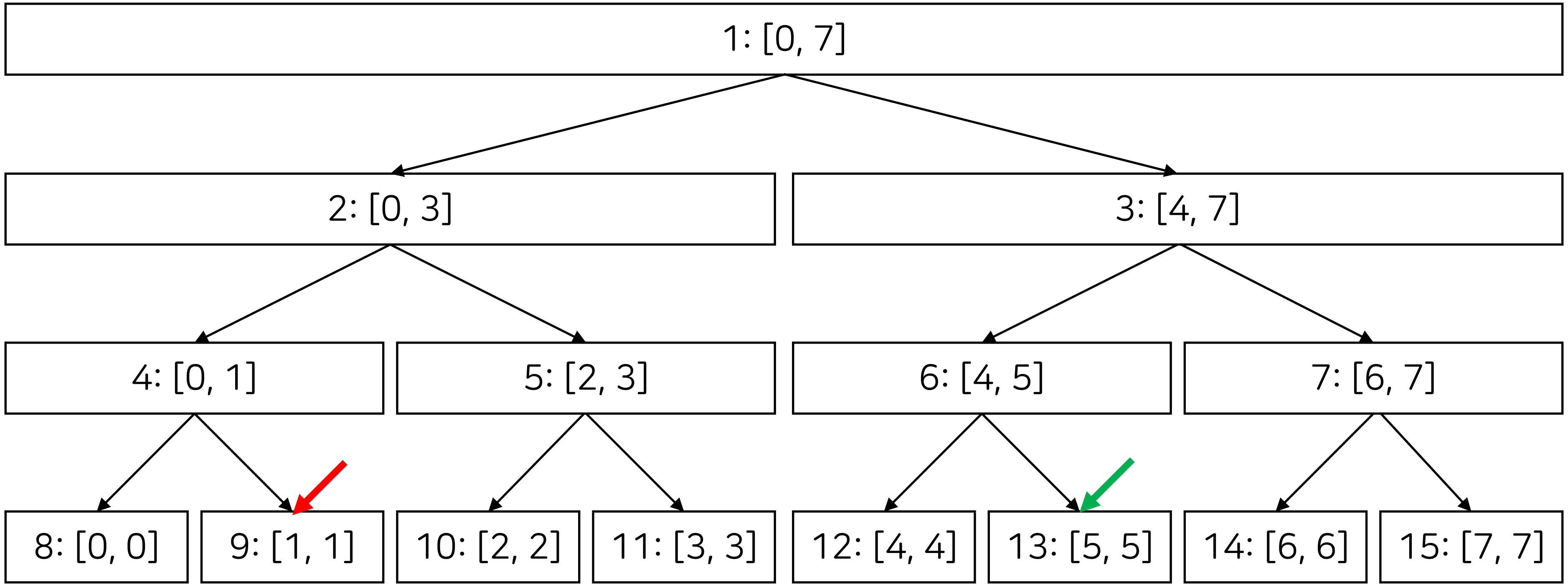
# Segment Tree

- Segment Tree의 비재귀 구현
  - Query(1, 5)에서 case 2에 해당하는 정점들을 구해보자.



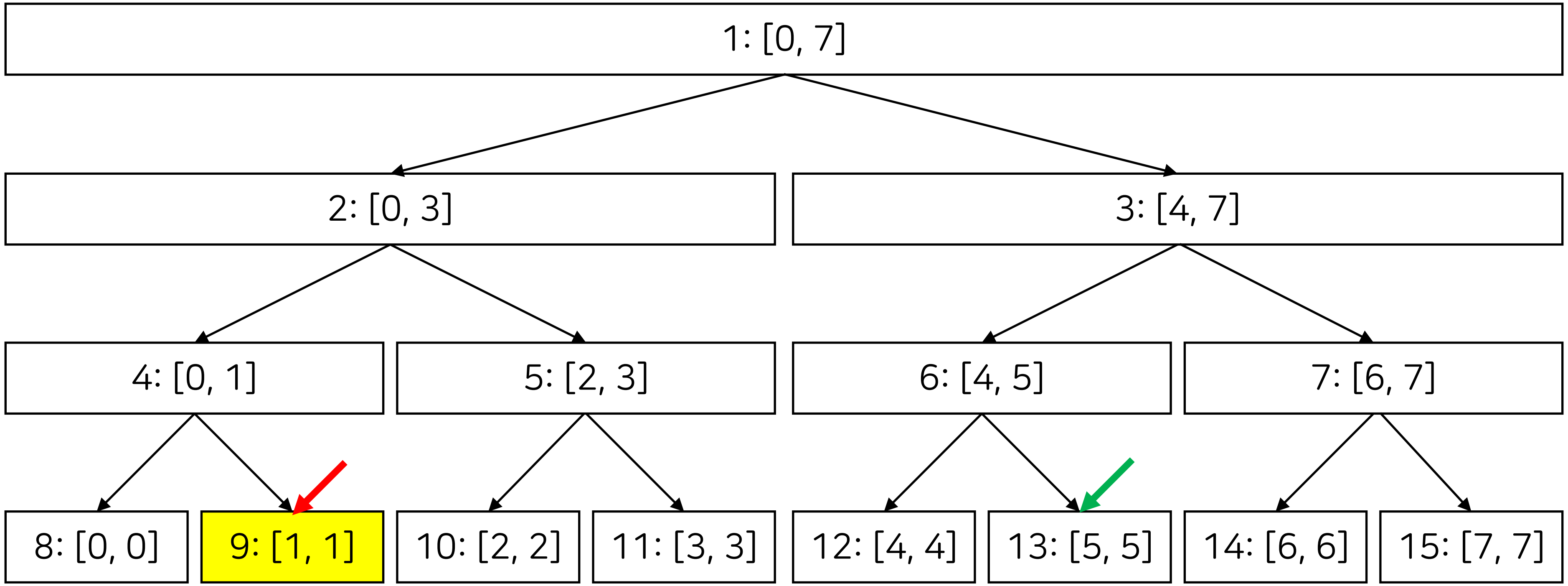
# Segment Tree

- Segment Tree의 비재귀 구현
  - Query(1, 5)에서 case 2에 해당하는 정점들을 구해보자.
    - 1번째 리프 정점과 5번째 리프 정점에서 시작
    - 각각 l, r이라고 하자. l과 r 모두 case 2에 해당한다.



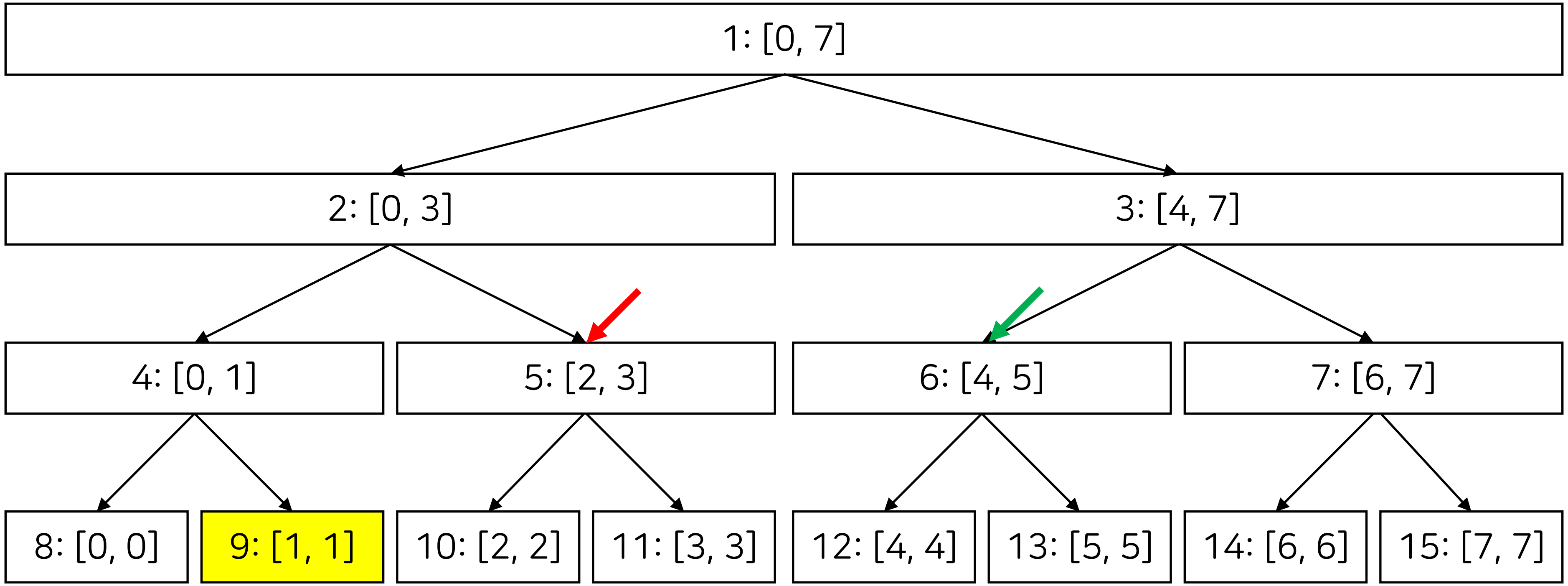
# Segment Tree

- Segment Tree의 비재귀 구현
  - Query(1, 5)에서 case 2에 해당하는 정점들을 구해보자.
    - l은 오른쪽 자식, l의 부모는 case 2에 해당하지 않으므로 l의 값을 결과에 반영하고 l++
    - r은 오른쪽 자식, r의 부모는 case 2에 해당하므로 r 대신 r의 부모만 고려해도 됨



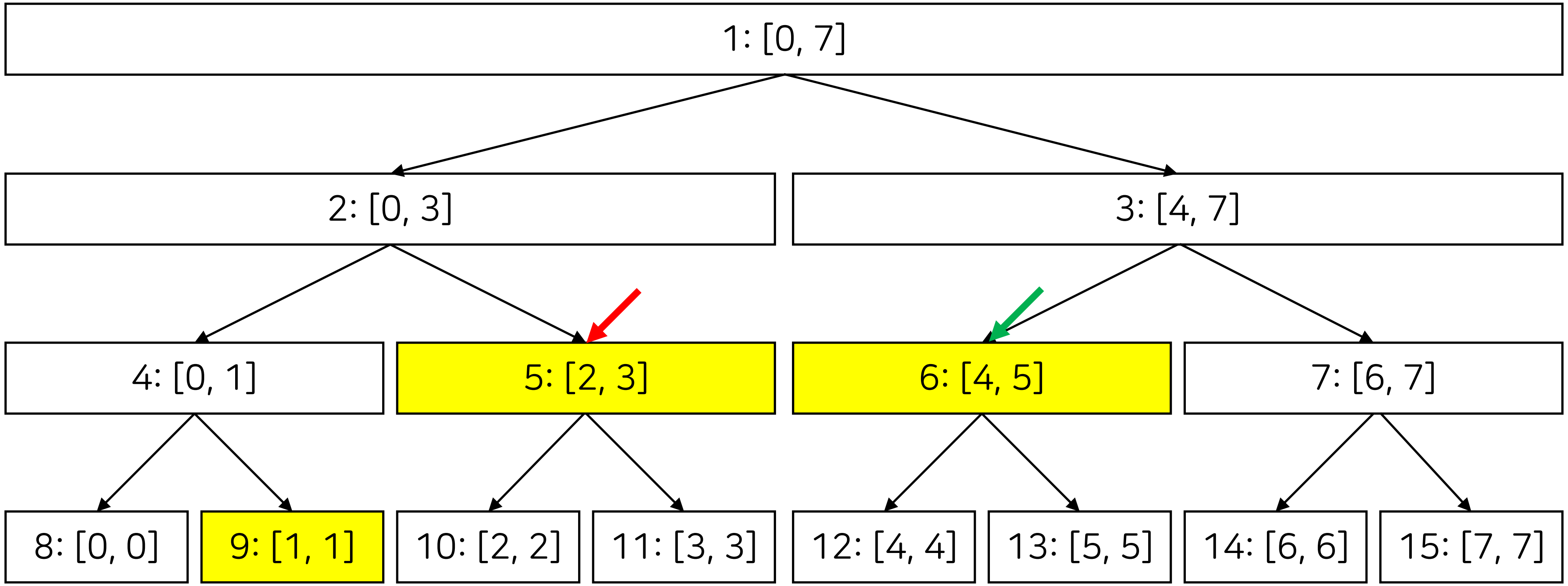
# Segment Tree

- Segment Tree의 비재귀 구현
  - Query(1, 5)에서 case 2에 해당하는 정점들을 구해보자.
    - 현재 깊이에 있는 정점을 모두 봤으므로 l과 r을 한 칸 위로 이동
    - l과 r 모두 case 2에 해당하는 것에 주목



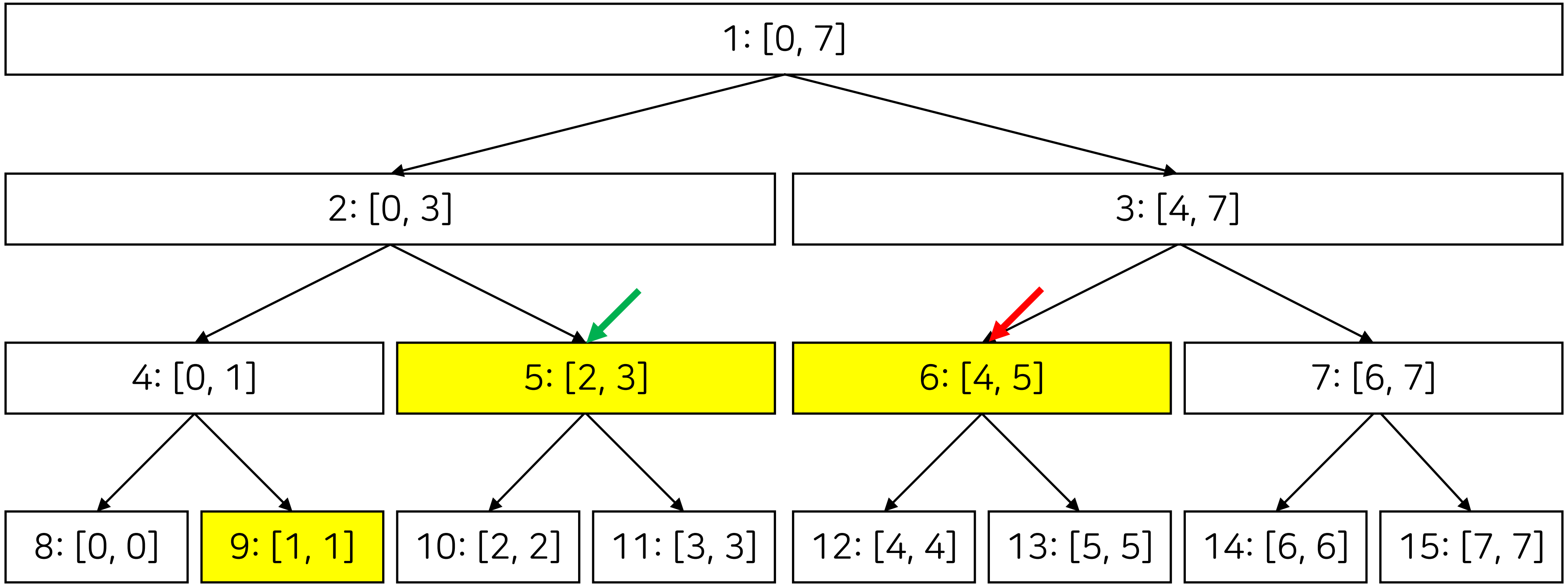
# Segment Tree

- Segment Tree의 비재귀 구현
  - Query(1, 5)에서 case 2에 해당하는 정점들을 구해보자.
    - l은 오른쪽 자식, l의 부모 정점은 case 2에 해당하지 않으므로 l의 값을 결과에 반영하고 l++
    - r은 왼쪽 자식, r의 부모 정점은 case 2에 해당하지 않으므로 r의 값을 결과에 반영하고 r--



# Segment Tree

- Segment Tree의 비재귀 구현
  - Query(1, 5)에서 case 2에 해당하는 정점들을 구해보자.
    - l > r이 되었으므로 종료



# Segment Tree

- Segment Tree의 비재귀 구현
  - [L, R] 구간의 값을 구하는 경우, L번째 리프 정점 l과 R번째 리프 정점 r에서 시작
  - 이때 l, r은 모두 case 2에 해당
  - l ≤ r이면 아래 과정을 반복
    - l의 정점 번호가 홀수인지 짝수인지 확인
      - l이 짝수면 왼쪽 자식: l의 부모 정점이 [L, R]에 온전히 포함되므로 l의 부모만 고려해도 됨
      - l이 홀수면 오른쪽 자식: l의 부모 정점이 [L, R]에 포함되지 않으므로 l의 결과를 가져가야 함
        - res += T[l++];
    - r의 정점 번호가 홀수인지 짝수인지 확인
      - r이 홀수면 오른쪽 자식: r의 부모 정점이 [L, R]에 온전히 포함되므로 r의 부모만 고려해도 됨
      - r이 짝수면 왼쪽 자식: r의 부모 정점이 [L, R]에 포함되지 않으므로 r의 결과를 가져가야 함
        - res += T[r--];

# Segment Tree

```cpp
constexpr int SZ = 1 << 17;
int N, Q, A[SZ], T[SZ << 1];

void Build(){
    memset(T, 0x3f, sizeof T);
    for(int i=1; i<=N; i++) T[i|SZ] = A[i];
    for(int i=SZ-1; i>=1; i--) T[i] = min(T[i << 1], T[i << 1 | 1]);
}

void Update(int x, ll v){
    x |= SZ; T[x] = v;
    while(x >>= 1) T[x] = min(T[x << 1], T[x << 1 | 1]);
}

int Query(int l, int r){
    l |= SZ; r |= SZ;
    int ret = 0x3f3f3f3f;
    while(l <= r){
        if(l & 1) ret = min(ret, T[l++]);
        if(~r & 1) ret = min(ret, T[r--]);
        l >>= 1; r >>= 1;
    }
    return ret;
}
```

질문?

# Segment Tree + Binary Search

# Segment Tree + Binary Search

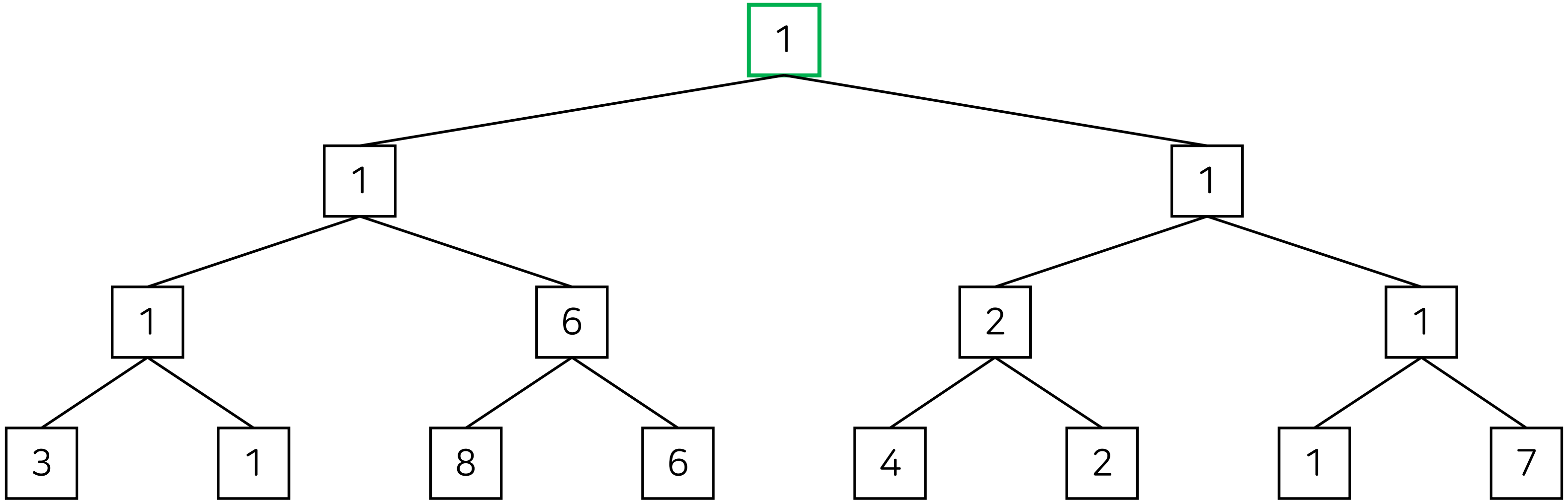
- BOJ 14427 수열과 쿼리 15
  - 1: $A_i$를 v로 바꾼다.
  - 2: 수열에서 최솟값의 인덱스를 출력한다. 그러한 값이 여러 개라면 가장 작은 인덱스 출력

  - 파라메트릭 서치를 이용해 정답을 구할 수 있음
    - 수열 전체의 최솟값을 V라고 하자.
    - Decision(x): $A_1$, $A_2$, … , $A_x$에 V가 있는가?
    - 세그먼트 트리를 이용해 결정 문제를 O(log N)에 해결하면 전체 시간 복잡도는 $O(\log^2 N)$

  - 세그먼트 트리의 성질을 활용하면 O(log N)에 해결 가능

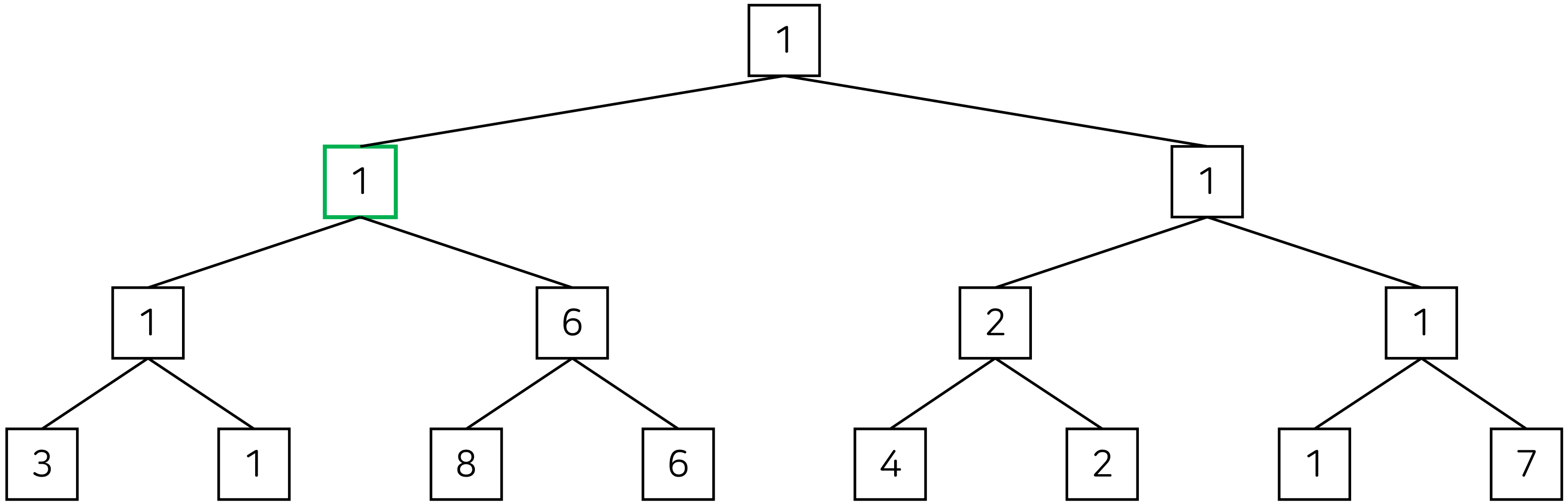
# Segment Tree + Binary Search

- BOJ 14427 수열과 쿼리 15
  - 루트 정점에서 출발, 루트 정점은 항상 최솟값을 갖고 있음
  - 만약 왼쪽 자식 정점의 값이 V라면 왼쪽 절반에 V가 존재한다는 뜻
    - 왼쪽 자식으로 이동
  - 왼쪽 자식 정점의 값이 V가 아니라면 오른쪽 절반에 V가 존재
    - 오른쪽 자식으로 이동
  - 이분 탐색과 동일한 과정



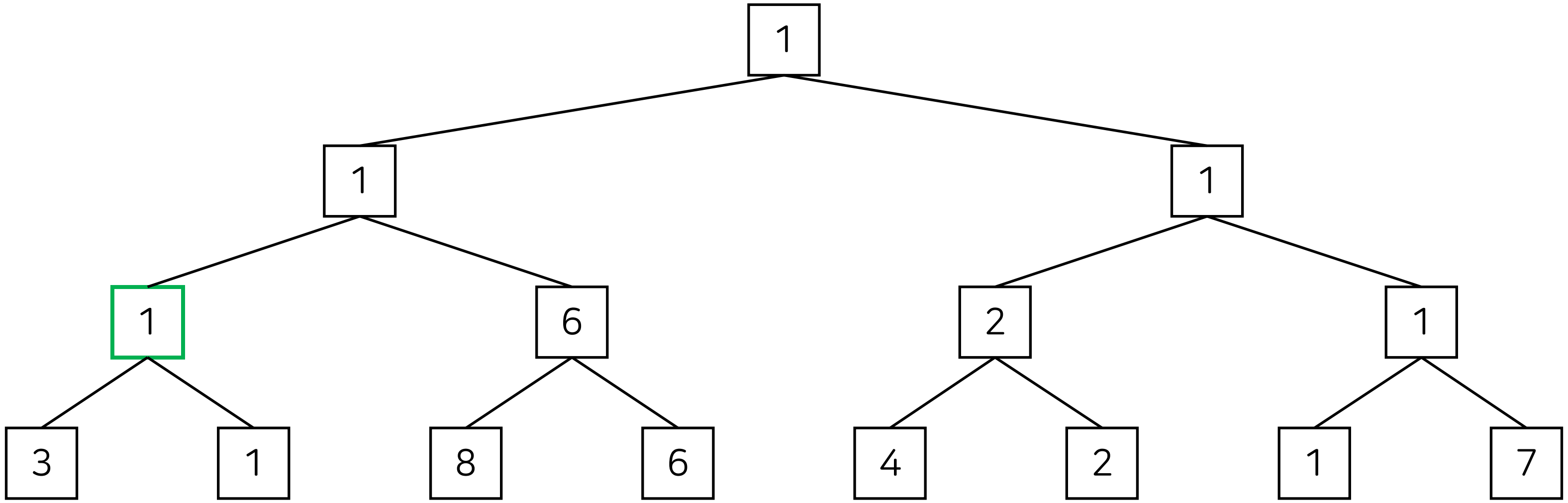
# Segment Tree + Binary Search

- BOJ 14427 수열과 쿼리 15
  - 루트 정점에서 출발, 루트 정점은 항상 최솟값을 갖고 있음
  - 만약 왼쪽 자식 정점의 값이 V라면 왼쪽 절반에 V가 존재한다는 뜻
    - 왼쪽 자식으로 이동
  - 왼쪽 자식 정점의 값이 V가 아니라면 오른쪽 절반에 V가 존재
    - 오른쪽 자식으로 이동
  - 이분 탐색과 동일한 과정



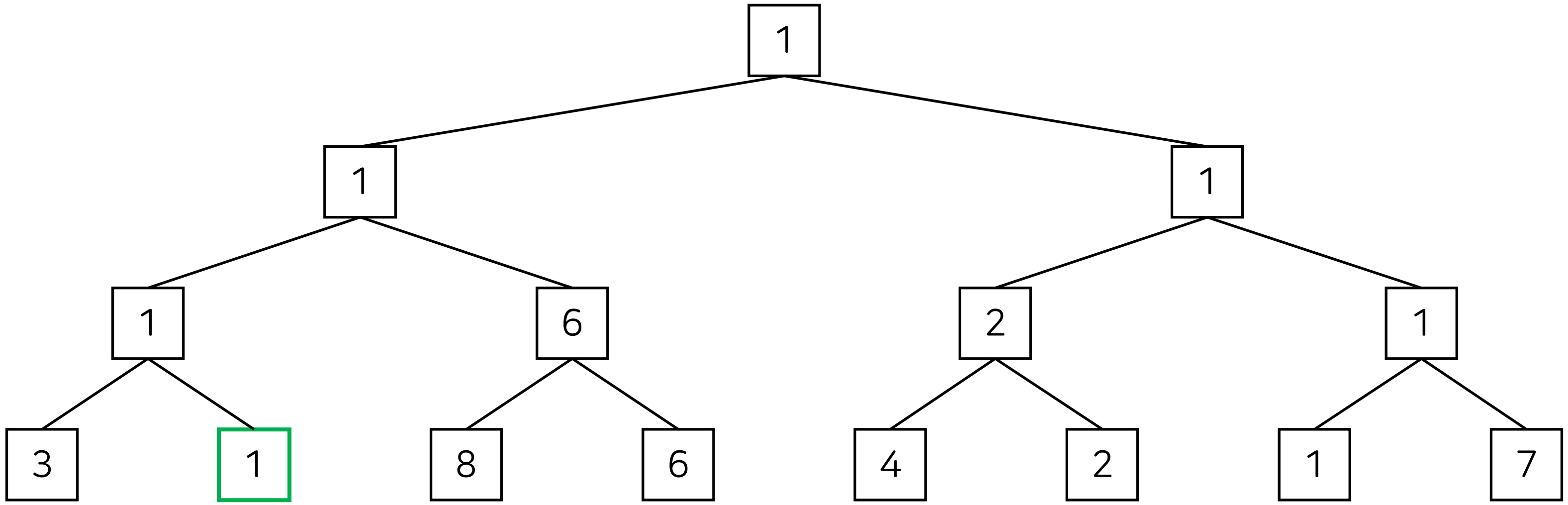
# Segment Tree + Binary Search

- BOJ 14427 수열과 쿼리 15
  - 루트 정점에서 출발, 루트 정점은 항상 최솟값을 갖고 있음
  - 만약 왼쪽 자식 정점의 값이 V라면 왼쪽 절반에 V가 존재한다는 뜻
    - 왼쪽 자식으로 이동
  - 왼쪽 자식 정점의 값이 V가 아니라면 오른쪽 절반에 V가 존재
    - 오른쪽 자식으로 이동
  - 이분 탐색과 동일한 과정



# Segment Tree + Binary Search

- BOJ 14427 수열과 쿼리 15
  - 루트 정점에서 출발, 루트 정점은 항상 최솟값을 갖고 있음
  - 만약 왼쪽 자식 정점의 값이 V라면 왼쪽 절반에 V가 존재한다는 뜻
    - 왼쪽 자식으로 이동
  - 왼쪽 자식 정점의 값이 V가 아니라면 오른쪽 절반에 V가 존재
    - 오른쪽 자식으로 이동
  - 이분 탐색과 동일한 과정



# Segment Tree + Binary Search

```
constexpr int SZ = 1 << 17;
int N, Q, A[SZ], T[SZ << 1];

int Query(){
    int x = 1;
    while(x < SZ){
        if(T[x*2] == T[x]) x = x * 2;
        else x = x * 2 + 1;
    }
    return x - SZ;
}
```

# Segment Tree + Binary Search

```
int Query(int node=1, int s=1, int e=N){
    if(s == e) return s;
    int m = (s + e) / 2;
    if(T[node*2] == T[node]) return Query(node*2, s, m);
    else return Query(node*2+1, m+1, e);
}
```

# 질문?

# Segment Tree + Binary Search

- BOJ 2243 사탕상자
  - 원소가 추가/제거되는 상황에서 k번째로 작은 수를 구하는 문제

  - 파라메트릭 서치로 해결할 수 있음
    - 세그먼트 트리의 i번째 리프 정점에서 i의 개수를 관리
    - 세그먼트 트리의 내부 정점은 구간의 합을 관리

    - Decision(x): k번째 수가 x 이하인가?
      - return sum(1, x) ≥ k;
    - 세그먼트 트리를 이용해 결정 문제를 해결하면 전체 시간 복잡도는 $O(\log^2 N)$
    - 수열과 쿼리 15처럼 해결할 수 있을까?

# Segment Tree + Binary Search

- BOJ 2243 사탕상자
  - 루트 정점에서 시작, 루트 정점의 값이 k 이상인 상황만 생각하자.
  - k번째 수는 루트 정점이 관리하는 구간에 포함되어 있음

  - 만약 왼쪽 자식 정점의 값이 k 이상이라면 k번째 수는 왼쪽 절반에 존재
    - 왼쪽 정점으로 이동
  - 그렇지 않으면 k번째 수는 오른쪽 절반에 존재
    - 오른쪽 정점으로 이동
    - T[x*2]개를 스킵했으므로 k에서 T[x*2]를 빼야 함
    - 오른쪽 자식 정점에서 k-T[x*2]번째 수 = 현재 정점에서 k번째 수

# Segment Tree + Binary Search

```
int kth(int k){
    int x = 1;
    while(x < SZ){
        if(k <= T[x*2]) x = x * 2;
        else k -= T[x*2], x = x * 2 + 1;
    }
    return x - SZ;
}
```

# Segment Tree + Binary Search

```
int Kth(int k, int node=1, int s=0, int e=SZ-1){
    if(s == e) return s;
    int m = (s + e) / 2;
    if(k <= T[node*2]) return Kth(k, node*2, s, m);
    else return Kth(k-T[node*2], node*2+1, m+1, e);
}
```

질문?

# Applications

# Applications

- BOJ 10090 Counting Inversions
  - i < j이면서 A[i] > A[j]인 순서쌍 (i, j)의 개수를 구하는 문제
    - 분할 정복에서 했던 문제

  - 세그먼트 트리를 사용하면 쉽게 해결할 수 있음
    - 앞에 있는 수부터 차례대로 보면서
    - 지금까지 봤던 수 중에서 자신보다 큰 수의 개수를 세면 됨
    - int 말고 long long 사용해야 함

    - for(int i=1; i<=n; i++)
      - res += Sum(A[i]+1, n)
      - Add(A[i], 1)

# Applications

- BOJ 25639 수열과 최대 상승 쿼리
  - 1: $A_i$를 v로 바꾼다.
  - 2: l, r이 주어지면 $l \le i \le j \le r$인 (i, j)에 대해 max(A[j] – A[i])를 출력
    - 구간 [l, r]에서 가장 많이 “상승”하는 값 출력

  - 분할 정복을 이용해 2번 쿼리를 처리하는 방법을 생각해 보자.
    - 중간 지점 m = (l + r) / 2를 잡고
    - [l, m]과 [m+1, r]에서 발생하는 “상승”은 재귀적으로 처리

    - $l \le i \le m < j \le r$인 (i, j)만 고려하면 됨
    - 이건 [l, m]에서 최솟값 구하고 [m+1, e]에서 최댓값을 구하면 됨

# Applications

- BOJ 25639 수열과 최대 상승 쿼리
  - 세그먼트 트리는 분할 정복 과정을 명시적으로 나타낸 구조라고 생각할 수 있음

  - [l, r] 구간을 관리하는 정점에서 아래 세 가지 값을 저장
    - [l, r] 구간 안에서의 최대 상승 res
    - [l, r] 구간의 최솟값 mn
    - [l, r] 구간의 최댓값 mx

  - 두 정점 a, b의 정보를 합치는 것은…
    - 최솟값과 최댓값은 단순히 min/max 사용
    - res가 될 수 있는 후보는 a.res, b.res, 그리고 b.mx – a.mn

# Applications

- pINF = 양의 무한대, nINF = 음의 무한대를 의미

```cpp
using ll = long long;
constexpr ll nINF = 0xc0c0c0c0c0c0c0c0;
constexpr ll pINF = 0x3f3f3f3f3f3f3f3f;

struct Node{
    ll mn, mx, res;
    Node( ) : Node(pINF, nINF, nINF) {}
    Node(ll v) : Node(v, v, 0) {}
    Node(ll mn, ll mx, ll res) : mn(mn), mx(mx), res(res) {}
} T[SZ << 1];

Node operator + (const Node &a, const Node &b){
    ll mn = min(a.mn, b.mn);
    ll mx = max(a.mx, b.mx);
    ll res = max({ a.res, b.res, b.mx - a.mn });
    return Node(mn, mx, res);
}
```

# Applications

- BOJ 25639 수열과 최대 상승 쿼리
  - Node 간의 덧셈 연산은 교환 법칙이 성립하지 않음
  - Query 함수 구현 조심

```
Node Query(int l, int r){
    l |= SZ; r |= SZ;
    Node lv, rv;
    while(l <= r){
        if(l & 1) lv = lv + T[l++];
        if(~r & 1) rv = T[r--] + rv;
        l >>= 1; r >>= 1;
    }
    return lv + rv;
}
```

질문?

# 과제

- 필수
  - 2042 구간 합 구하기
  - 14438 수열과 쿼리 17
  - 10090 Counting Inversions
  - 14427 수열과 쿼리 15
  - 2243 사탕상자
- 심화
  - 25639 수열과 최대 상승 쿼리
  - 16993 연속합과 쿼리
  - 13537 수열과 쿼리 1
  - 2336 굉장한 학생
  - 16136 준하의 정수론 과제
  - 18798 OR과 쿼리