02주차: Lazy Propagation & Plane Sweeping

강사: 김휘수

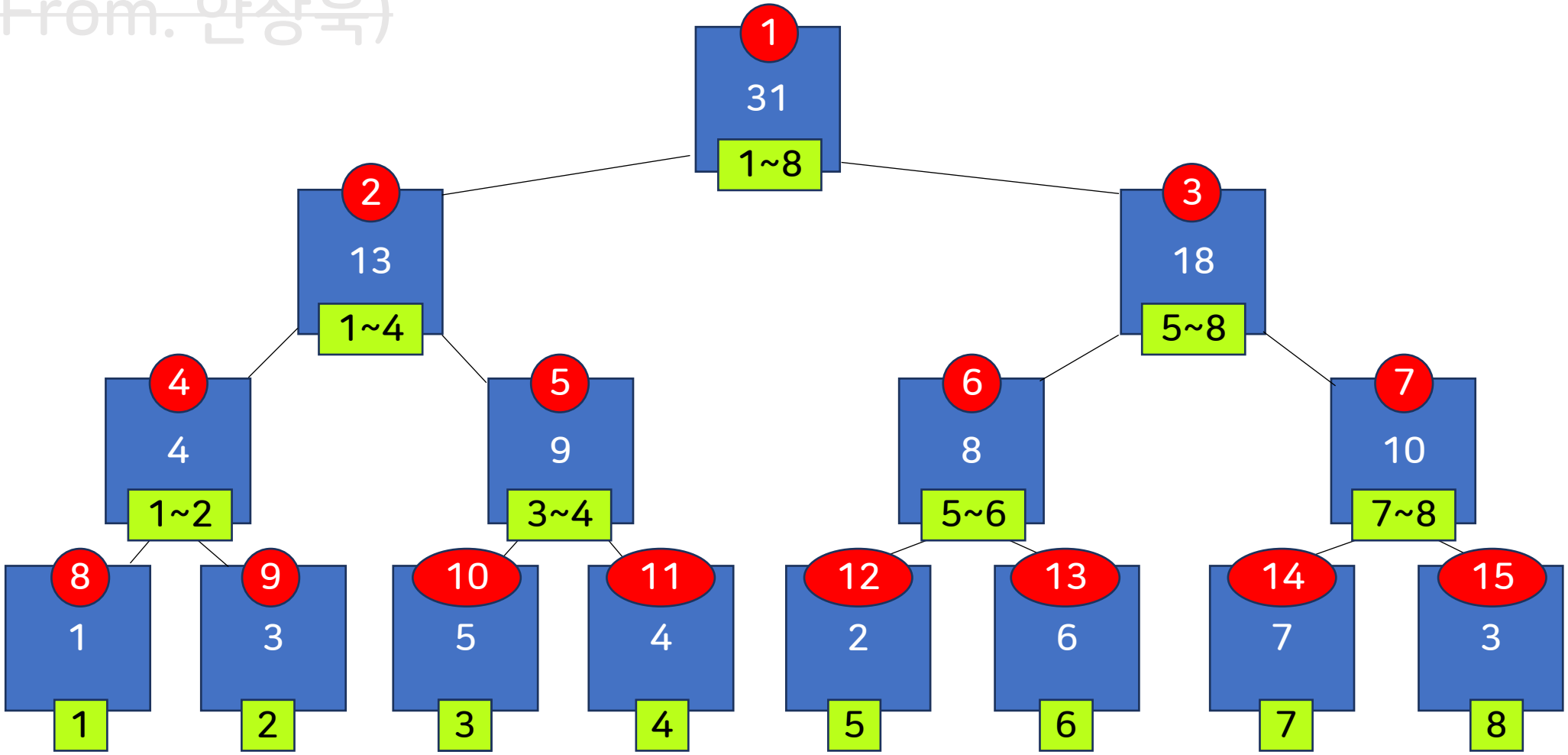
챕터 1: Lazy Propagation

지난 시간에 배운 Seg Tree 기억하고 계시죠?

Segment Tree 복습

(From. 안상욱)

빨간 원의 숫자는 segment tree 배열의 index를 뜻한다.



Segment Tree 복습

(From. 안상욱)

index	1	2	3	4	5	6	7
tree	31	13	18	4	9	8	10

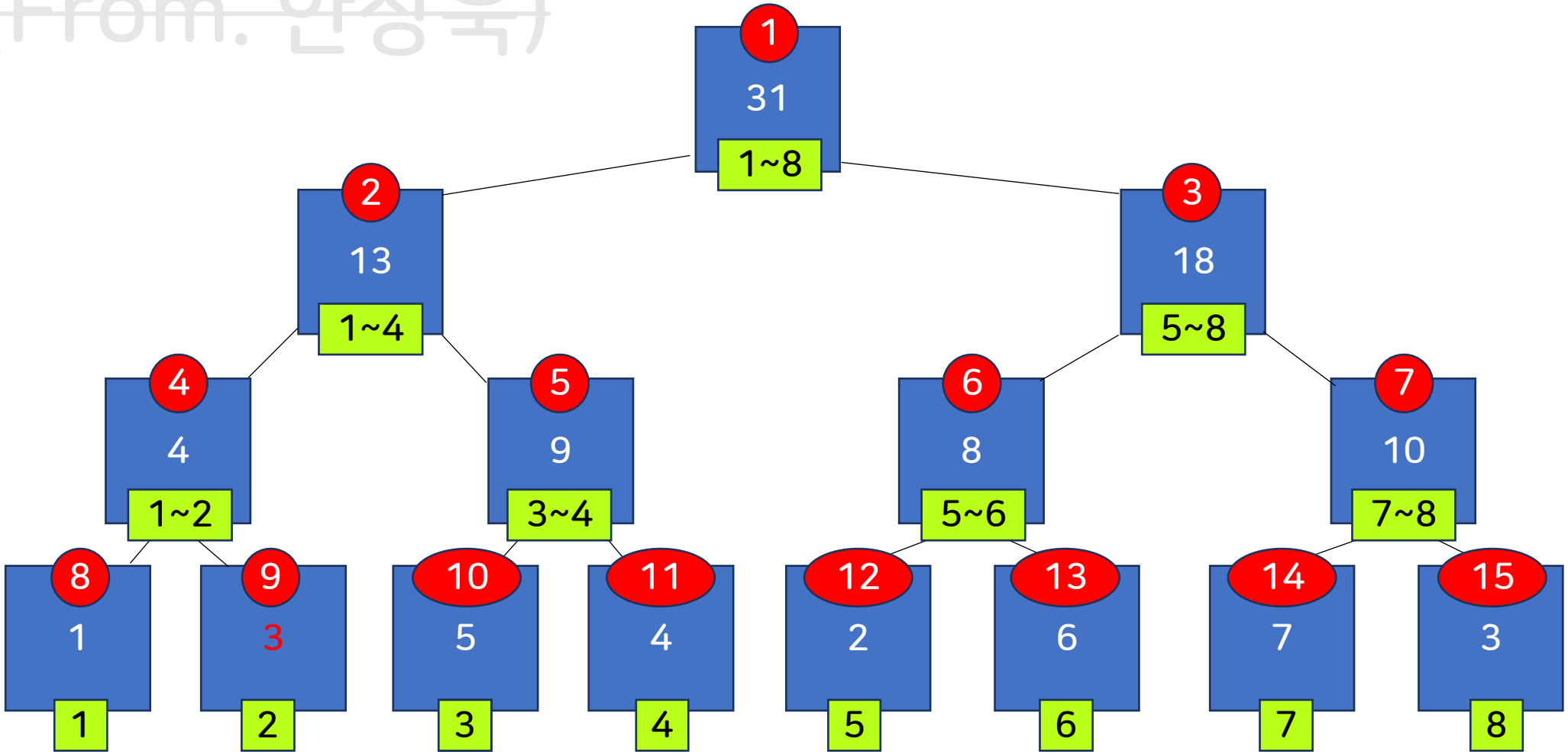
8	9	10	11	12	13	14	15
1	3	5	4	2	6	7	3

이런 모양의 배열이 된다

Update

~~(From. 안상욱)~~

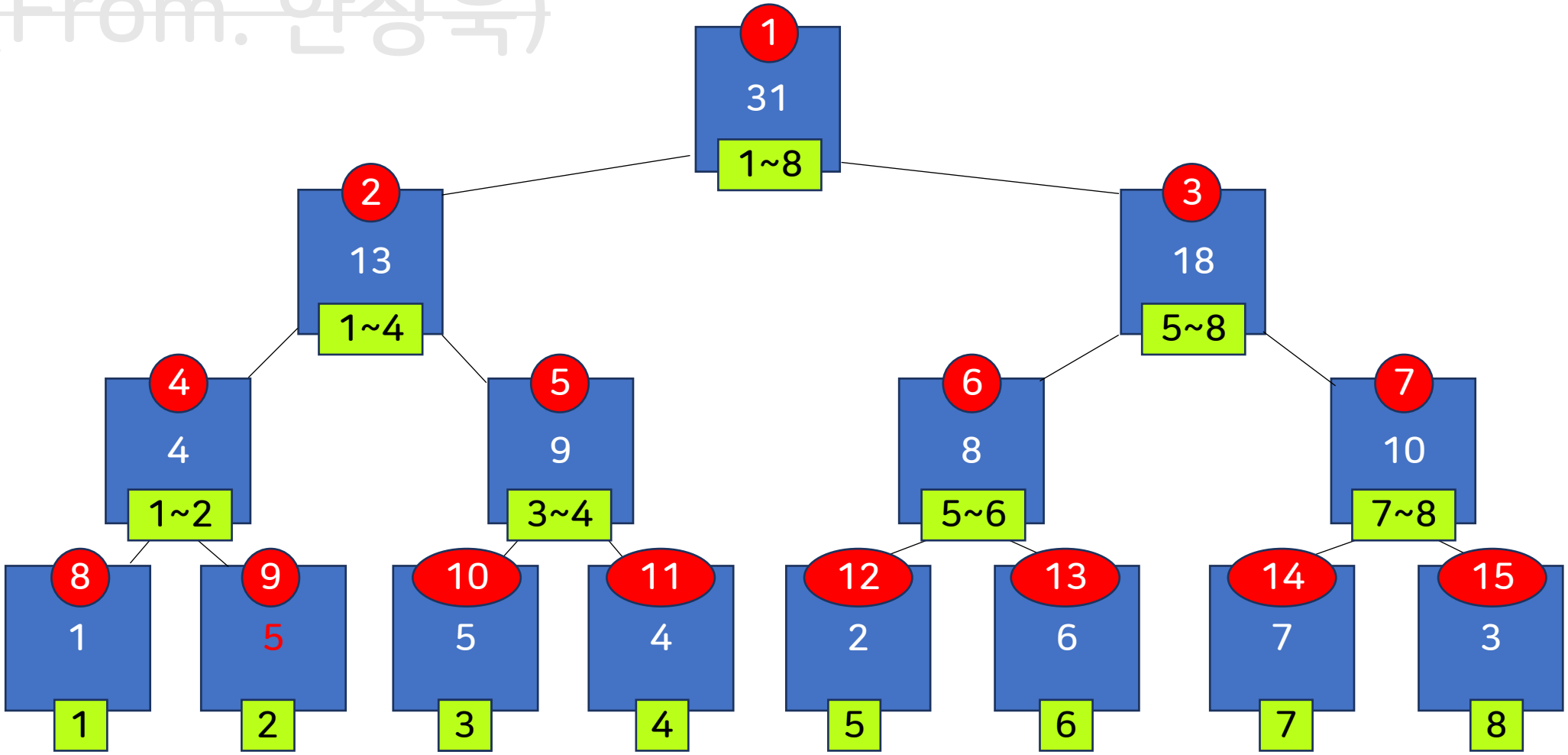
만약 2번 index에 저장된 값이 변한다면????



Update

~~(From. 안상욱)~~

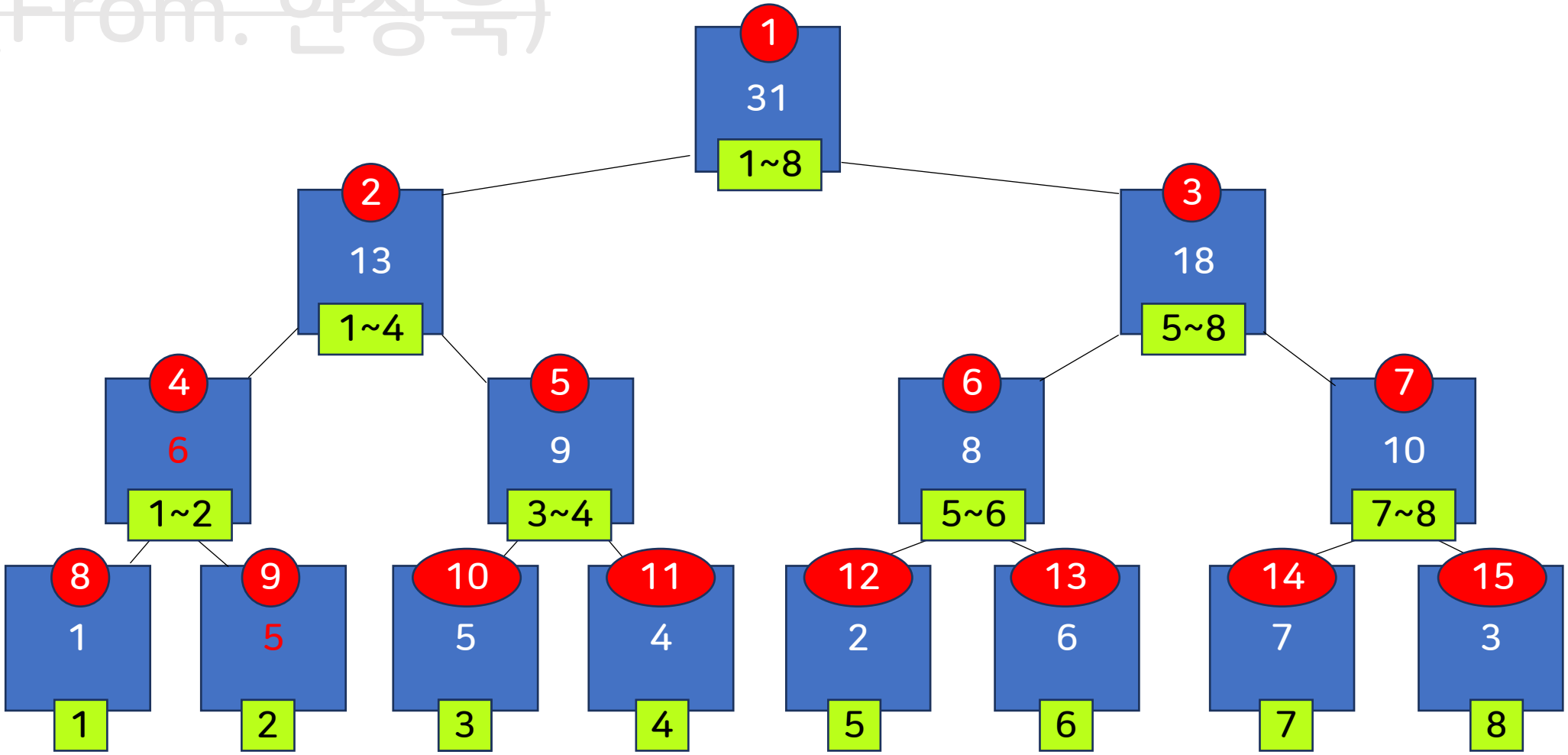
만약 2번 index에 저장된 값이 변한다면????



Update

~~(From. 안상욱)~~

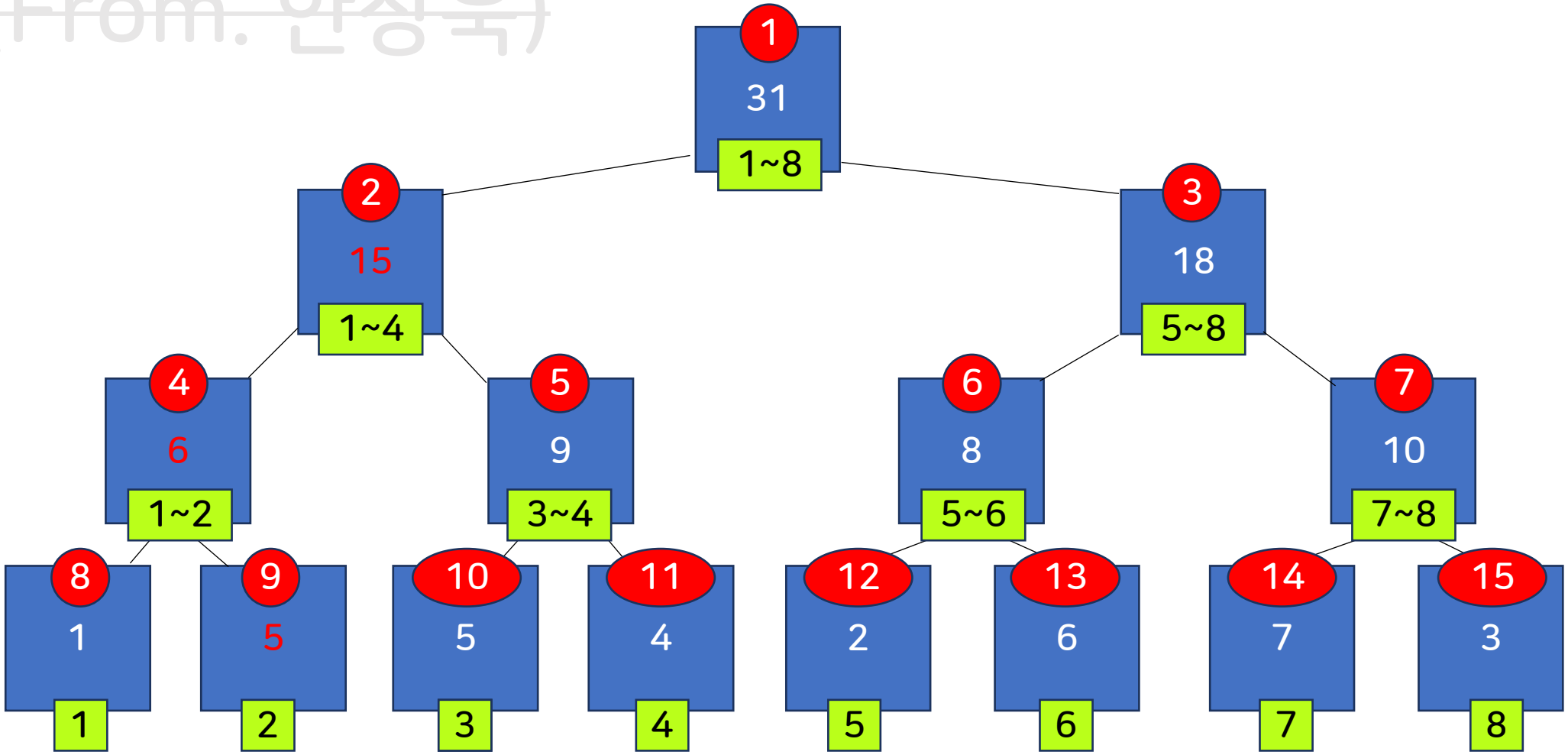
만약 2번 index에 저장된 값이 변한다면????



Update

~~(From. 안상욱)~~

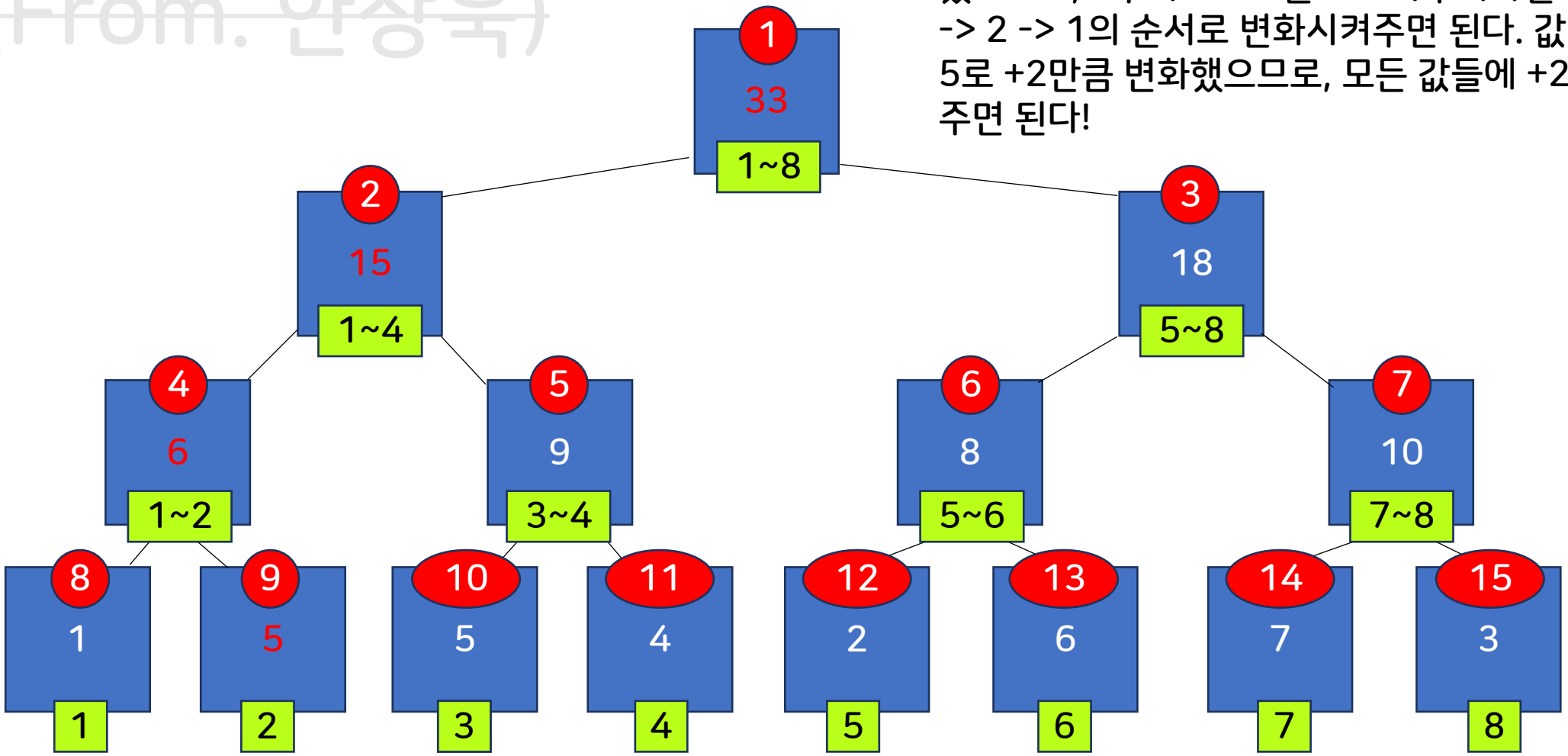
만약 2번 index에 저장된 값이 변한다면????



Update

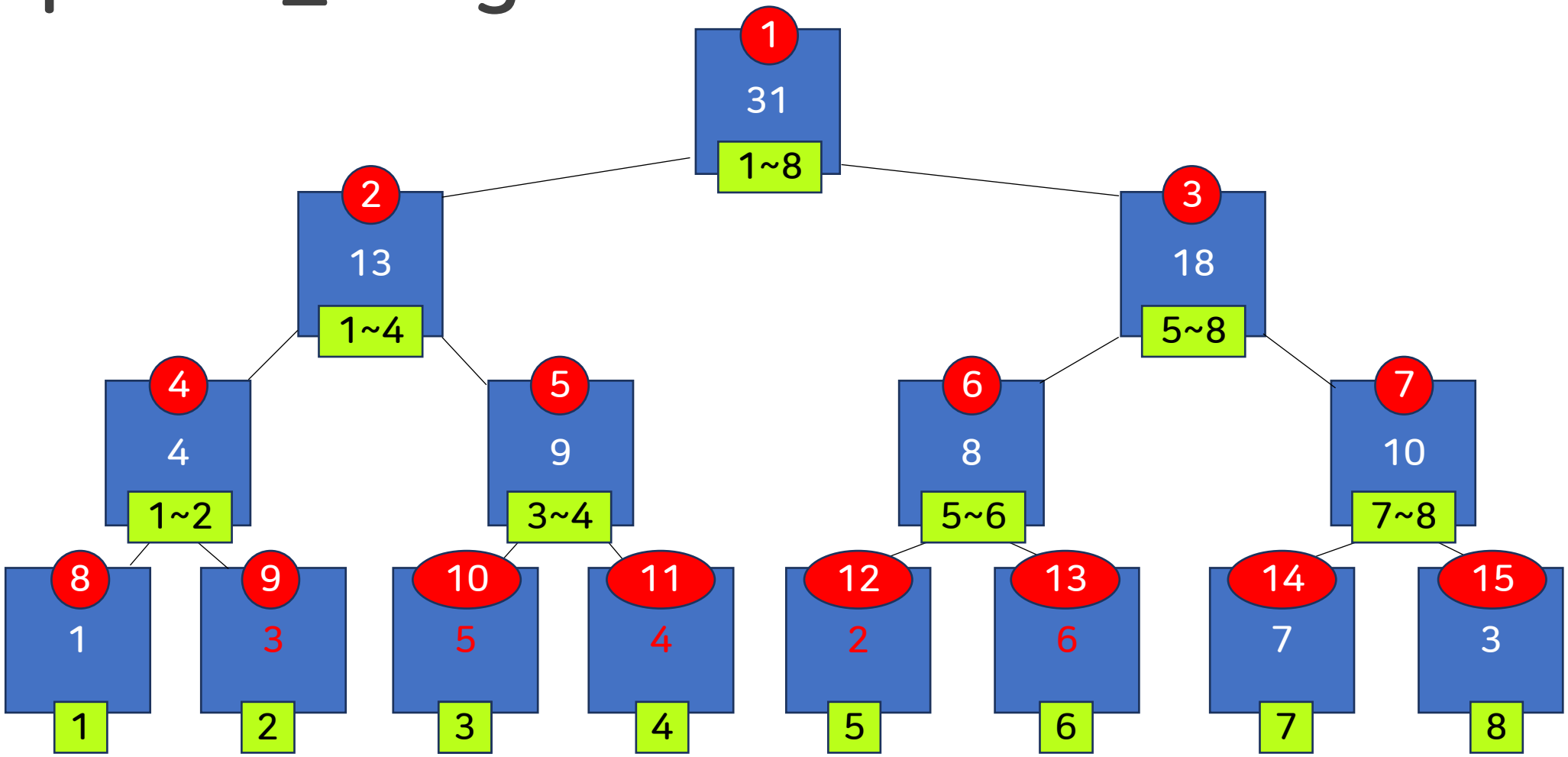
~~(From. 안상욱)~~

부모 node의 값들만 변화시키면 된다! 즉, 2번째 원소는 segment tree 배열의 9번째 index에 담겨 있으므로, 9부터 index를 2로 나누어가면서 9 -> 4 -> 2 -> 1의 순서로 변화시켜주면 된다. 값이 3에서 5로 +2만큼 변화했으므로, 모든 값들에 +2씩을 해주면 된다!



Update_range

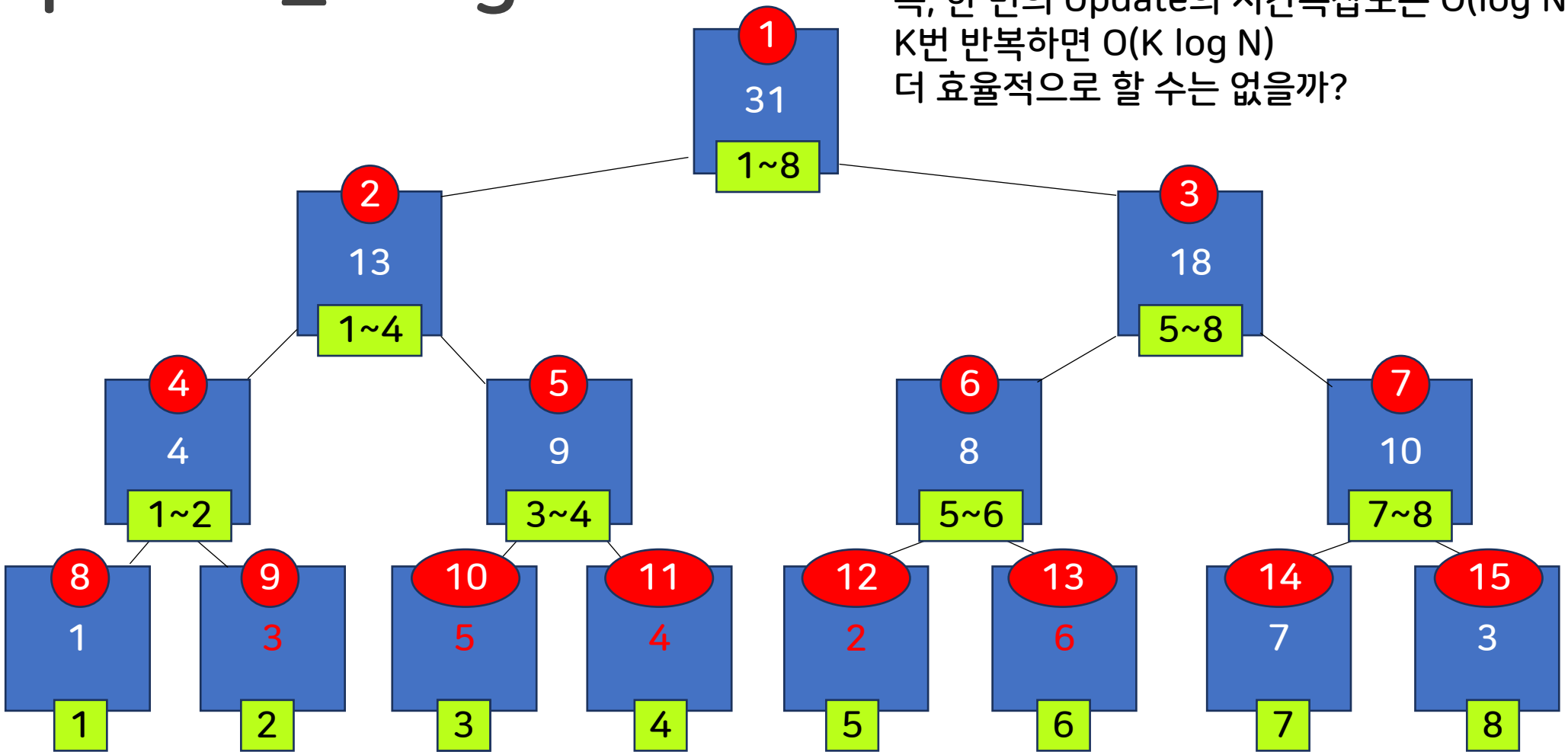
만약 구간 2번~6번 index에 저장된 값이 변한다면?
확장) 구간의 길이가 k에 해당하는 index들에 저장된 값이 변한다면?



Update_range

단순하게 생각하면 Update를 K번 실행하면 된다! ~~(될 리가 없죠?)~~
N개의 Seg Tree의 높이는 (log N).

즉, 한 번의 Update의 시간복잡도는 O(log N)
K번 반복하면 O(K log N)
더 효율적으로 할 수는 없을까?



Lazy Propagation 느리게 전파하자 ~~(직역)~~

Lazy Propagation은 이름에서 말하듯 전파를 늦추는 방법입니다.

Seg Tree의 Update에서 수정된 내용을 연관된 모든 노드에 전달하지 않기 때문에 O (log N)보다 더 효율적인 시간복잡도를 갖습니다.

그래서 길이가 K인 구간에 대한 Update를 해도 O(K log N)보다 효율적이게 됩니다.

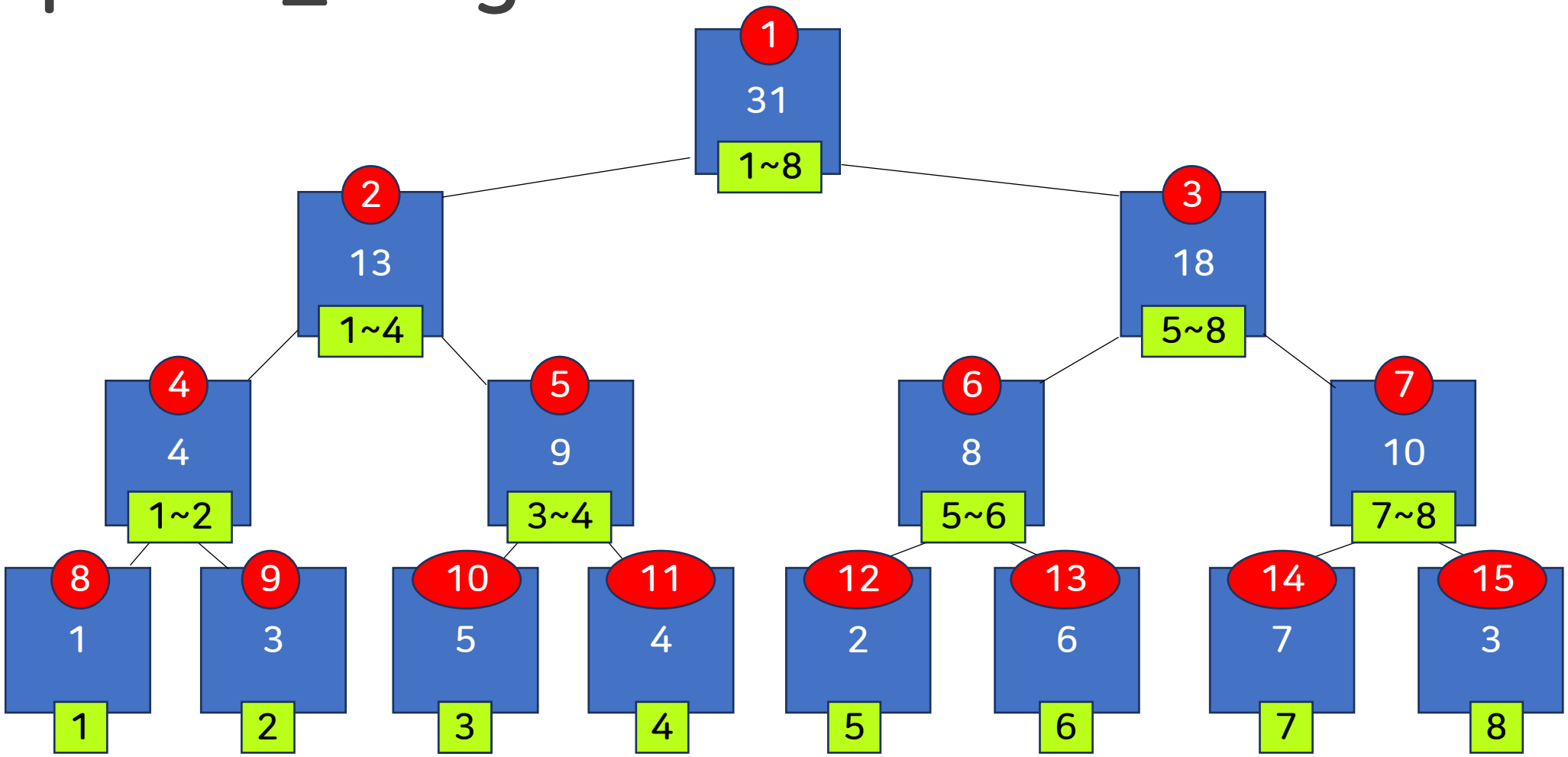
Lazy Propagation 느리게 전파하자 ~~(직역)~~

구현 방법

1. 먼저 구간을 대표하는 노드를 찾는다.
2. 가던 도중에 Lazy가 있는 노드를 만난다면,
    1) Lazy를 그 노드에 반영한다.
    2) 자식노드들에게 Lazy를 전파시켜 준다.
    3) 동일한 Lazy가 다시 나중에 전파되지 않도록 Lazy를 0으로 수정한다.
3. 위의 과정을 반복한다.
4. 마지막에 자식노드의 변경값을 반영한다.

2번~6번 index에 저장된 값에 5를 더하는 연산을 해봅시다!

Update_range



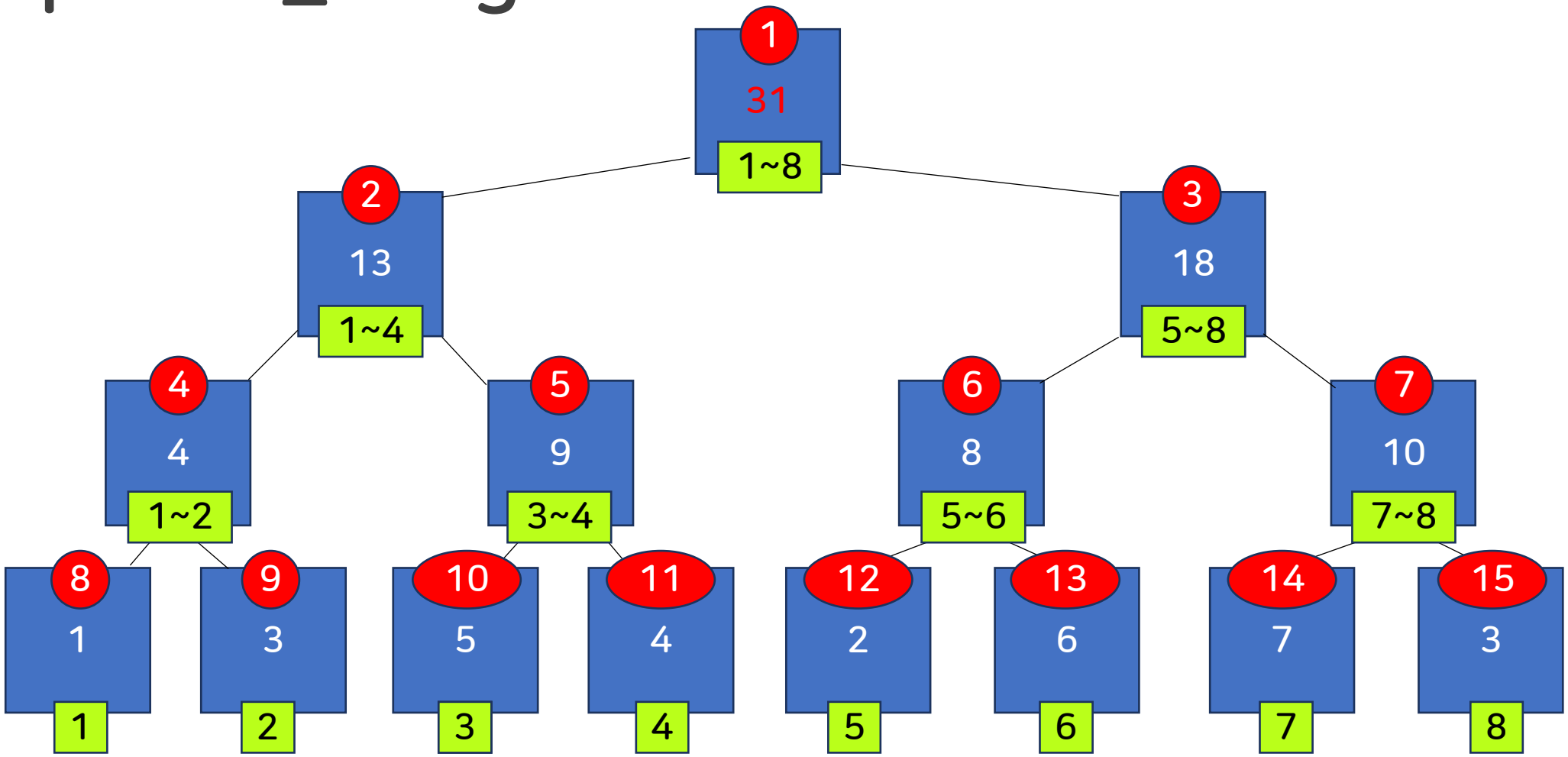
Lazy Propagation 느리게 전파하자 ~~(직역)~~

구현 방법

1. 먼저 구간을 대표하는 노드를 찾는다.
2. 가던 도중에 Lazy가 있는 노드를 만난다면,
    1) Lazy를 그 노드에 반영한다.
    2) 자식노드들에게 Lazy를 전파시켜 준다.
    3) 동일한 Lazy가 다시 나중에 전파되지 않도록 Lazy를 0으로 수정한다.
3. 위의 과정을 반복한다.
4. 마지막에 자식노드의 변경값을 반영한다.

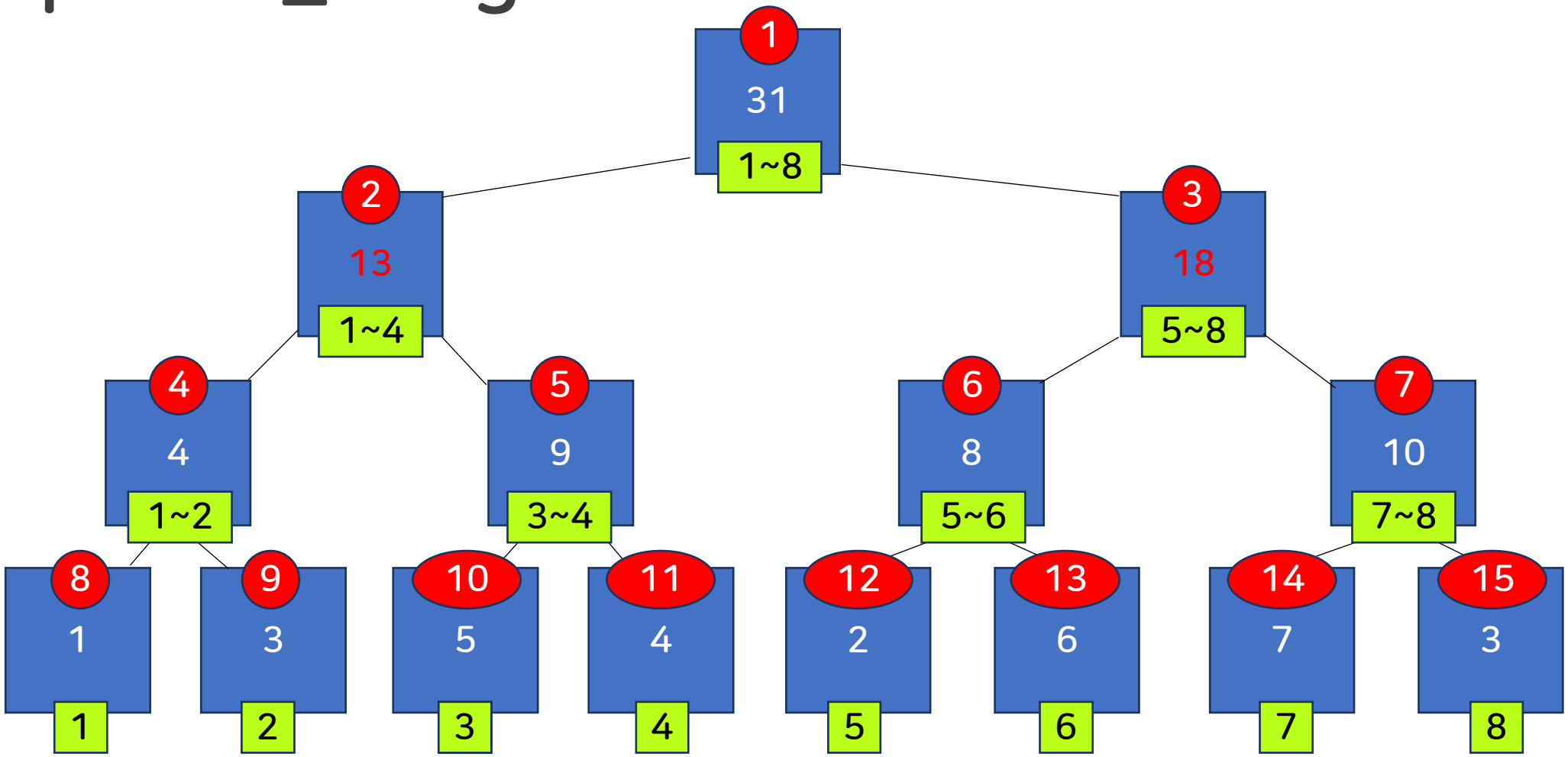
Update_range

2번~6번 index를 대표하는 노드를 찾아가 볼까요?
노드에 적힌 수의 색 변화에 집중해주세요!



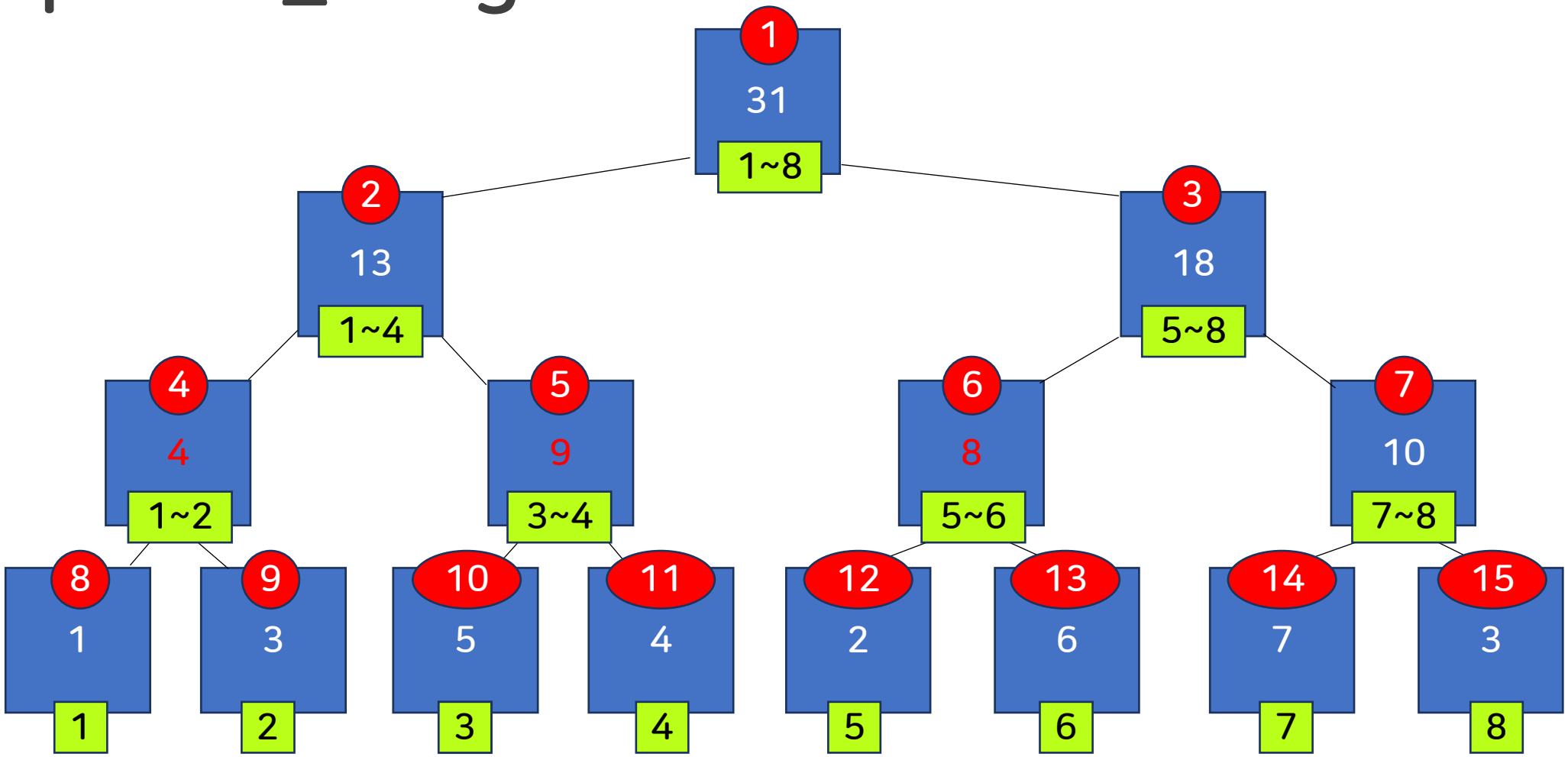
Update_range

2번~6번 index를 대표하는 노드를 찾아가 볼까요?
노드에 적힌 수의 색 변화에 집중해주세요!



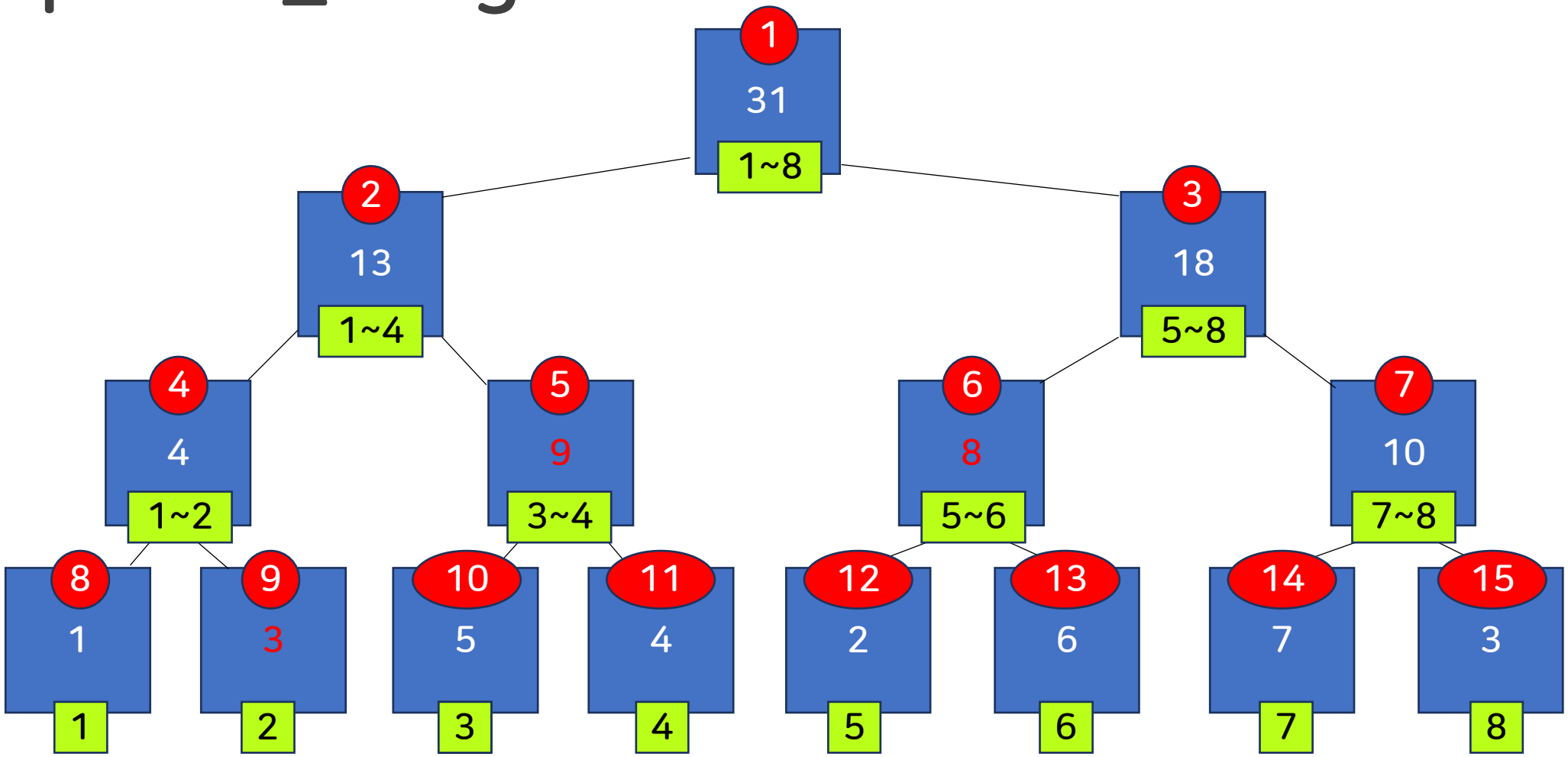
Update_range

2번~6번 index를 대표하는 노드를 찾아가 볼까요?
노드에 적힌 수의 색 변화에 집중해주세요!



Update_range

2번~6번 index를 대표하는 노드를 찾아가 볼까요?
노드에 적힌 수의 색 변화에 집중해주세요!



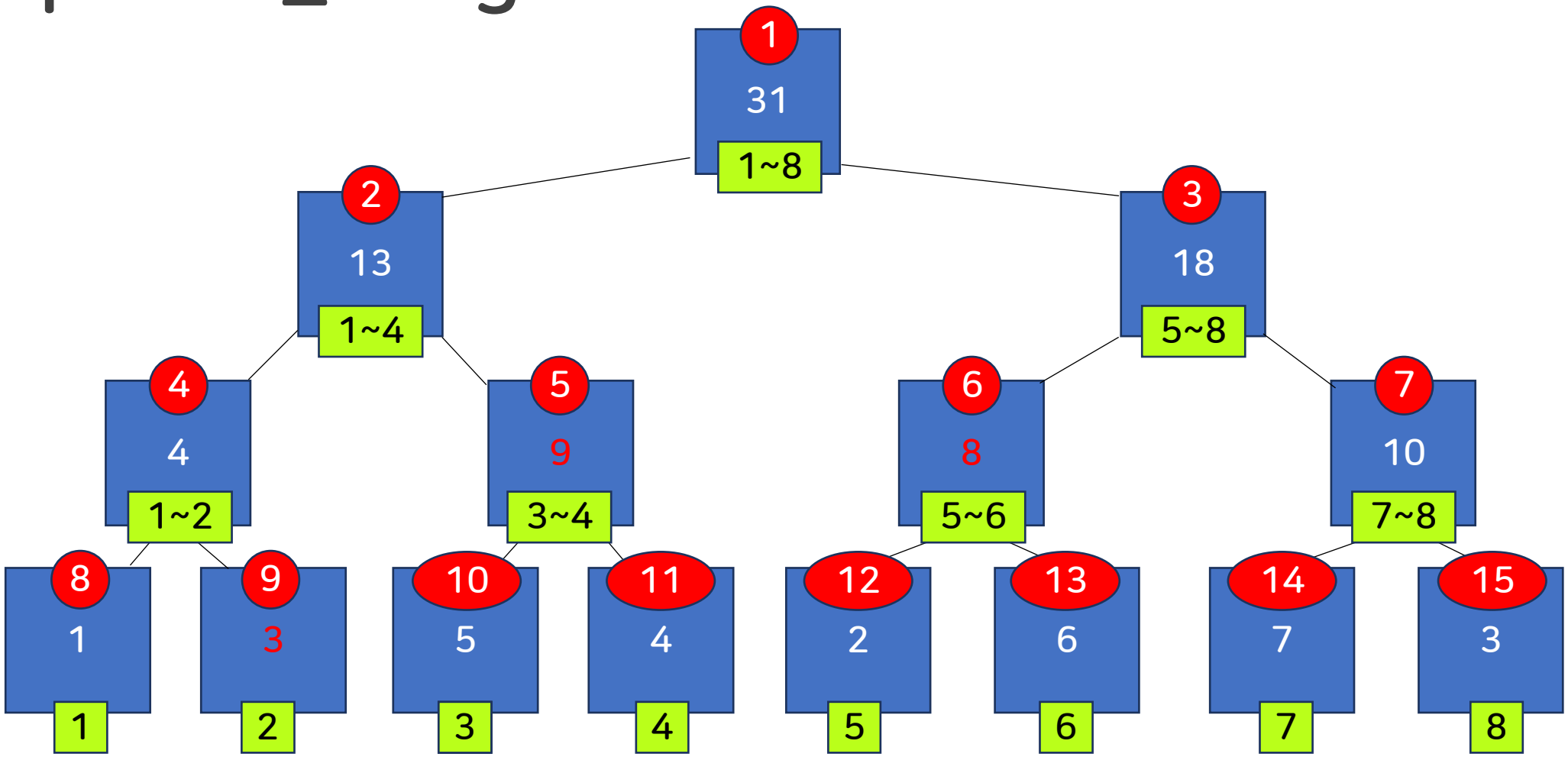
Lazy Propagation 느리게 전파하자 ~~(직역)~~

구현 방법

1. 먼저 구간을 대표하는 노드를 찾는다.
2. 가던 도중에 Lazy가 있는 노드를 만난다면,
    1) Lazy를 그 노드에 반영한다.
    2) 자식노드들에게 Lazy를 전파시켜 준다.
    3) 동일한 Lazy가 다시 나중에 전파되지 않도록 Lazy를 0으로 수정한다.
3. 위의 과정을 반복한다.
4. 마지막에 자식노드의 변경값을 반영한다.

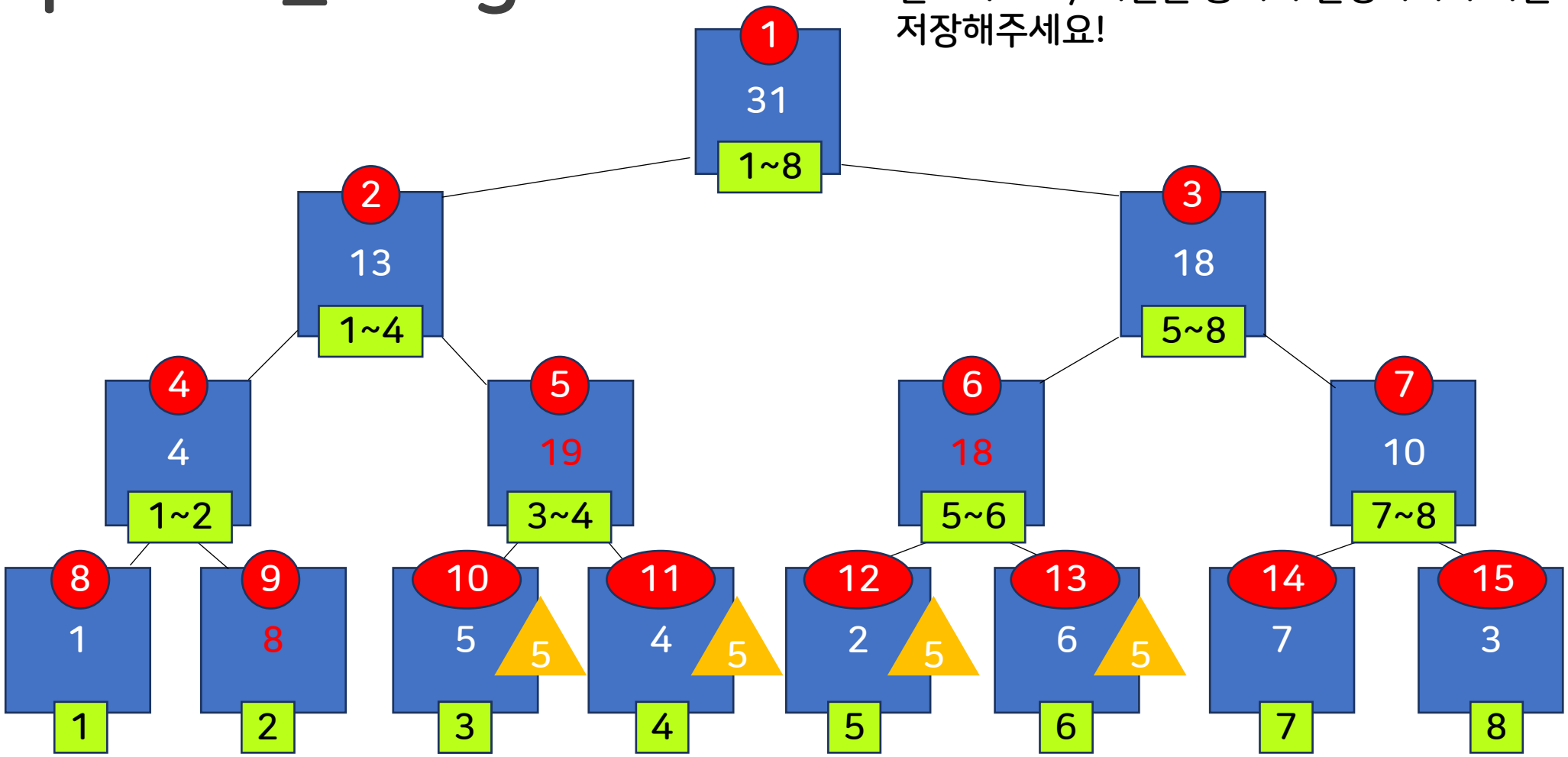
지금은 Lazy가 없으니 Lazy를 저장해 봅시다!

Update_range



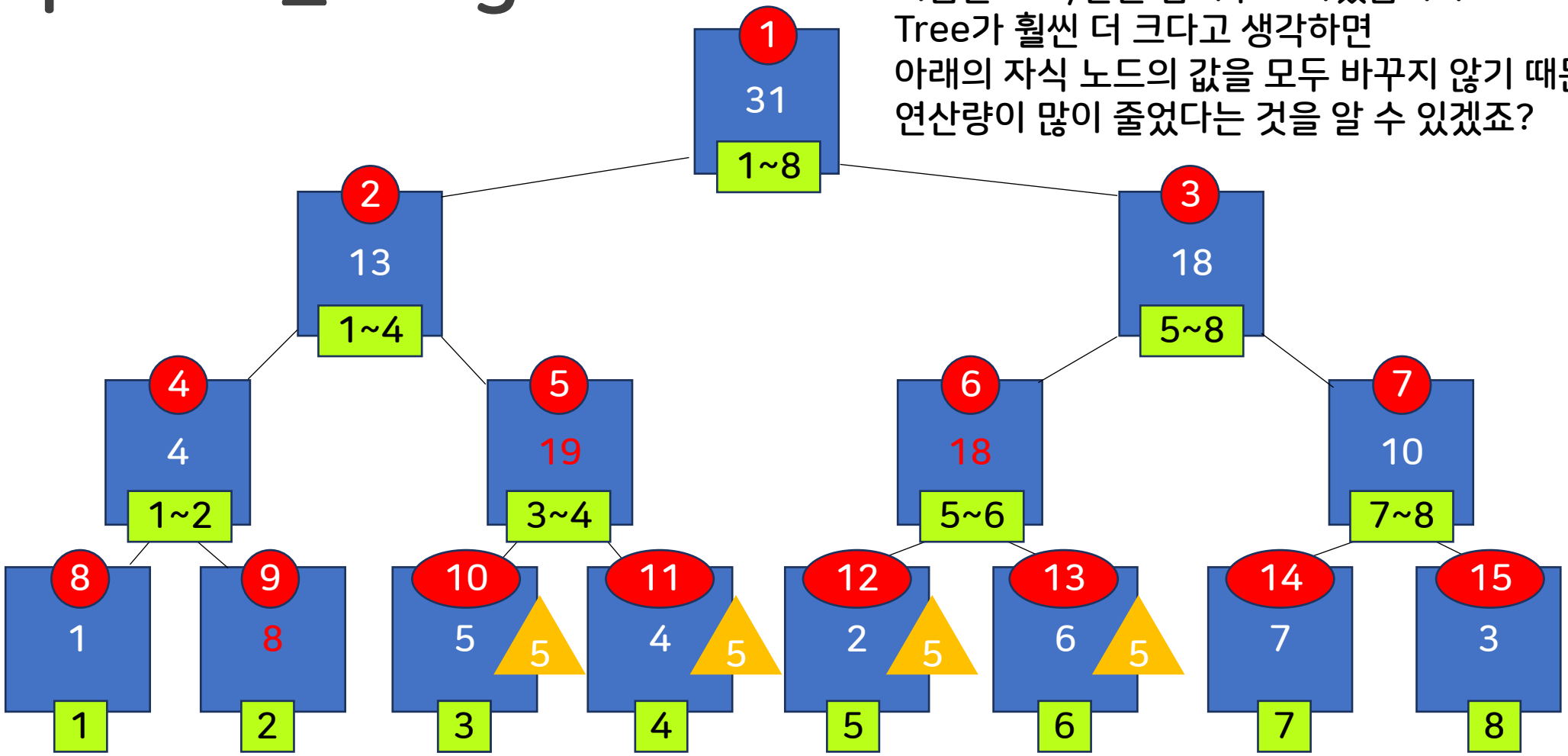
Update_range

우선 값을 변경해주세요!
그리고 자식 노드가 있는 경우에는
별도의 Lazy 배열을 통해서 변경시켜야 하는 값을
저장해주세요!



Update_range

Lazy 배열로 저장하지 않았다고 생각하면 10 ~ 13번
의 노드에 있는 값을 모두 변경했을 텐데
지금은 Lazy만을 남겨두고 마쳤습니다.
Tree가 훨씬 더 크다고 생각하면
아래의 자식 노드의 값을 모두 바꾸지 않기 때문에
연산량이 많이 줄었다는 것을 알 수 있겠죠?



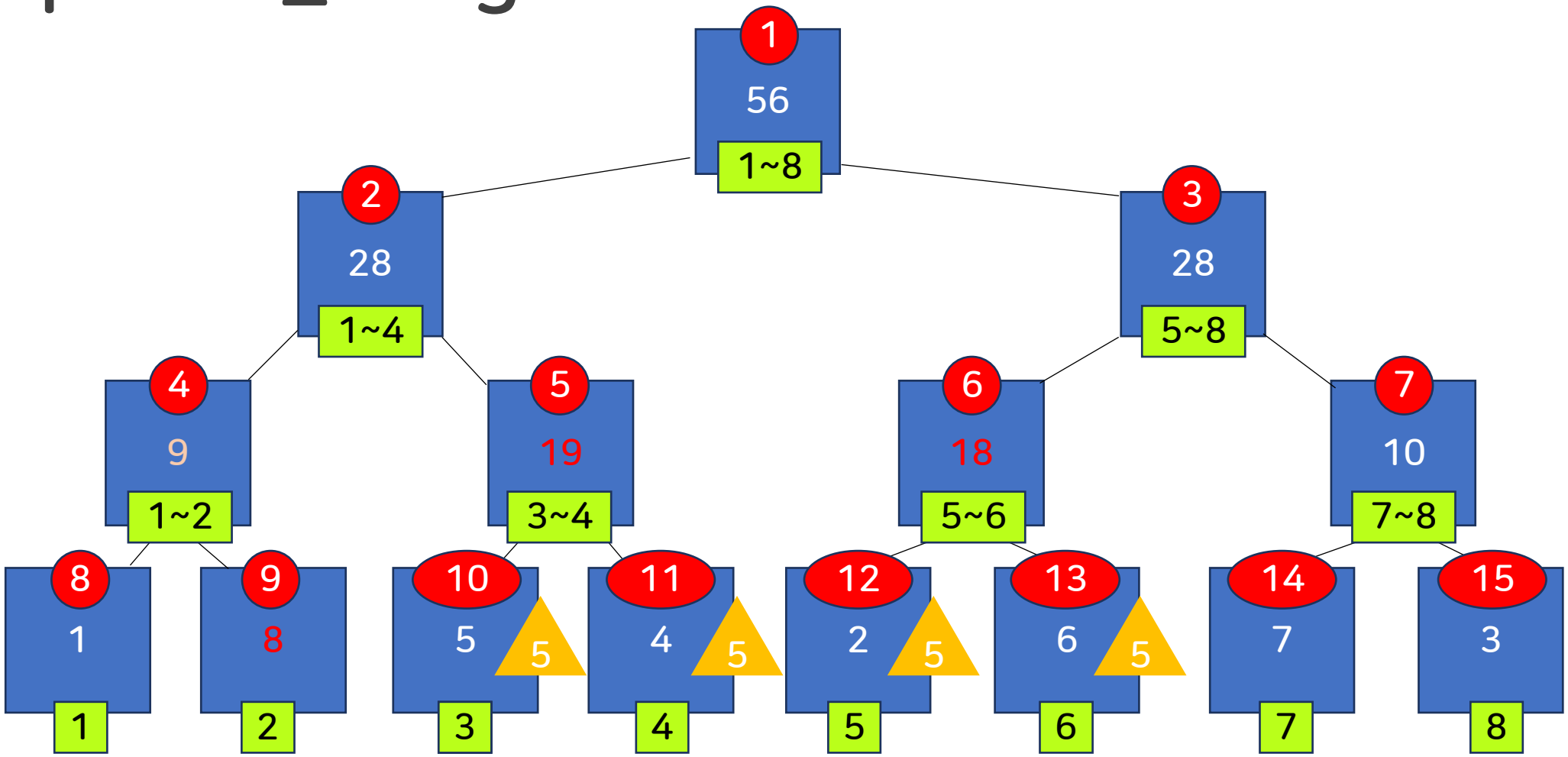
Lazy Propagation 느리게 전파하자 ~~(직역)~~

구현 방법

1. 먼저 구간을 대표하는 노드를 찾는다.
2. 가던 도중에 Lazy가 있는 노드를 만난다면,
    1) Lazy를 그 노드에 반영한다.
    2) 자식노드들에게 Lazy를 전파시켜 준다.
    3) 동일한 Lazy가 다시 나중에 전파되지 않도록 Lazy를 0으로 수정한다.
3. 위의 과정을 반복한다.
4. 마지막에 자식노드의 변경값을 반영한다.

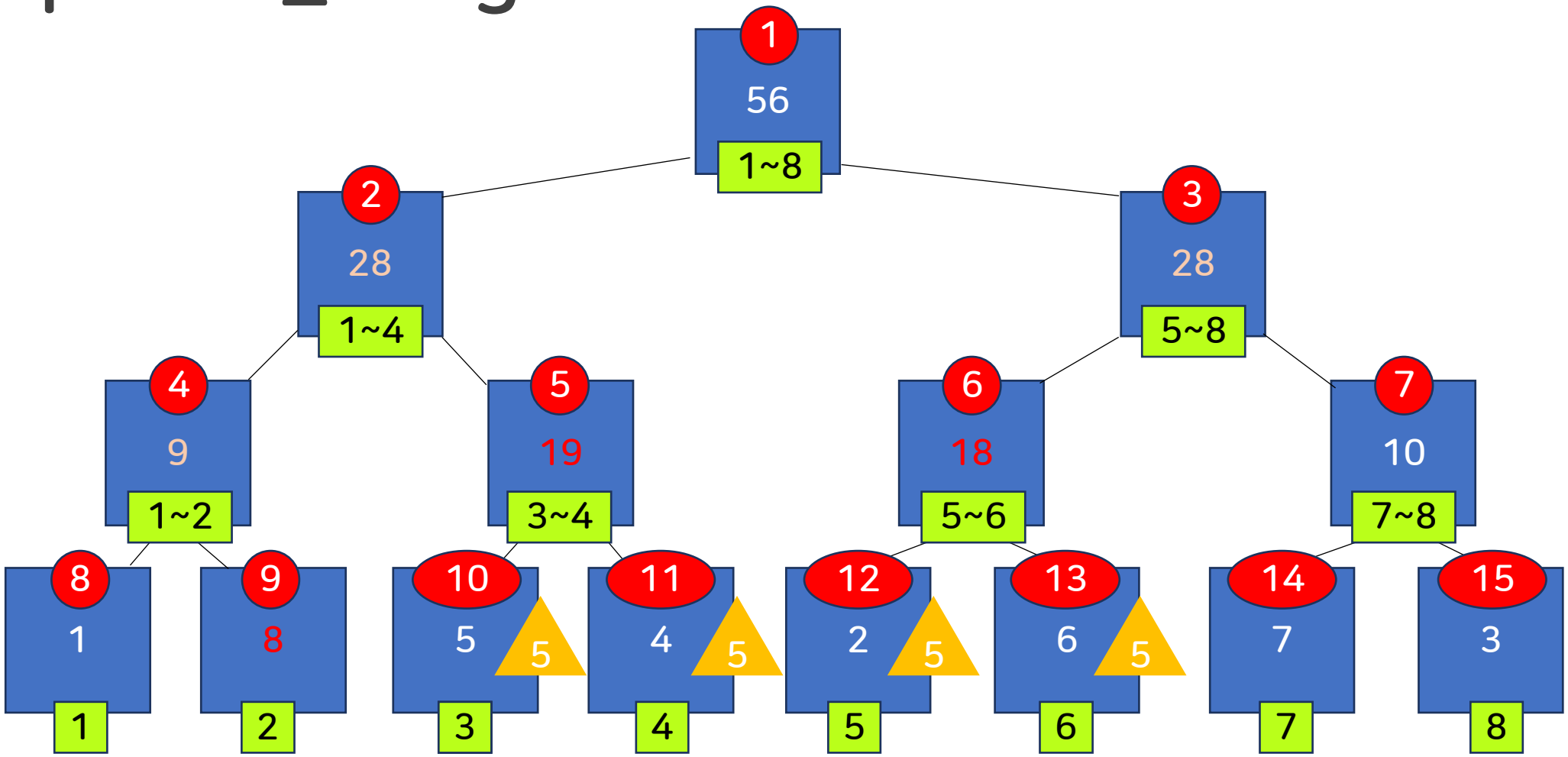
Update_range

자식 노드의 값이 변경된 것을 부모 노드에도 반영해주어야 합니다.
(연한 색으로 표시했어요)



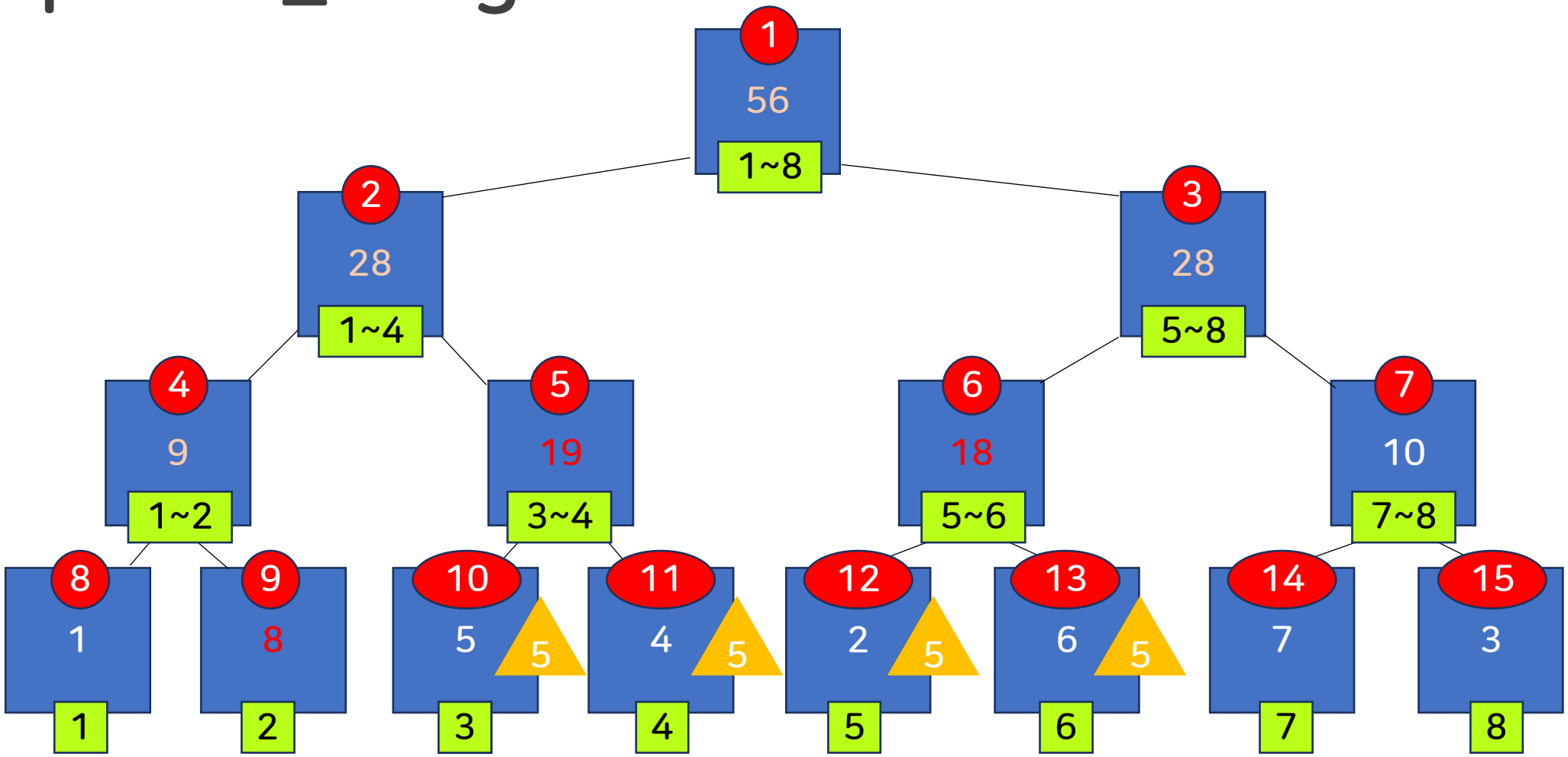
Update_range

자식 노드의 값이 변경된 것을 부모 노드에도 반영해주어야 합니다.
(연한 색으로 표시했어요)



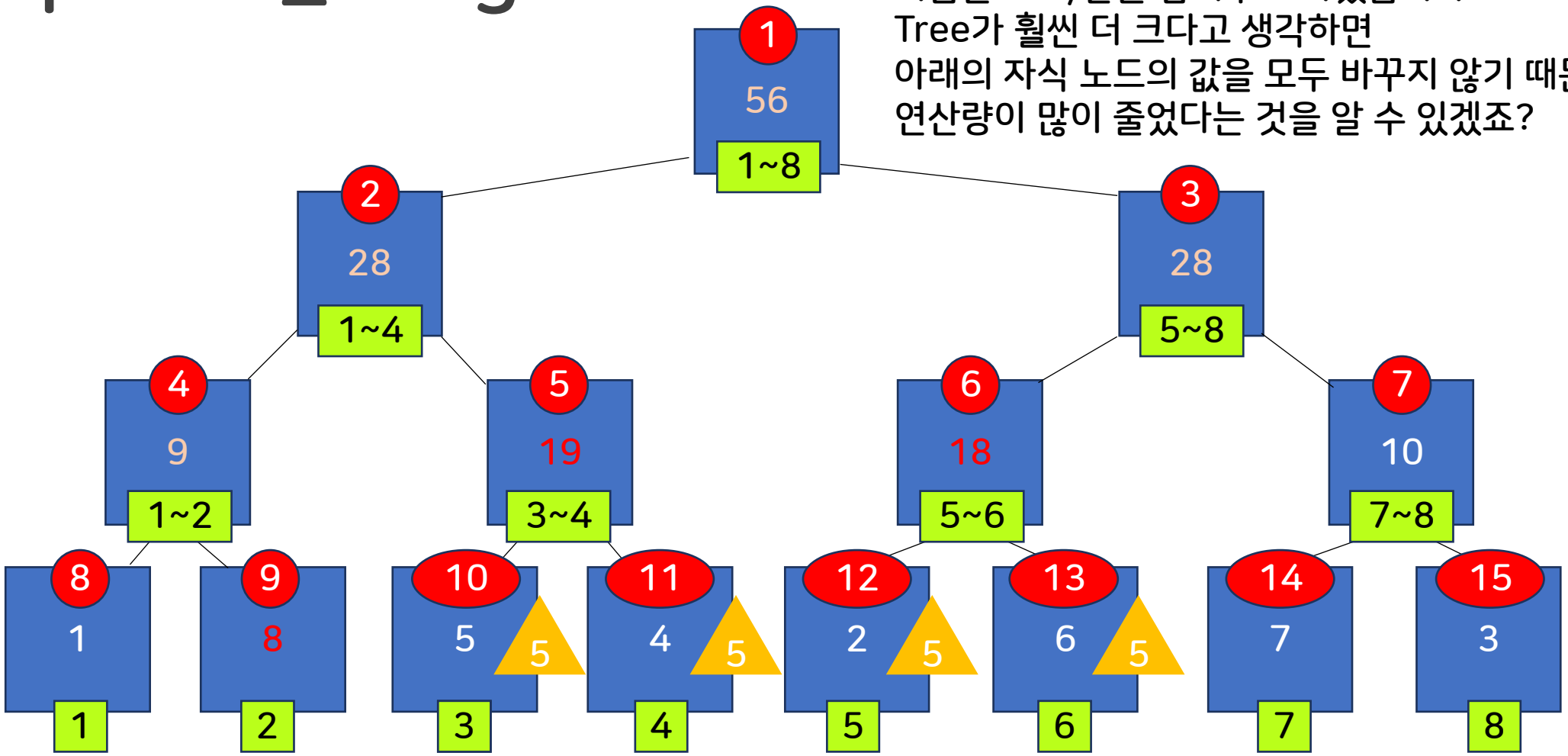
Update_range

자식 노드의 값이 변경된 것을 부모 노드에도 반영해주어야 합니다.
(연한 색으로 표시했어요)



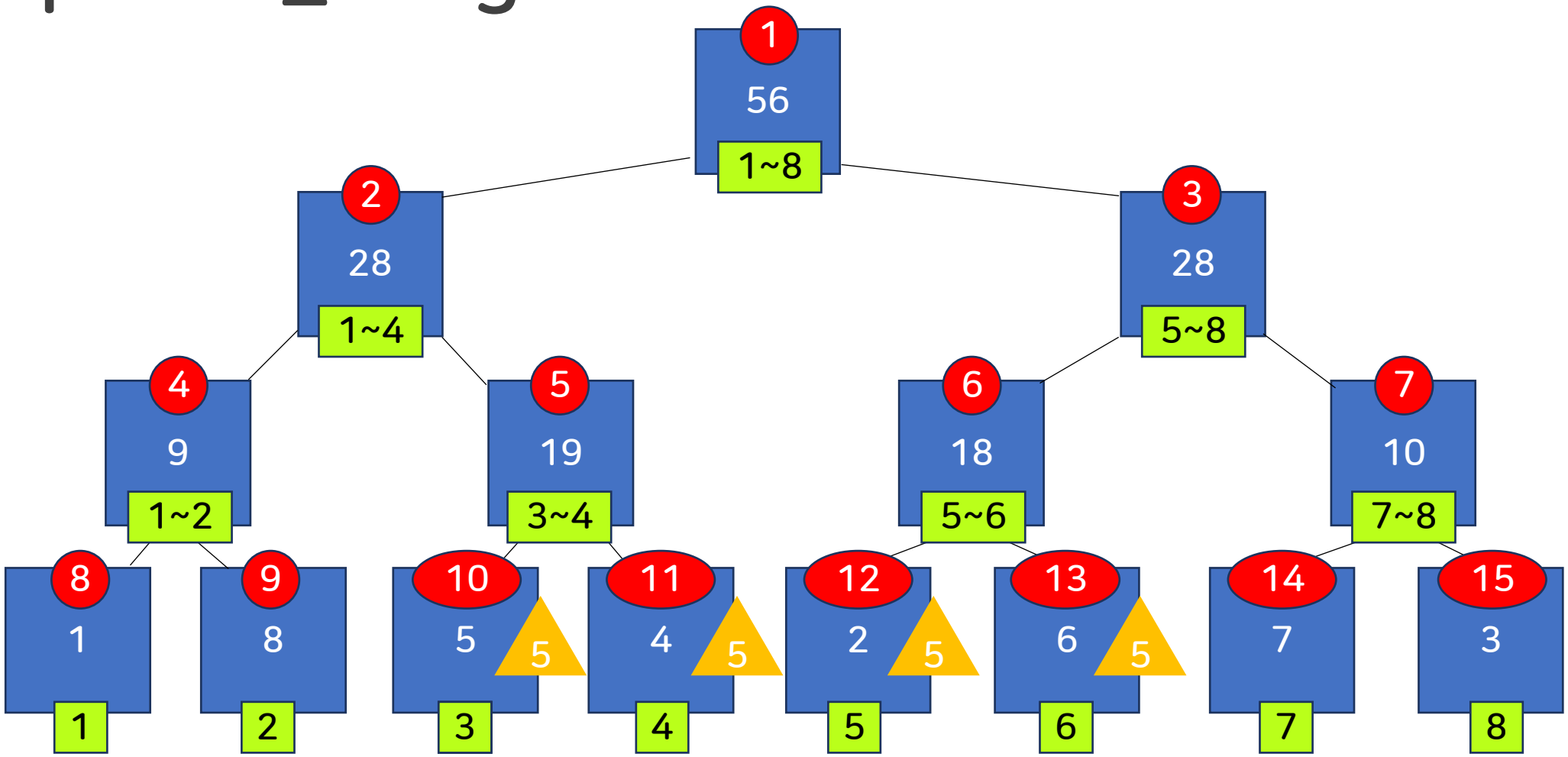
Update_range

Lazy 배열로 저장하지 않았다고 생각하면 10 ~ 13번
의 노드에 있는 값을 모두 변경했을 텐데
지금은 Lazy만을 남겨두고 마쳤습니다.
Tree가 훨씬 더 크다고 생각하면
아래의 자식 노드의 값을 모두 바꾸지 않기 때문에
연산량이 많이 줄었다는 것을 알 수 있겠죠?



이번엔 4번 ~ 7번 index의 값에 -3을 해볼게요!

Update_range



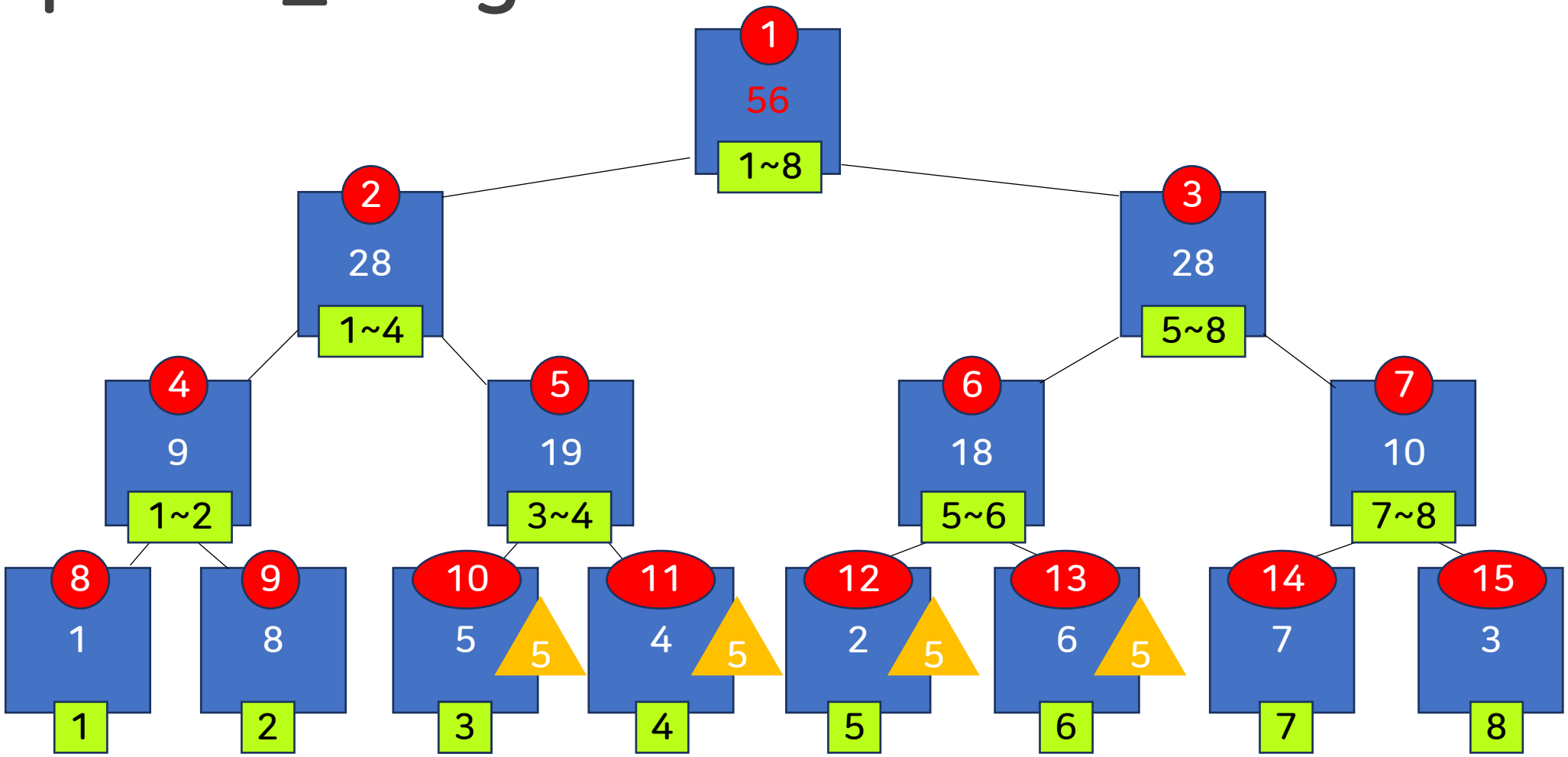
Lazy Propagation 느리게 전파하자 ~~(직역)~~

구현 방법

1. 먼저 구간을 대표하는 노드를 찾는다.
2. 가던 도중에 Lazy가 있는 노드를 만난다면,
    1) Lazy를 그 노드에 반영한다.
    2) 자식노드들에게 Lazy를 전파시켜 준다.
    3) 동일한 Lazy가 다시 나중에 전파되지 않도록 Lazy를 0으로 수정한다.
3. 위의 과정을 반복한다.
4. 마지막에 자식노드의 변경값을 반영한다.

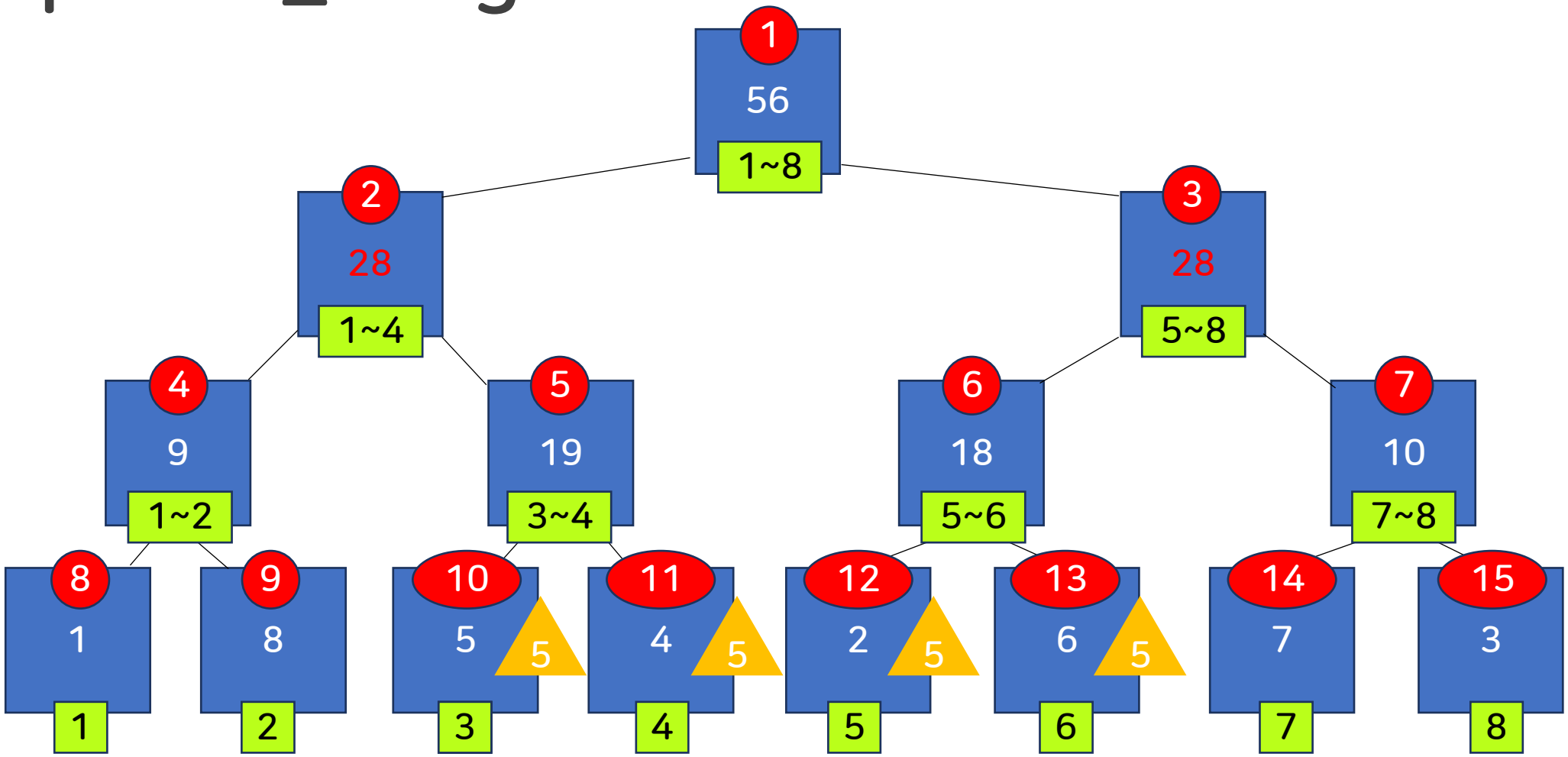
Update_range

4번~7번 index를 대표하는 노드를 찾아가 볼까요?
노드에 적힌 수의 색 변화에 집중해주세요!



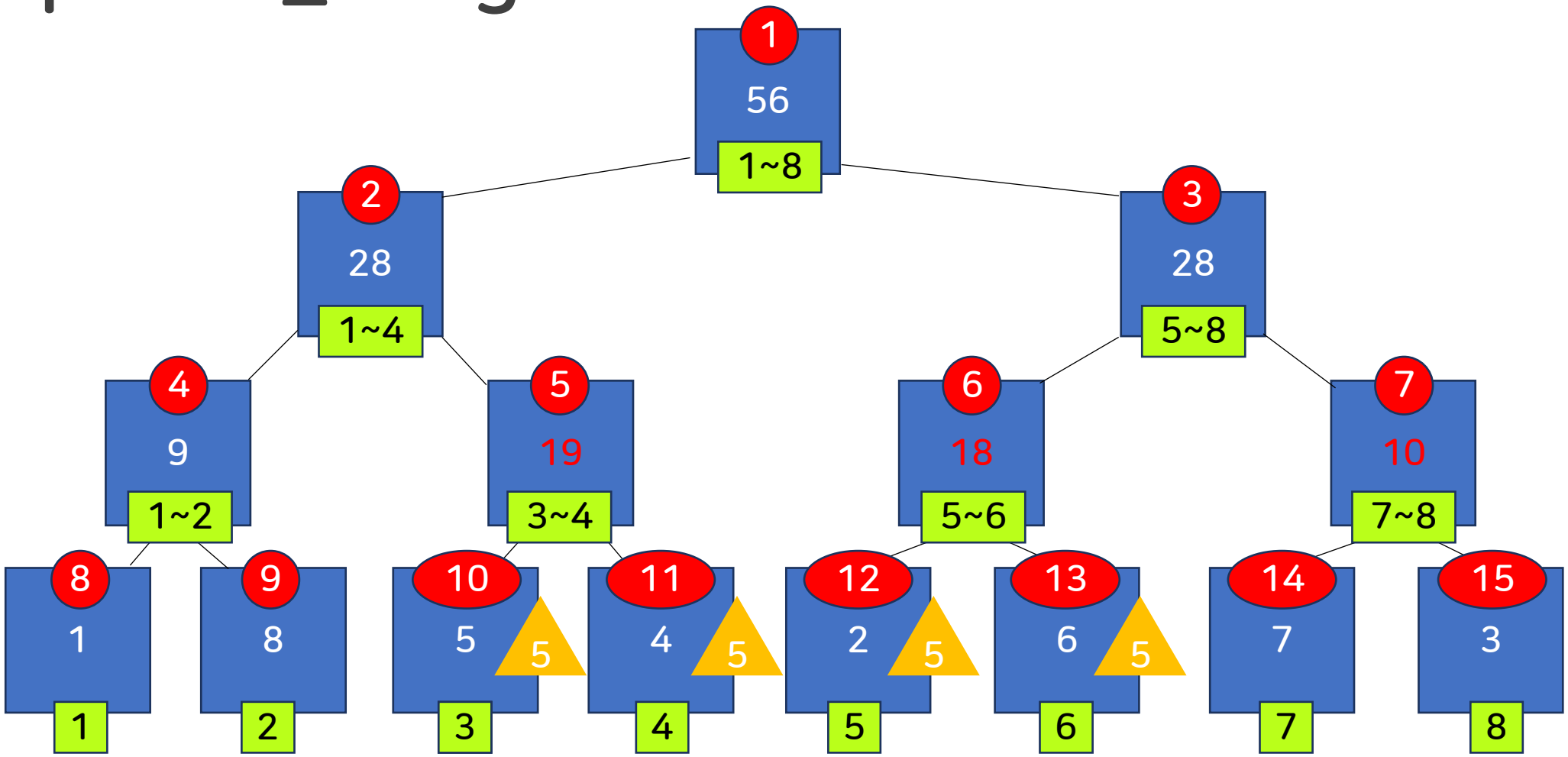
Update_range

4번~7번 index를 대표하는 노드를 찾아가 볼까요?
노드에 적힌 수의 색 변화에 집중해주세요!



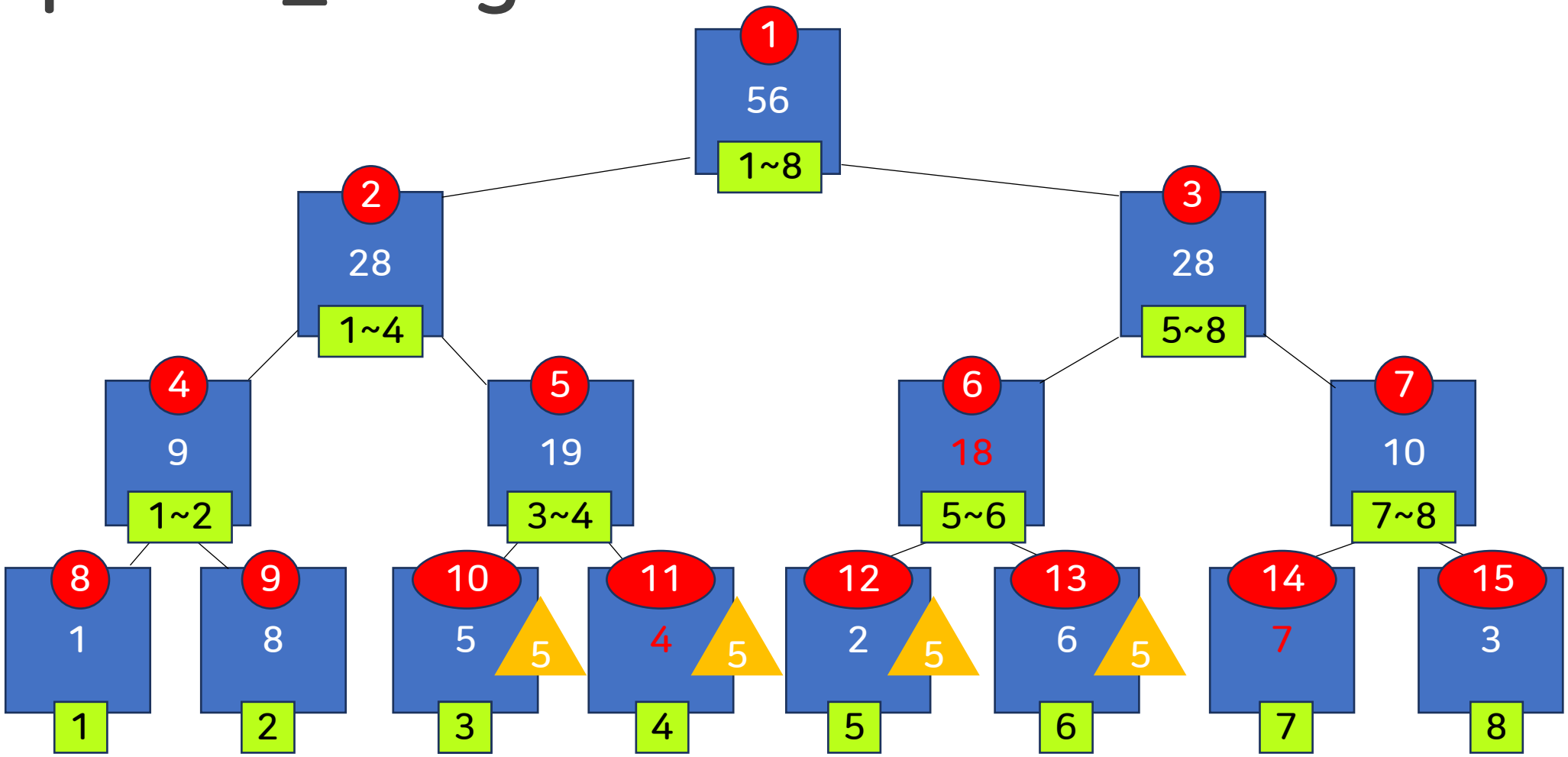
Update_range

4번~7번 index를 대표하는 노드를 찾아가 볼까요?
노드에 적힌 수의 색 변화에 집중해주세요!



Update_range

4번~7번 index를 대표하는 노드를 찾아가 볼까요?
노드에 적힌 수의 색 변화에 집중해주세요!



잠깐!

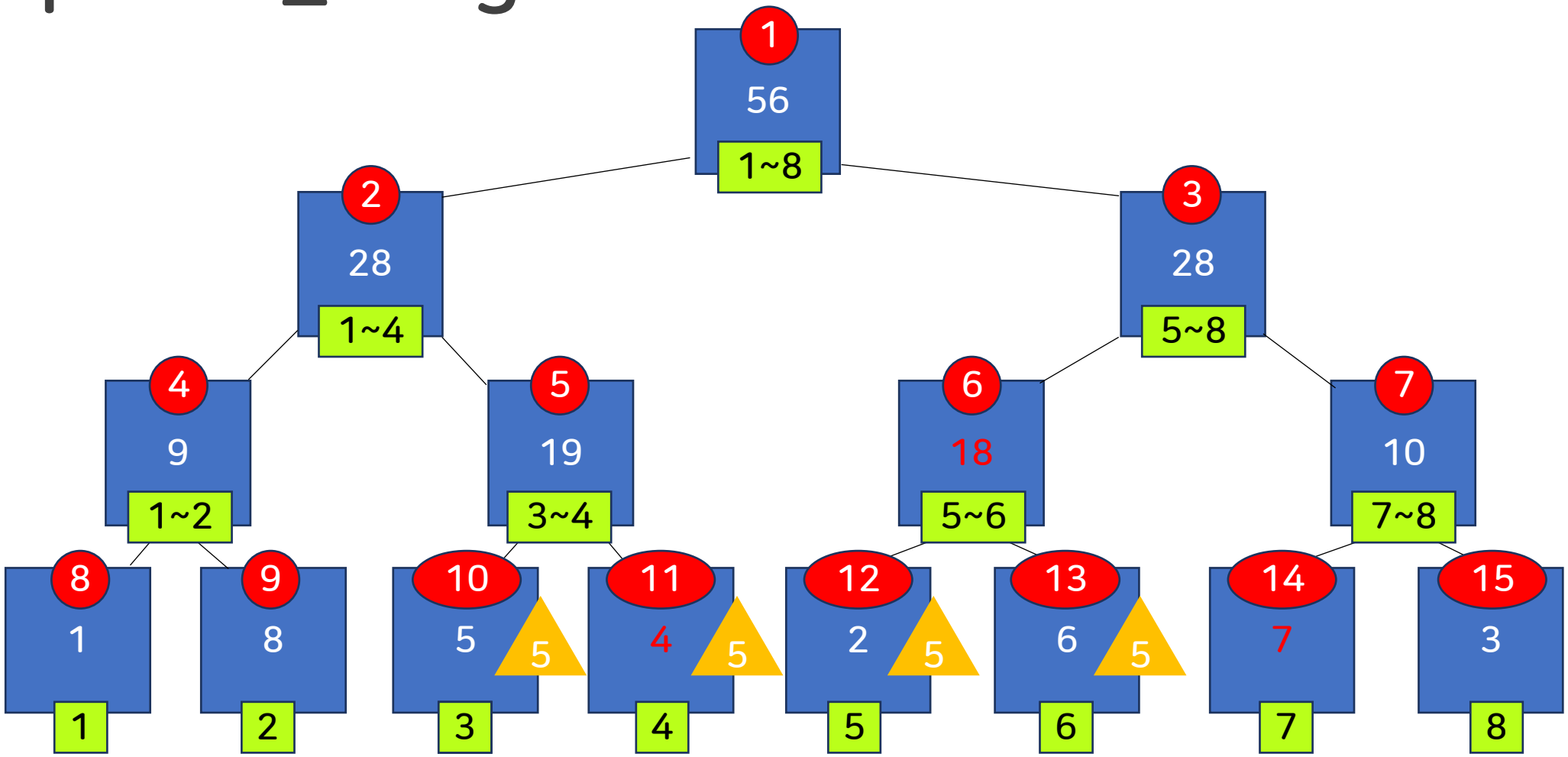
Lazy Propagation 느리게 전파하자 ~~(직역)~~

구현 방법

1. 먼저 구간을 대표하는 노드를 찾는다.
2. 가던 도중에 Lazy가 있는 노드를 만난다면,
    1) Lazy를 그 노드에 반영한다.
    2) 자식노드들에게 Lazy를 전파시켜 준다.
    3) 동일한 Lazy가 다시 나중에 전파되지 않도록 Lazy를 0으로 수정한다.
3. 위의 과정을 반복한다.
4. 마지막에 자식노드의 변경값을 반영한다.

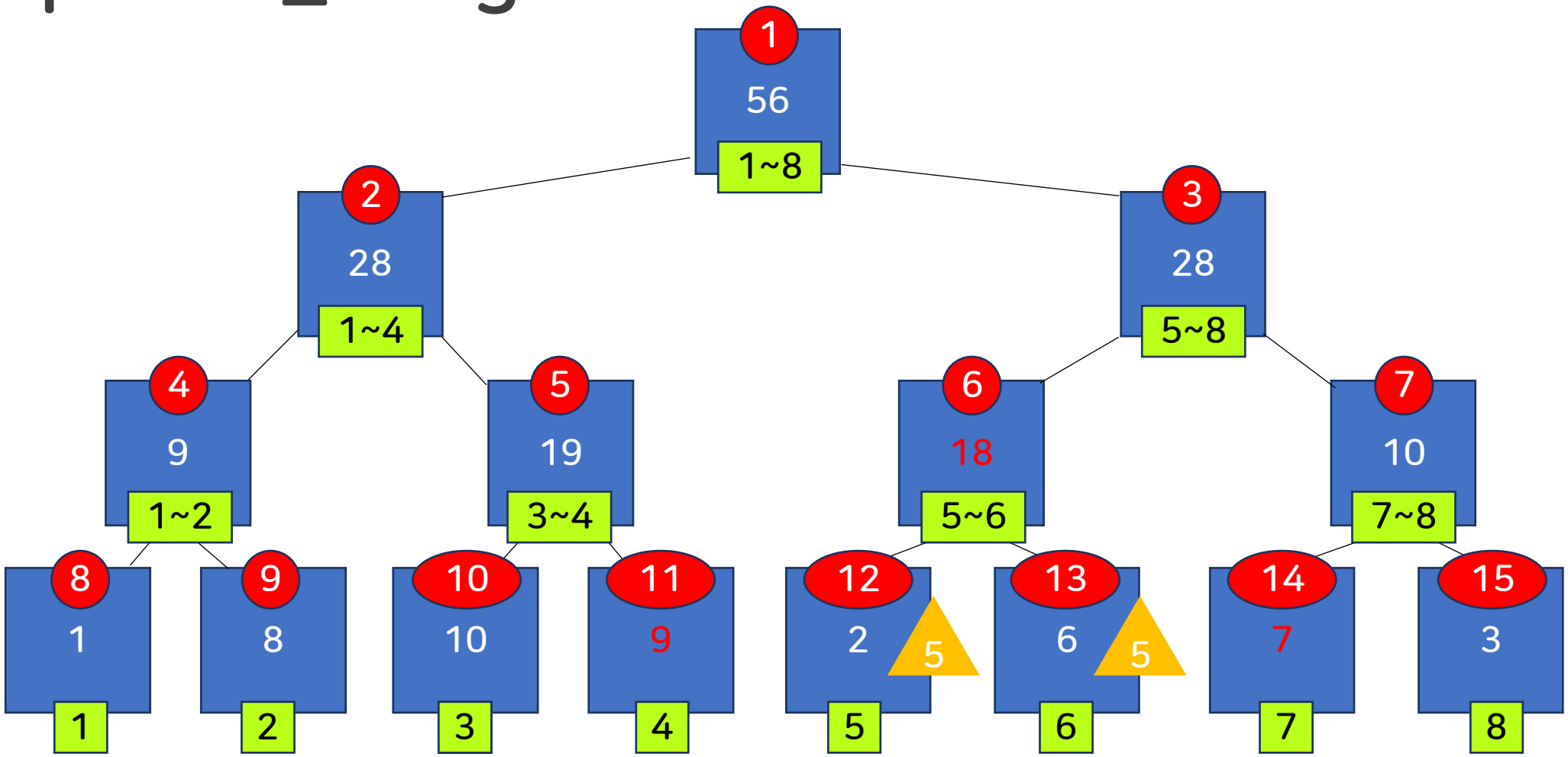
Update_range

5번 노드의 자식들(10번, 11번 노드)에 Lazy가 들어 있으니 반영해줍니다.



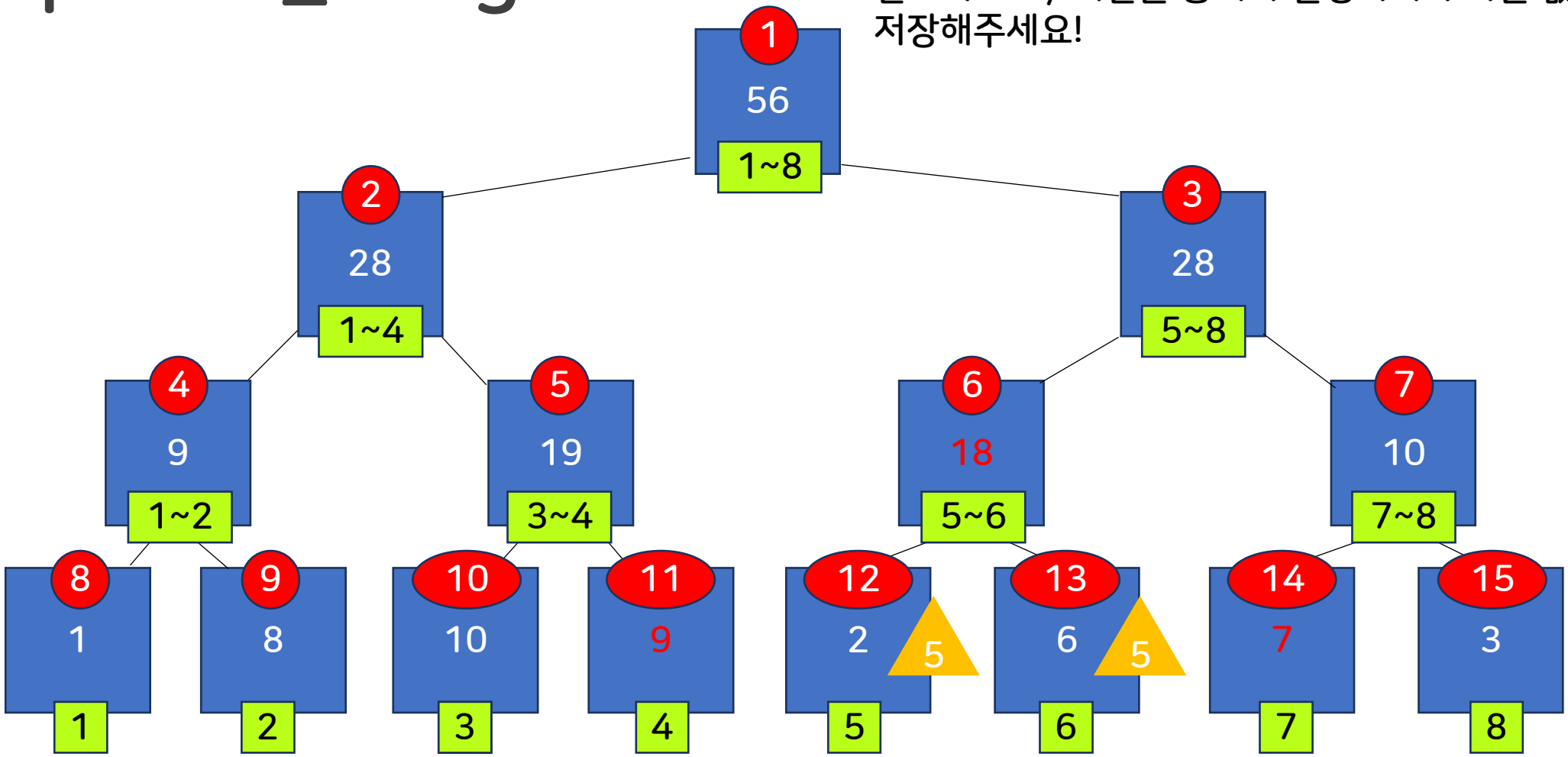
Update_range

5번 노드의 자식들(10번, 11번 노드)에 Lazy가 들어 있으니 반영해줍니다.



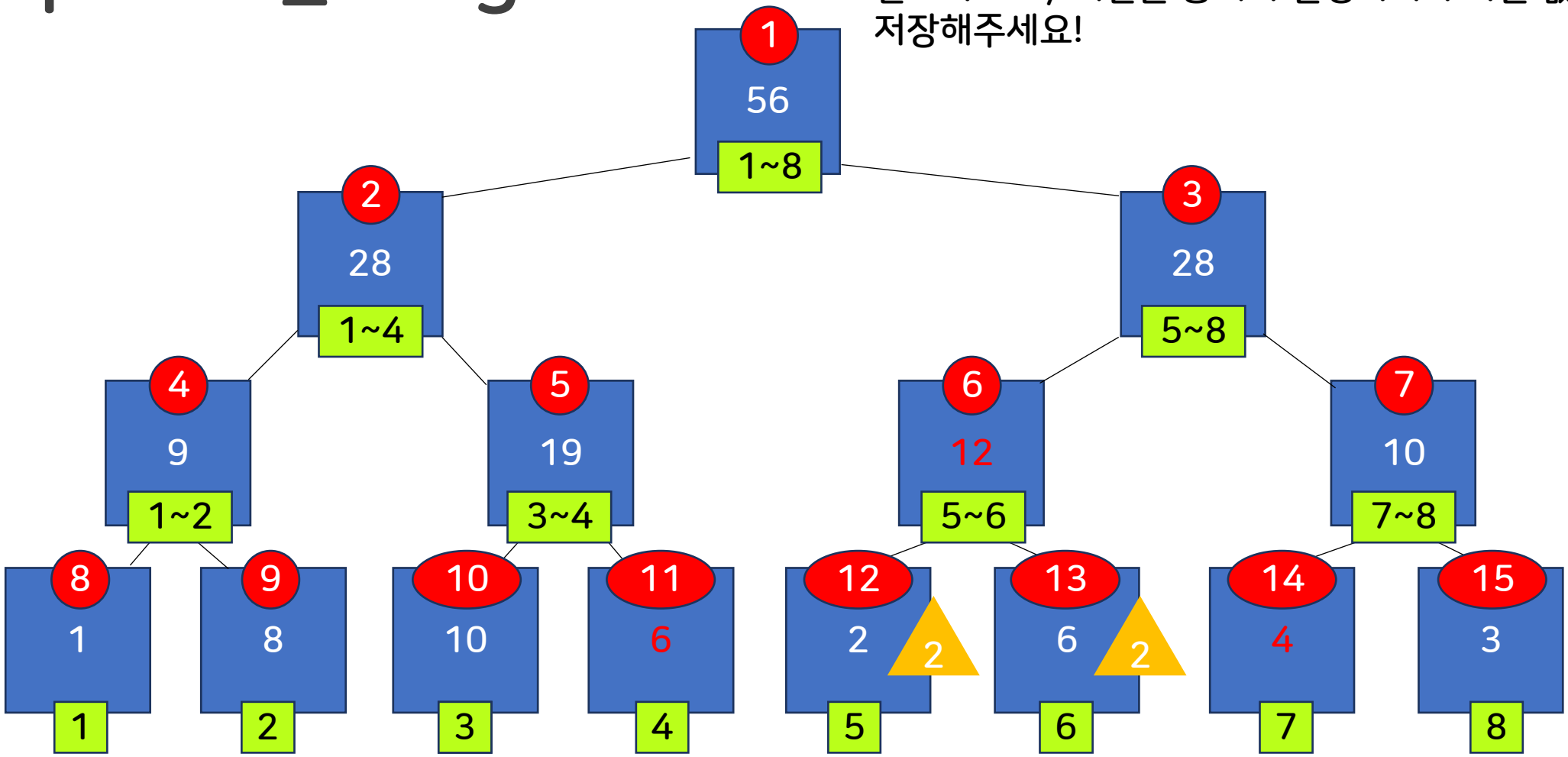
Update_range

이제 구간을 찾았으니 -3을 반영합니다.
자식 노드가 있는 경우에는
별도의 Lazy 배열을 통해서 변경시켜야 하는 값을
저장해주세요!



Update_range

이제 구간을 찾았으니 -3을 반영합니다.
자식 노드가 있는 경우에는
별도의 Lazy 배열을 통해서 변경시켜야 하는 값을
저장해주세요!



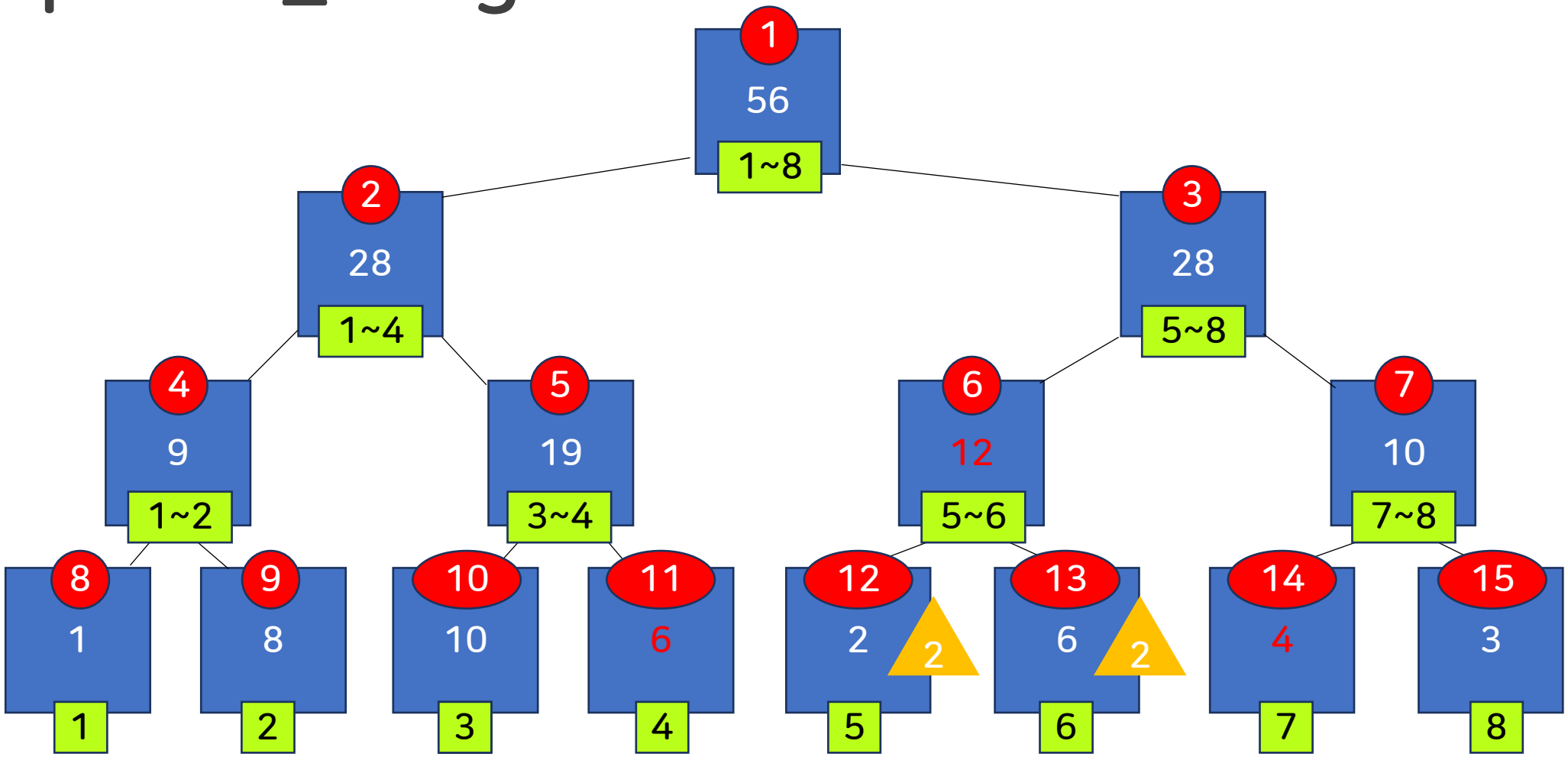
Lazy Propagation 느리게 전파하자 ~~(직역)~~

구현 방법

1. 먼저 구간을 대표하는 노드를 찾는다.
2. 가던 도중에 Lazy가 있는 노드를 만난다면,
    1) Lazy를 그 노드에 반영한다.
    2) 자식노드들에게 Lazy를 전파시켜 준다.
    3) 동일한 Lazy가 다시 나중에 전파되지 않도록 Lazy를 0으로 수정한다.
3. 위의 과정을 반복한다.
4. 마지막에 자식노드의 변경값을 반영한다.

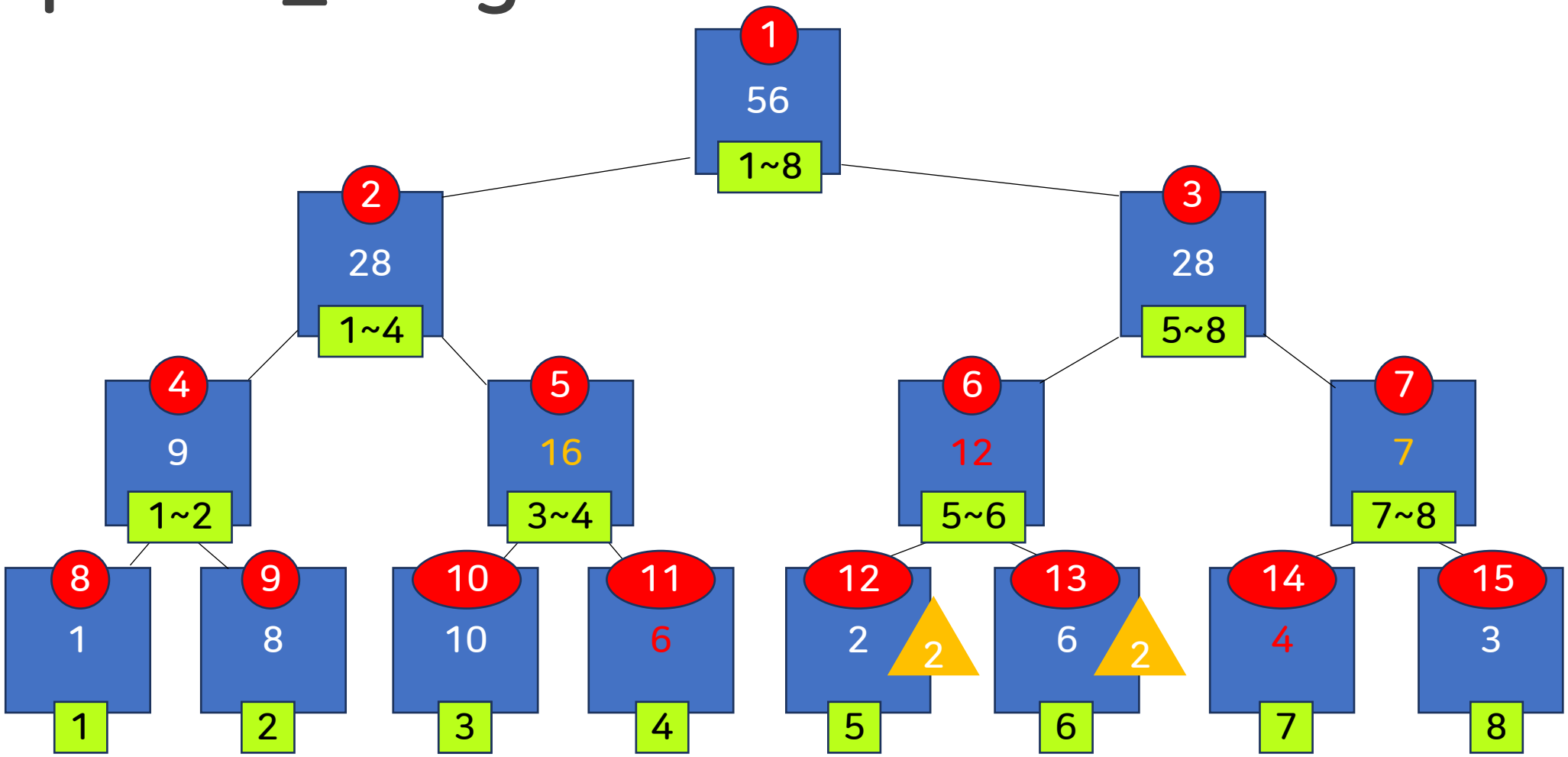
Update_range

자식 노드의 값이 변경된 것을 부모 노드에도 반영해주어야 합니다.
(연한 색으로 표시했어요)



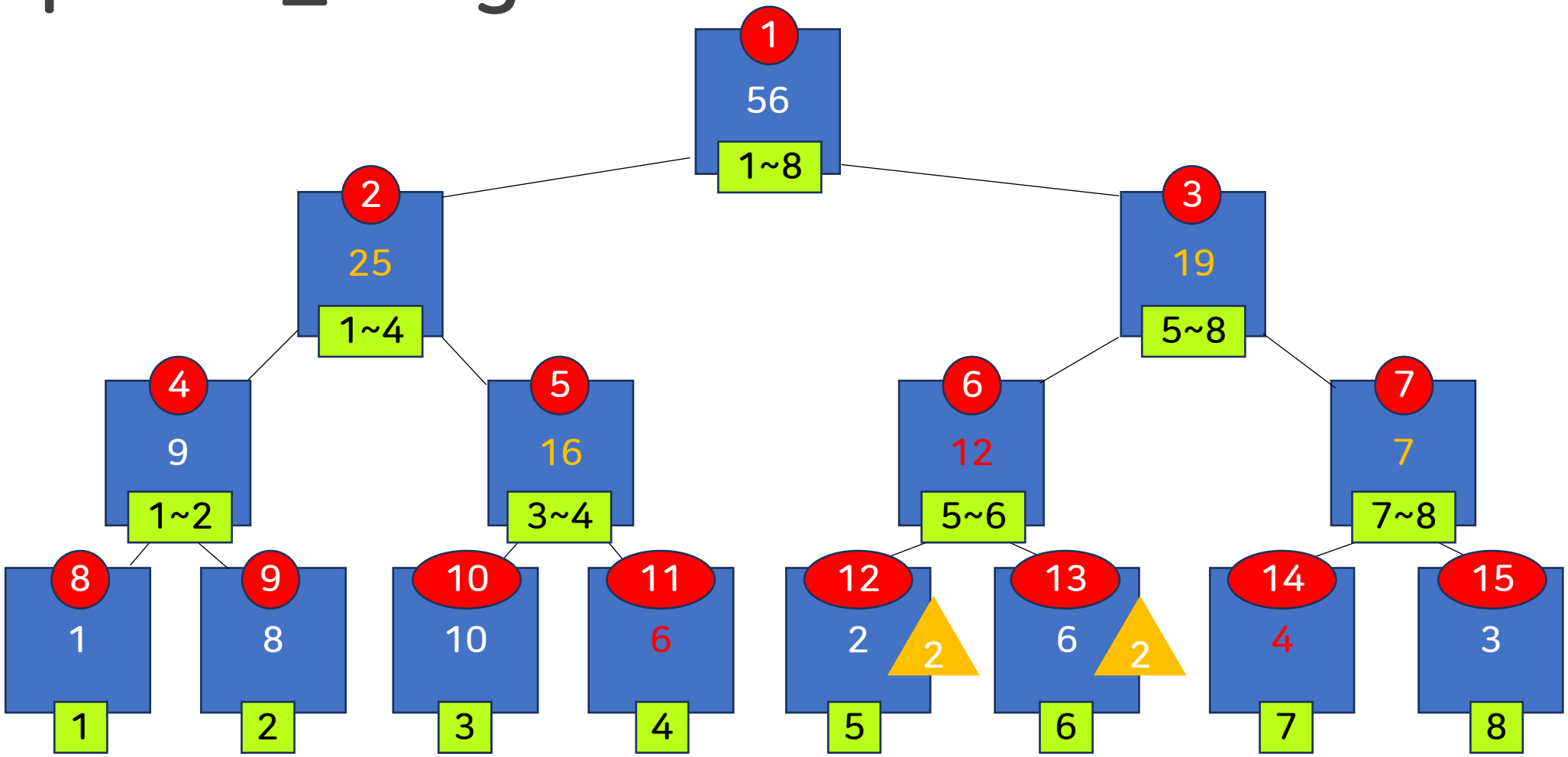
Update_range

자식 노드의 값이 변경된 것을 부모 노드에도 반영해주어야 합니다.
(연한 색으로 표시했어요)



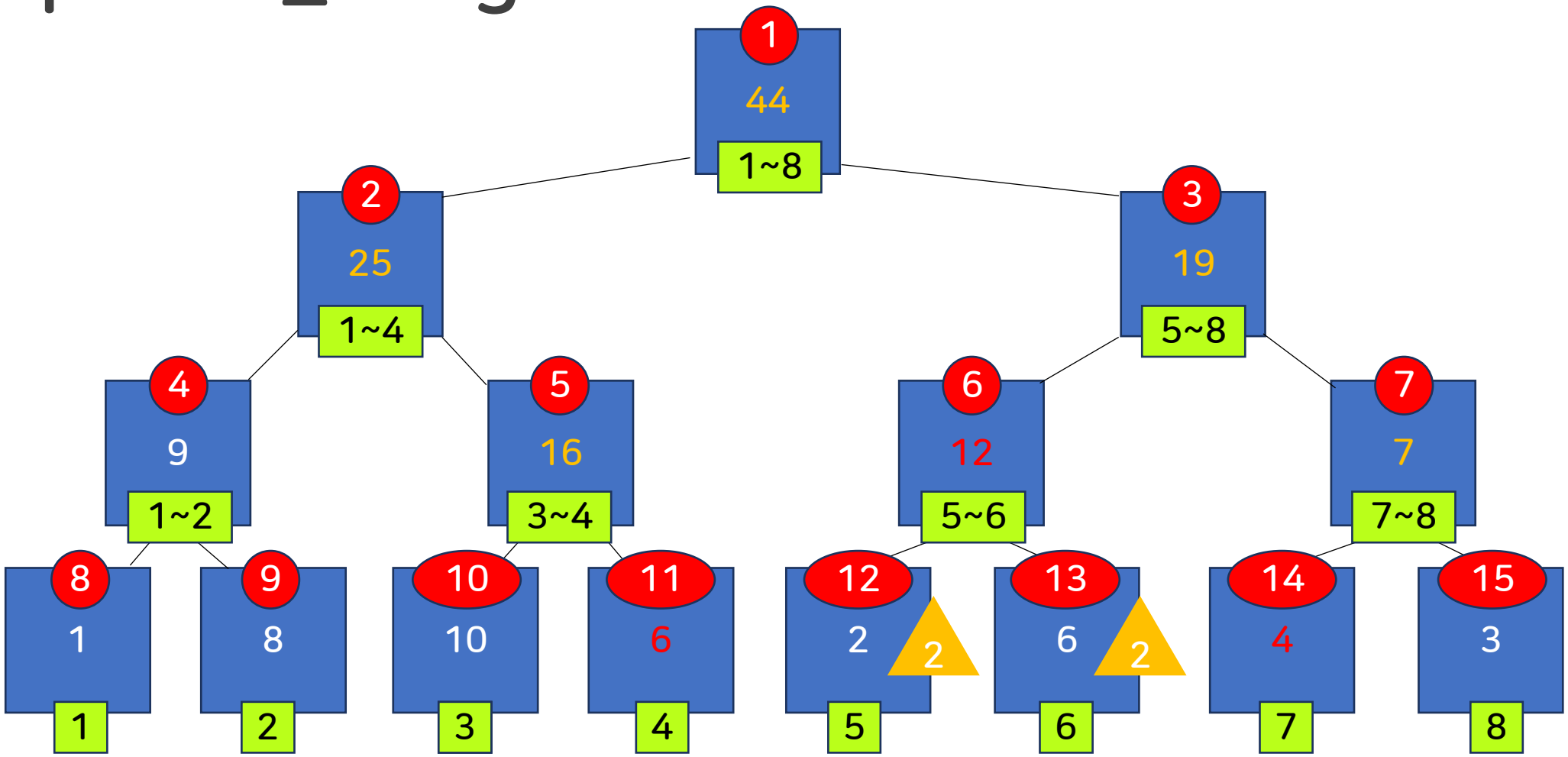
Update_range

자식 노드의 값이 변경된 것을 부모 노드에도 반영해주어야 합니다.
(연한 색으로 표시했어요)



Update_range

자식 노드의 값이 변경된 것을 부모 노드에도 반영해주어야 합니다.
(연한 색으로 표시했어요)



구현

```
void update_range(int start, int end, int node, int left, int right, long long diff) {
    // 노드에 Lazy 값이 저장되어 있나 확인
    if (lazy[node] != 0) {  // Lazy 값이 저장되어 있는 경우
        tree[node] += (end - start + 1)*lazy[node]; //  Seg Tree 노드에 Lazy 값 반영 ( 구간 길이 만큼 )
        if (start != end) { // Seg Tree 노드에 자식 노드가 있는 경우, 자식 노드의 Lazy 배열에 Lazy값 반영
            lazy[node * 2] += lazy[node];
            lazy[node * 2 + 1] += lazy[node];
        }
        lazy[node] = 0; // Lazy 값 0으로 초기화
    }

    if (right < start || end < left) return; // 노드가 범위 밖인 경우, 함수 종료

    if (left <= start && end <= right) { // 노드가 범위 안인 경우
        tree[node] += (end - start + 1)*diff; // Seg Tree 노드에 변화량인 diff 반영
        if (start != end) { // Seg Tree 노드에 자식 노드가 있는 경우, 자식 노드의 Lazt 배열에 diff값 반영
            lazy[node * 2] += diff;
            lazy[node * 2 + 1] += diff;
        }
        return; // 함수 종료
    }

    int mid = (start + end) / 2;
    update_range(start, mid, node * 2, left, right, diff); // 재귀적 호출
    update_range(mid + 1, end, node * 2 + 1, left, right, diff); // 재귀적 호출

    tree[node] = tree[node * 2] + tree[node * 2 + 1]; // 자식 노드에서 변화된 값을 반영
}
```

변수 설명

start : Seg Tree 노드가 가리키는 시작구간

end : Seg Tree 노드가 가리키는 끝 구간

node : Seg Tree의 node 번호

left : 변경 구간의 시작

right : 변경 구간의 끝

diff : 변화량

tree : Seg Tree를 나타내는 배열

lazy : Lazy를 저장할 배열

BOJ 10999 구간 합 구하기 2

구간 합 구하기 2

구간 합 구하기에 대해서 Lazy Propagation만 구현하면 된다.
위에 있는 코드만 그대로 구현하면 정!답!
(틀리신 분은 SUM에서 Lazy를 Check 안하셔서 그런거에요!)

BOJ 1395 스위치 ~~(모여봐요 동물의 숲) ((나비보벳따우~))~~

스위치

Lazy Propagation인건 아는데 어떻게?!
Lazy는 켜졌다 꺼졌다만 반복하니 boolean형식으로 구현하고
Seg Tree 노드는 꺼져있던 만큼 켜져야 하므로
(길이 – 현재 값)이 켜져야한다!
정답 코드는 다음 장!

스위치

```
void update_range(int start, int end, int node, int left, int right, int diff) {
    if (lazy[node] != 0) {
        tree[node] = (end - start + 1) - tree[node];
        if (start != end) {
            lazy[node * 2] = !lazy[node * 2];
            lazy[node * 2 + 1] = !lazy[node * 2 + 1];
        }
        lazy[node] = 0;
    }

    if (right < start || end < left) return;

    if (left <= start && end <= right) {
        tree[node] = (end - start + 1) - tree[node];
        if (start != end) {
            lazy[node * 2] = !lazy[node * 2];
            lazy[node * 2 + 1] = !lazy[node * 2 + 1];
        }
        return;
    }

    int mid = (start + end) / 2;
    update_range(start, mid, node * 2, left, right, diff);
    update_range(mid + 1, end, node * 2 + 1, left, right, diff);

    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
```

여기 달라졌어요!

여기 달라졌어요!

챕터 2: Plane Sweeping

Lazy를 이용해서 Plane Sweeping을 구현해봅시다!

자료 출처 : https://codedoc.tistory.com/421

BOJ 2563 색종이 ~~(설명안하고 문제주기)~~

색종이

Plane Sweeping을 이용하지 않고 풀기
전체 배경을 배열로 잡고 사각형이 차지하는 부분에 1을 채워 넣어서 나중에 전체를 탐색해서 계산하면 된다.

Plane Sweeping

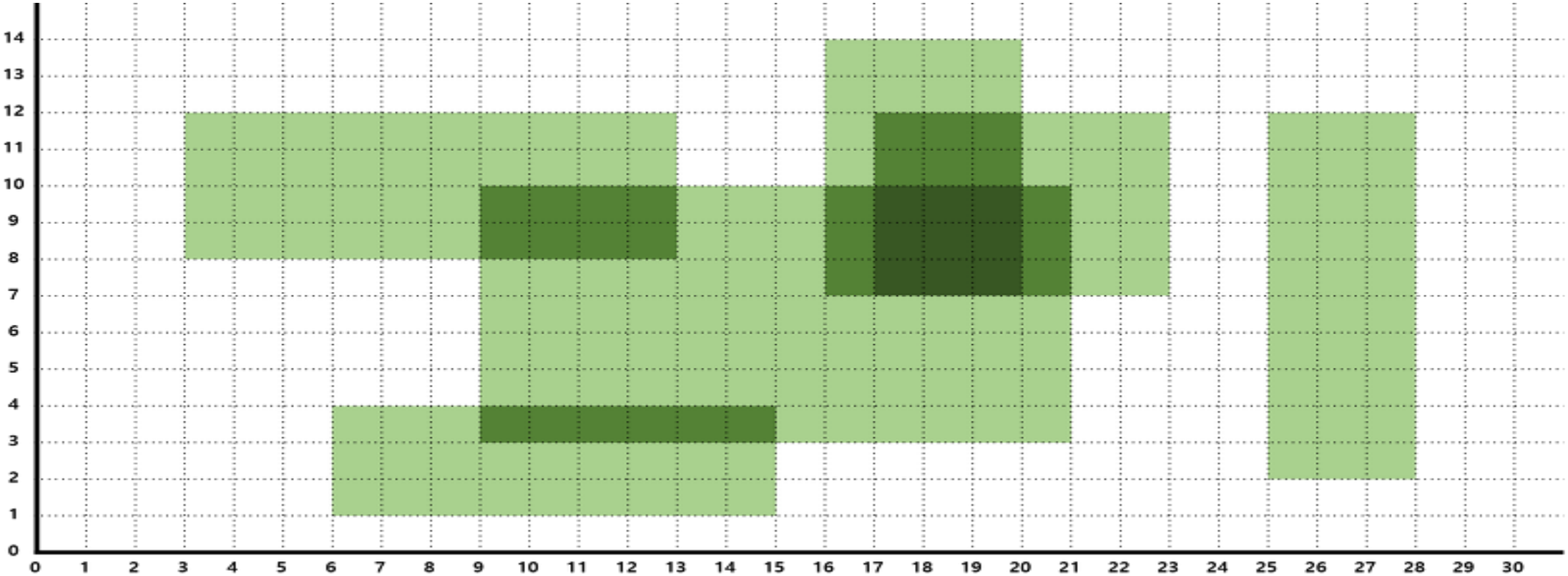
오~ 그러면 Plane Sweeping 안 배워도 되는 거 아니야?
~~(응 아니야~)~~

색종이 문제처럼 구현할 때의 단점 : 공간 & 시간 복잡도의 비효율성 - 둘 다 O (X*Y)의 복잡도를 가진다.

더 효율적으로 하는 방법이 바로 Plane Sweeping!

Plane Sweeping

아래와 같이 직사각형이 겹쳐졌을 때 색칠된 부분의 넓이 합 구하기



Plane Sweeping

초기화 방법

1. 가로선을 포함한 X축과 평행한 선으로 자른다.
2. 잘라진 구간에 따라 번호를 지정하고 Segment Tree를 구성한다.
3. 각 직사각형의 세로선 데이터를 얻어오고, 각 선들을 x좌표 기준으로 정렬한다.

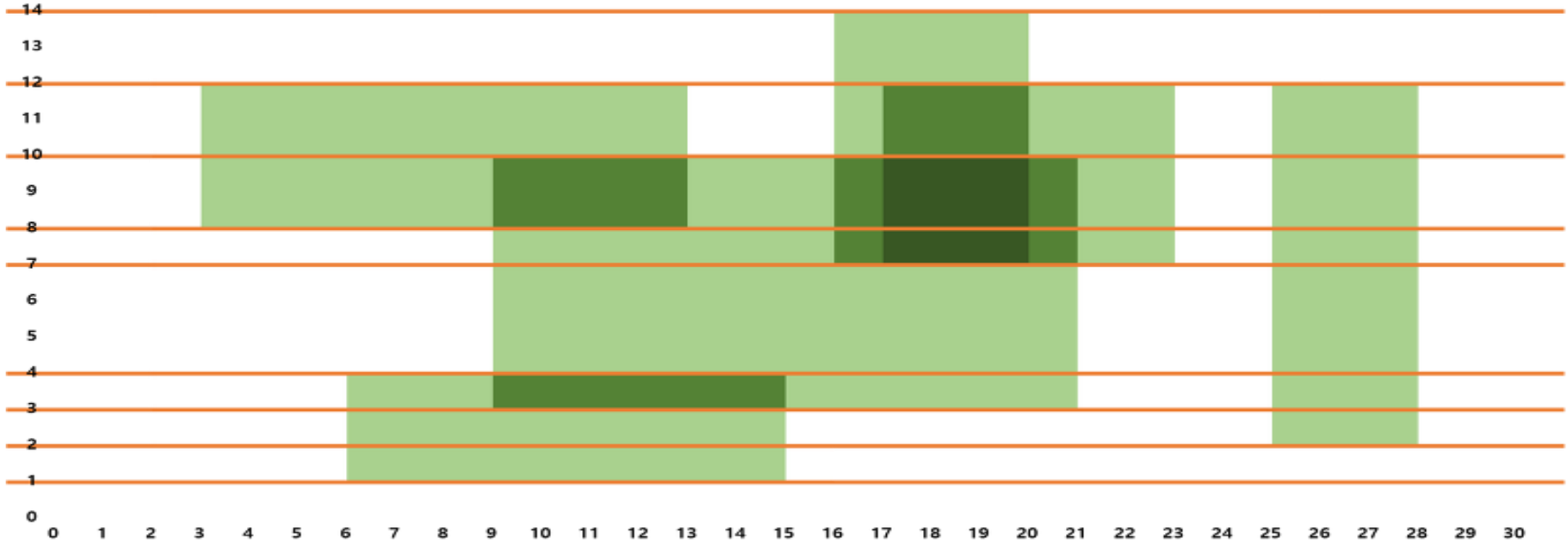
Plane Sweeping

초기화 방법

1. 가로선을 포함한 X축과 평행한 선으로 자른다.
2. 잘라진 구간에 따라 번호를 지정하고 Segment Tree를 구성한다.
3. 각 직사각형의 세로선 데이터를 얻어오고, 각 선들을 x좌표 기준으로 정렬한다.

Plane Sweeping

아래와 같이 직사각형이 겹쳐졌을 때 색칠된 부분의 넓이 합 구하기



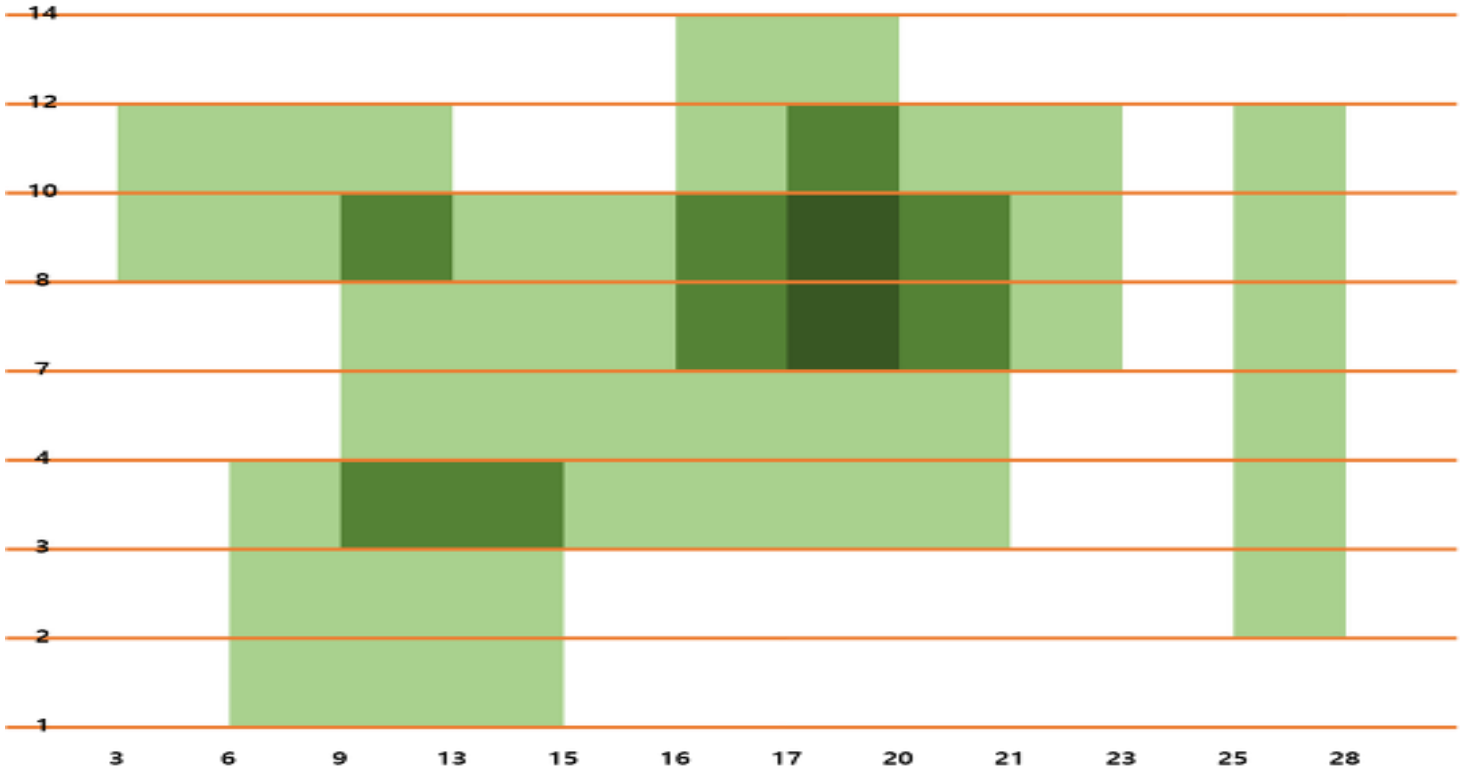
Plane Sweeping

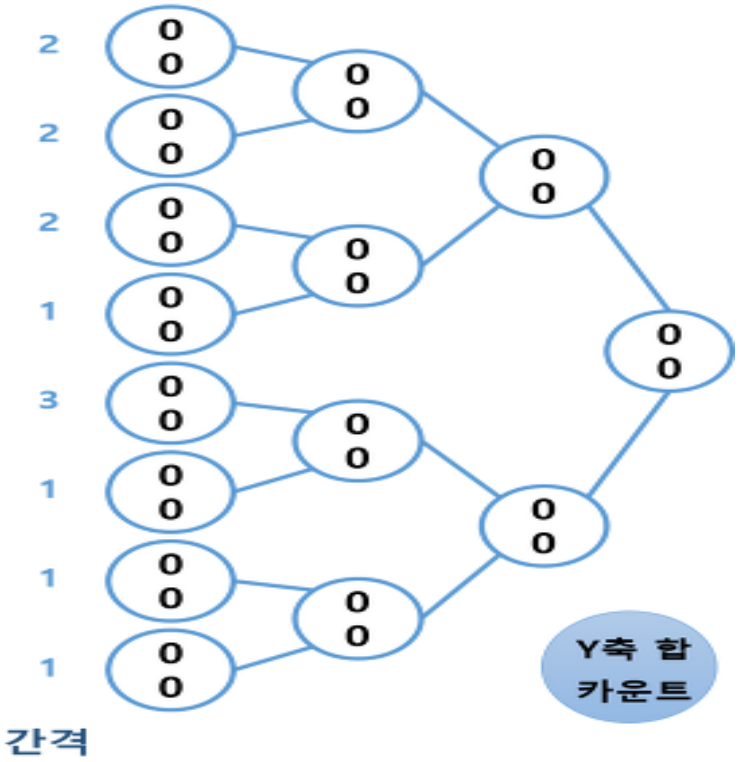
초기화 방법

1. 가로선을 포함한 X축과 평행한 선으로 자른다.
2. 잘라진 구간에 따라 번호를 지정하고 Segment Tree를 구성한다.
3. 각 직사각형의 세로선 데이터를 얻어오고, 각 선들을 x좌표 기준으로 정렬한다.

Plane Sweeping

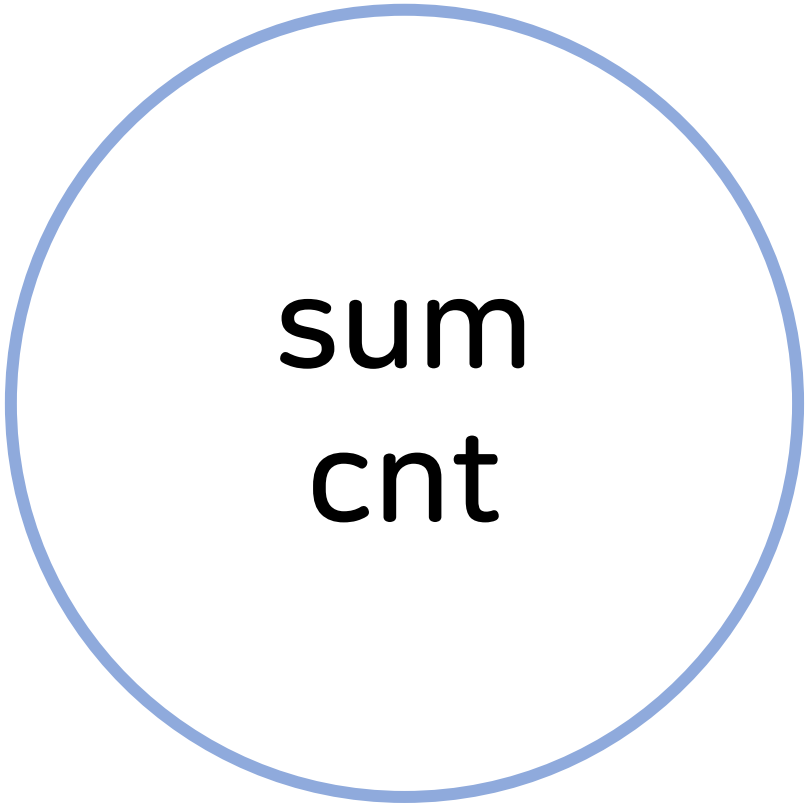
아래와 같이 직사각형이 겹쳐졌을 때 색칠된 부분의 넓이 합 구하기

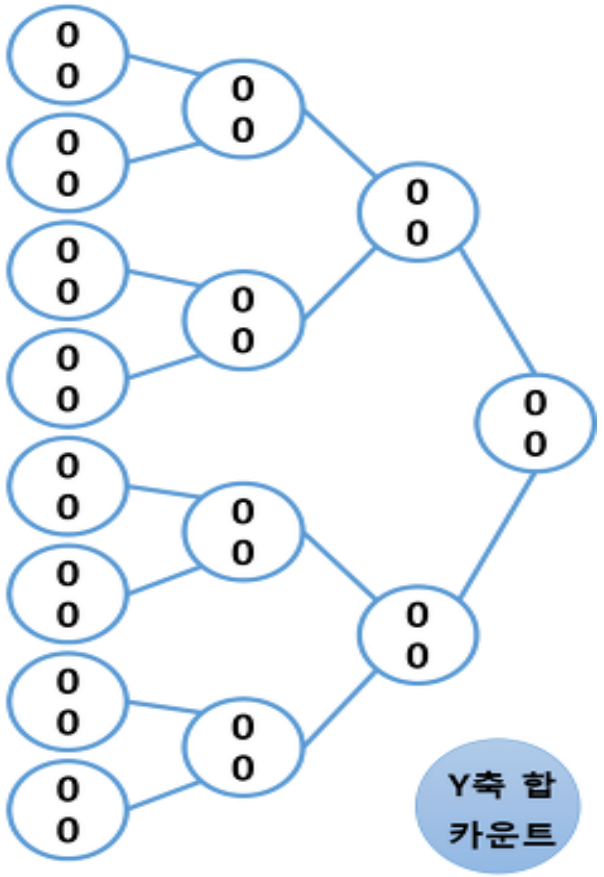




Plane Sweeping

Segment Tree의 노드의 의미





Plane Sweeping

초기화 방법

1. 가로선을 포함한 X축과 평행한 선으로 자른다.
2. 잘라진 구간에 따라 번호를 지정하고 Segment Tree를 구성한다.
3. 각 직사각형의 세로선 데이터를 얻어오고, 각 선들을 x좌표 기준으로 정렬한다.

Plane Sweeping

이렇게 해야 사각형의 가장 왼쪽 변부터 Sweeping이 가능합니다.

Plane Sweeping

갱신 방법

1. 직사각형의 세로선 데이터를 받아온다.
2. 트리의 세로선에 해당하는 구간의 노드를 구한다.
3. 세로선의 종류에 따라서 다음과 같이 실행한다.
    1) 세로선이 시작선일 경우,
        해당하는 구간의 노드의 cnt값을 1 증가시킨다.
    2) 세로선이 종료선일 경우,
        해당하는 구간의 노드의 cnt값을 1 감소시킨다.

Plane Sweeping

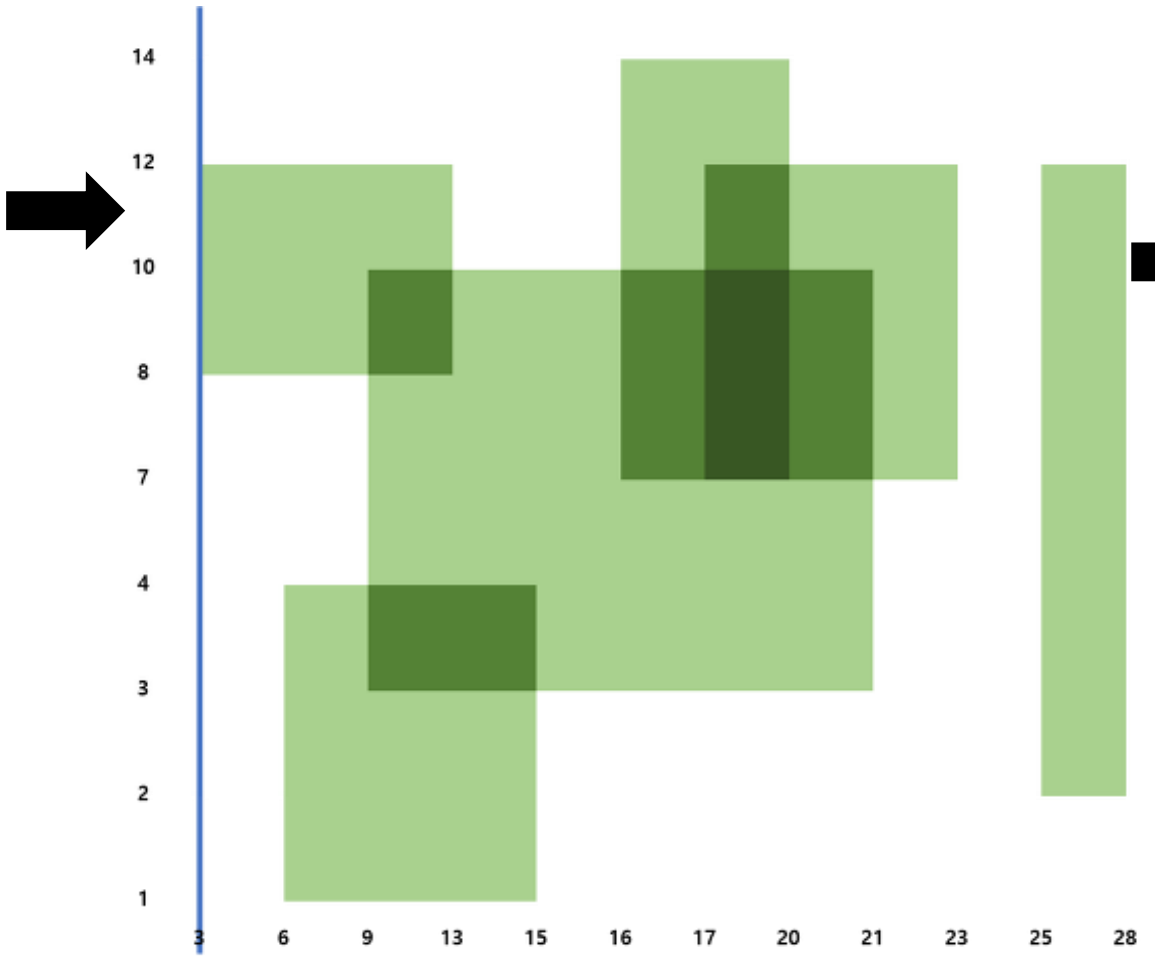
4. 해당하는 구간의 노드를 포함해서, 다시 루트노드까지 올라가면서 다음과 같이 실행한다.
   1) 위치한 노드의 cnt값이 0인 경우,
      하위 두 자식 노드의 sum값을 더해 위치한 노드의 sum에 저장시킨다.
   2) 위치한 노드의 cnt값이 0보다 큰 경우,
      해당하는 y축 구간 크기를 sum에 저장시킨다.

Plane Sweeping

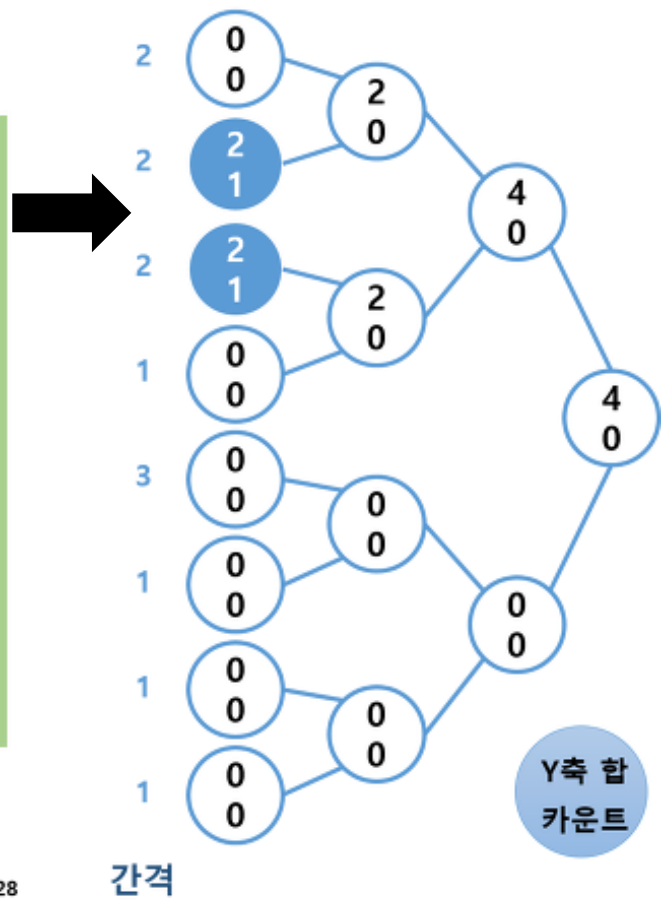
가장 왼쪽 변부터 Sweeping을 하면서 1~4의 방법을 반복합니다.

(x축 구간 * 루트 노드의 sum)이 각 과정의 넓이 입니다.

Plane Sweeping



찾은 시작선
(3, 8~12)



3번 과정)
8~12를 나타내는 노드의 cnt를 1증가시킵니다.
(시작선이기 때문)

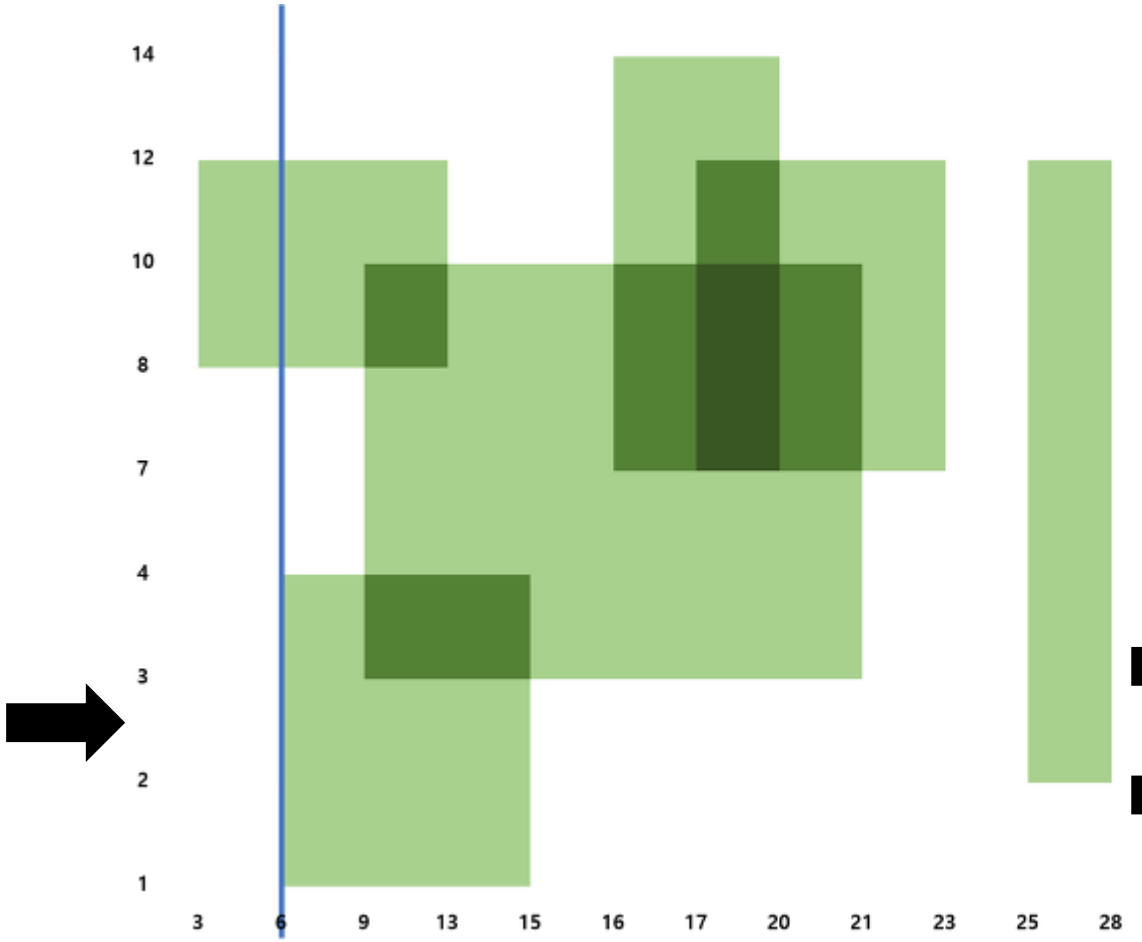
4번 과정)
노드의 cnt값 > 0인 경우, 자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우, 해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

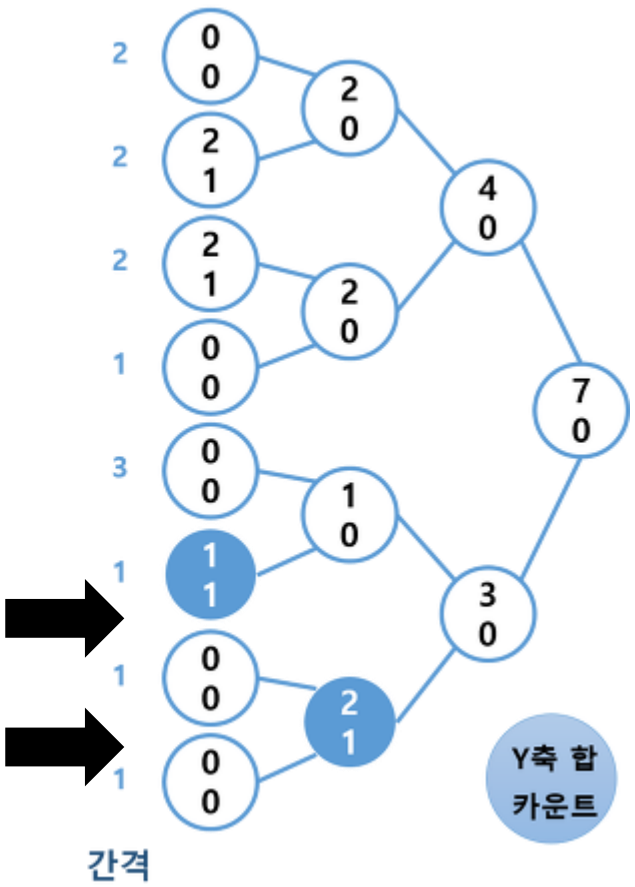
Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 시작선
(6, 1~4)

이 때 넓이는 (6-3)*4 =12





3번 과정)
1~4를 나타내는 노드의 cnt를 1증가시킵니다.
(시작선이기 때문)

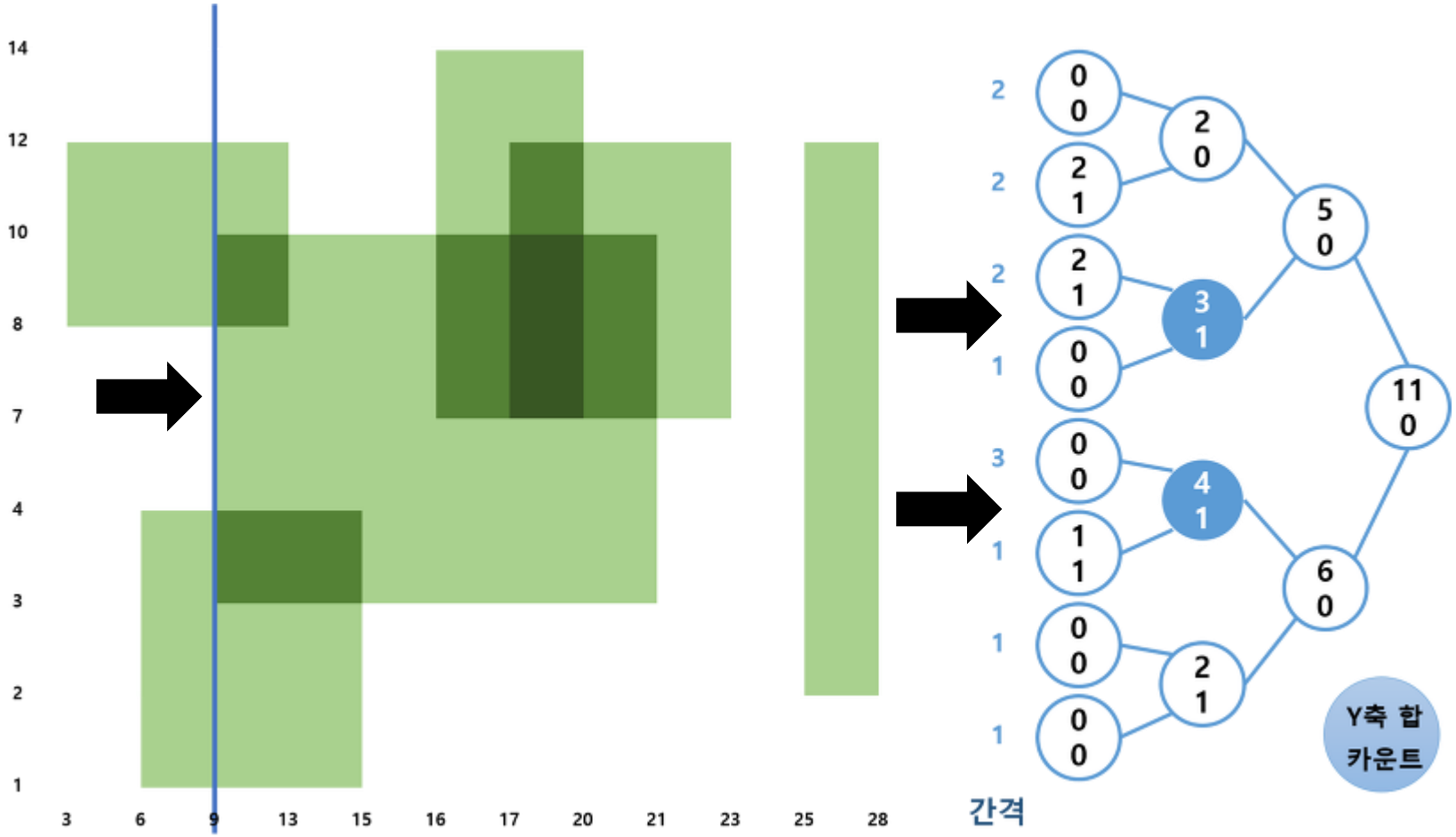
4번 과정)
노드의 cnt값 > 0인 경우,
자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우,
해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 시작선
(9, 3~10)

이 때 넓이는 (9-6)*7 =21



3번 과정)
3~10을 나타내는 노드의 cnt를 1증가시킵니다.
(시작선이기 때문)

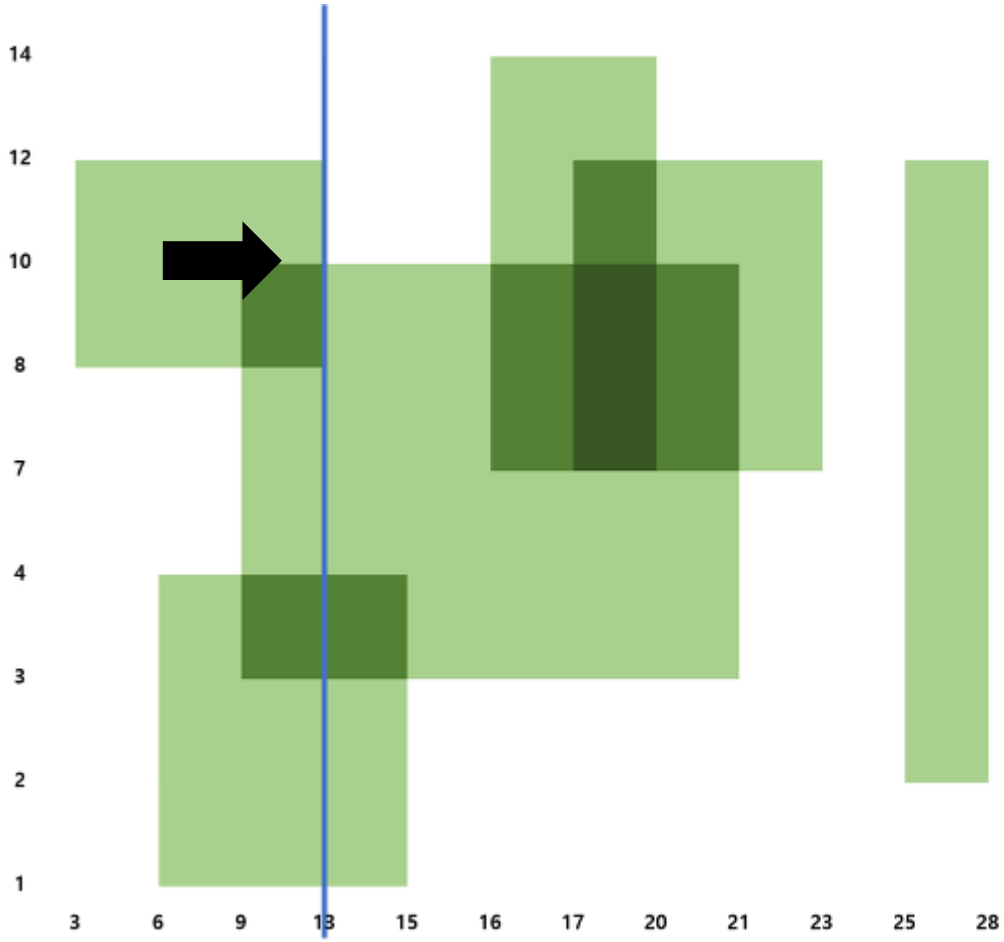
4번 과정)
노드의 cnt값 > 0인 경우,
자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우,
해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

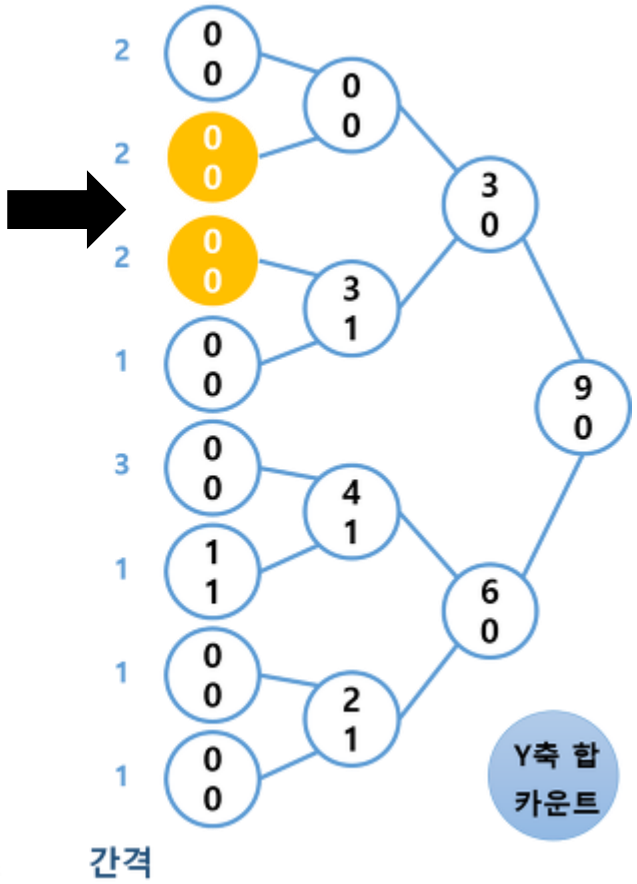
Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 종료선
(13, 8~12)

이 때 넓이는 (13-9)*11
=44





3번 과정)
8~12를 나타내는 노드의 cnt를 1감소시킵니다.
(종료선이기 때문)

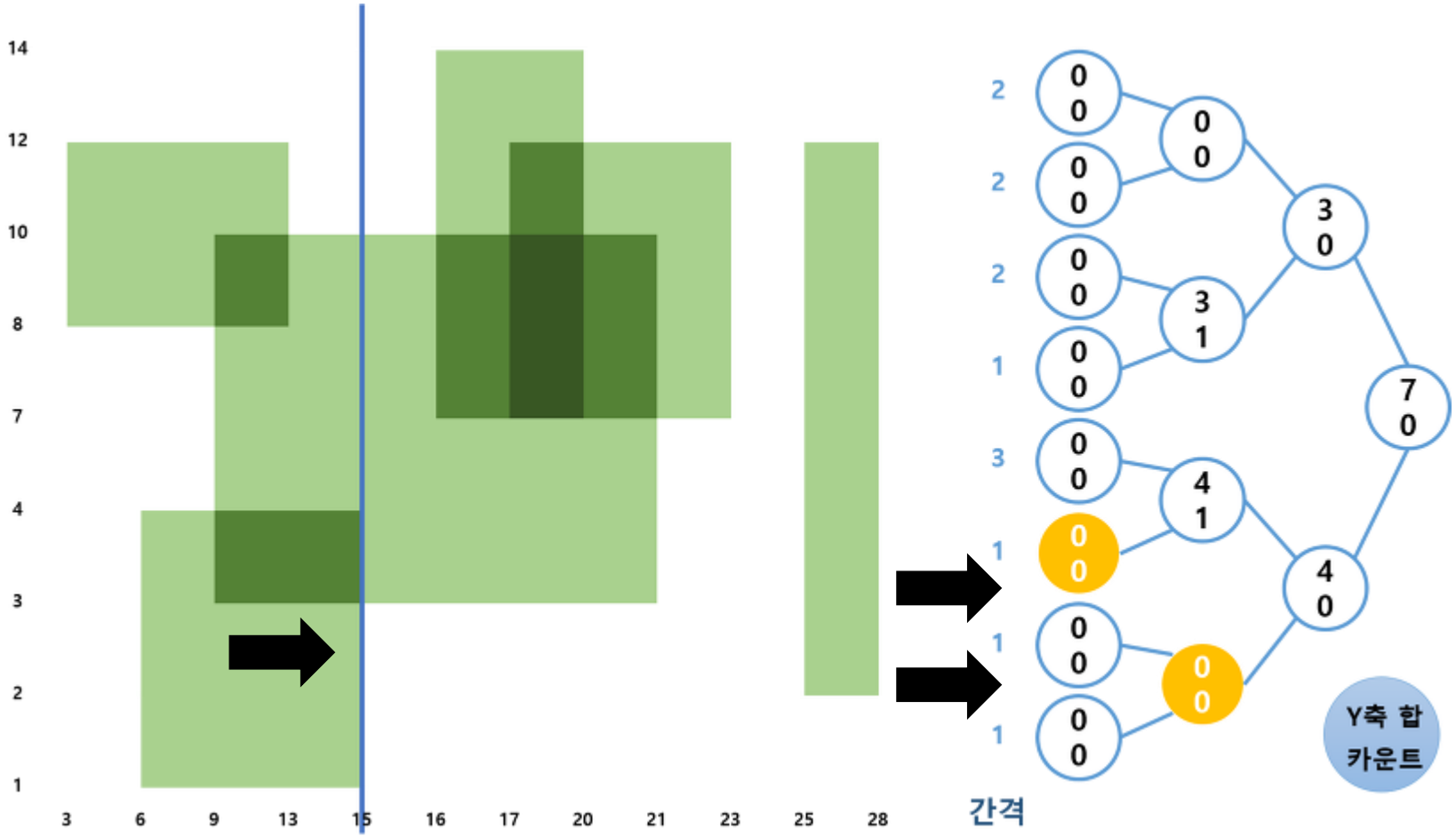
4번 과정)
노드의 cnt값 > 0인 경우, 자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우, 해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 종료선
(15, 1~4)

이 때 넓이는 (15-13)*9
=18



3번 과정)
1~4를 나타내는 노드의 cnt를 1감소시킵니다.
(종료선이기 때문)

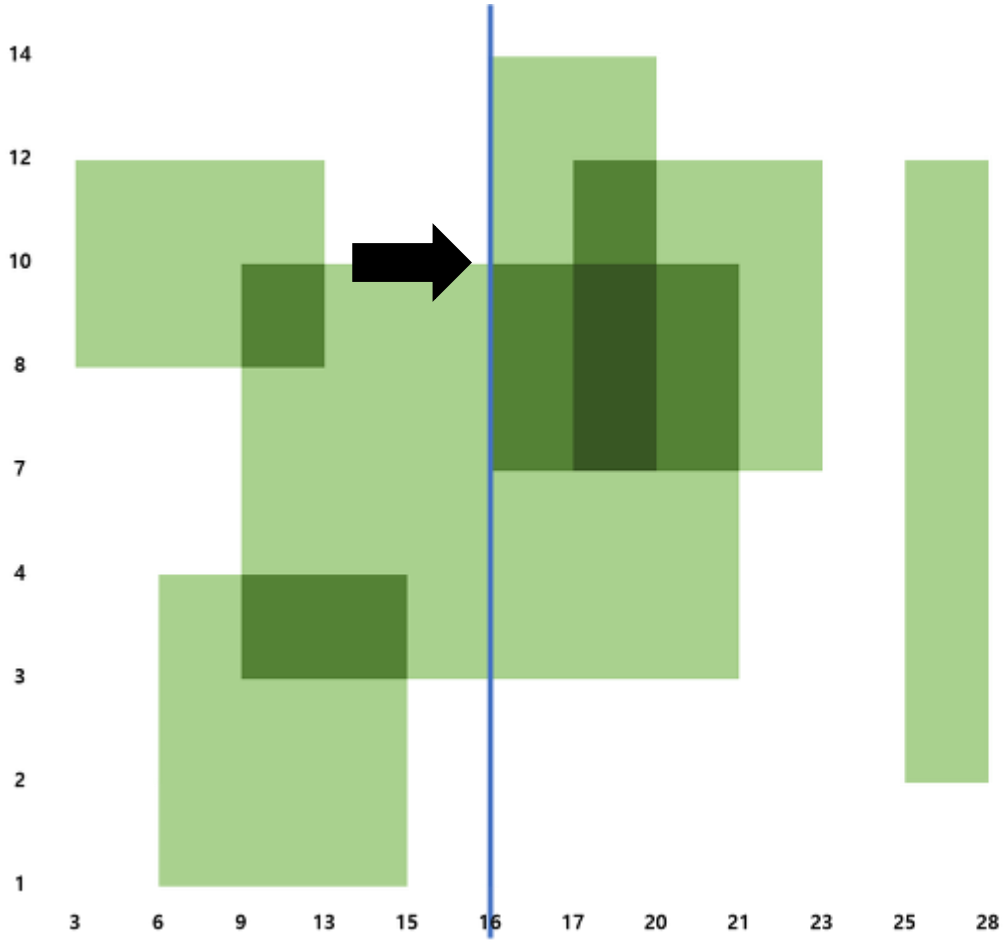
4번 과정)
노드의 cnt값 > 0인 경우,
자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우,
해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

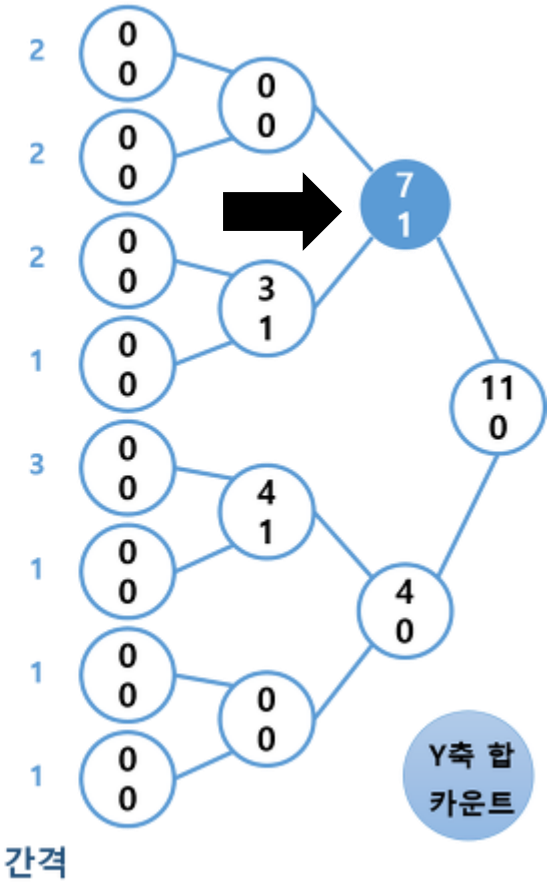
Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 시작선
(16, 7~14)

이 때 넓이는 (16-15)*7 =7





3번 과정)
7~14를 나타내는 노드의 cnt를 1증가시킵니다.
(시작선이기 때문)

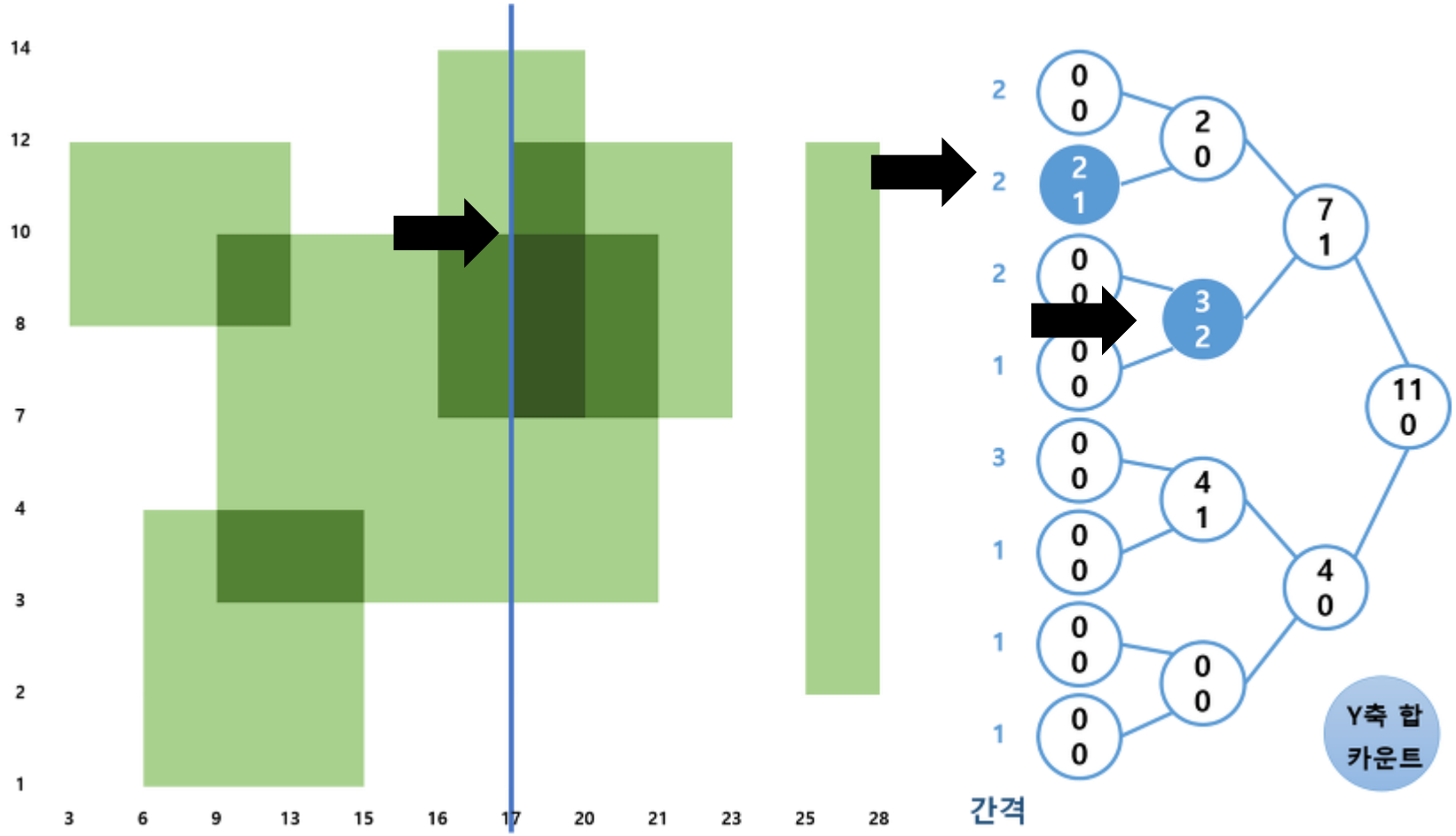
4번 과정)
노드의 cnt값 > 0인 경우,
자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우,
해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 시작선
(17, 7~12)

이 때 넓이는 (17-16)*11
=11



3번 과정)
7~12를 나타내는 노드의 cnt를 1증가시킵니다.
(시작선이기 때문)

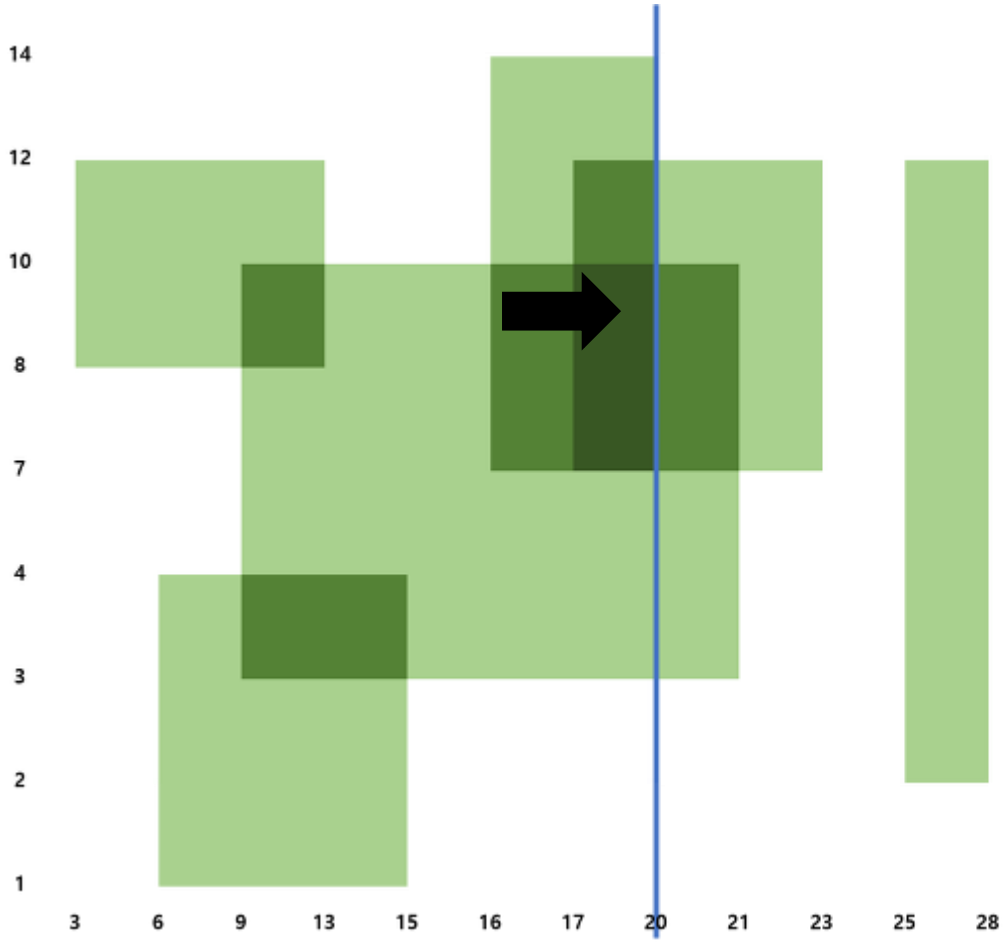
4번 과정)
노드의 cnt값 > 0인 경우, 자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우, 해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

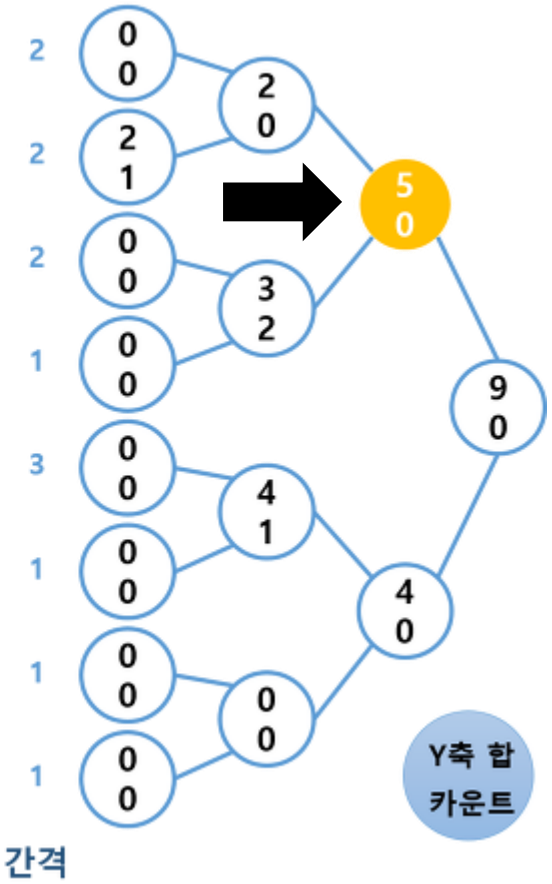
Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 종료선
(20, 7~14)

이 때 넓이는 (20-17)*11
=33





3번 과정)
7~14를 나타내는 노드의 cnt를 1감소시킵니다.
(종료선이기 때문)

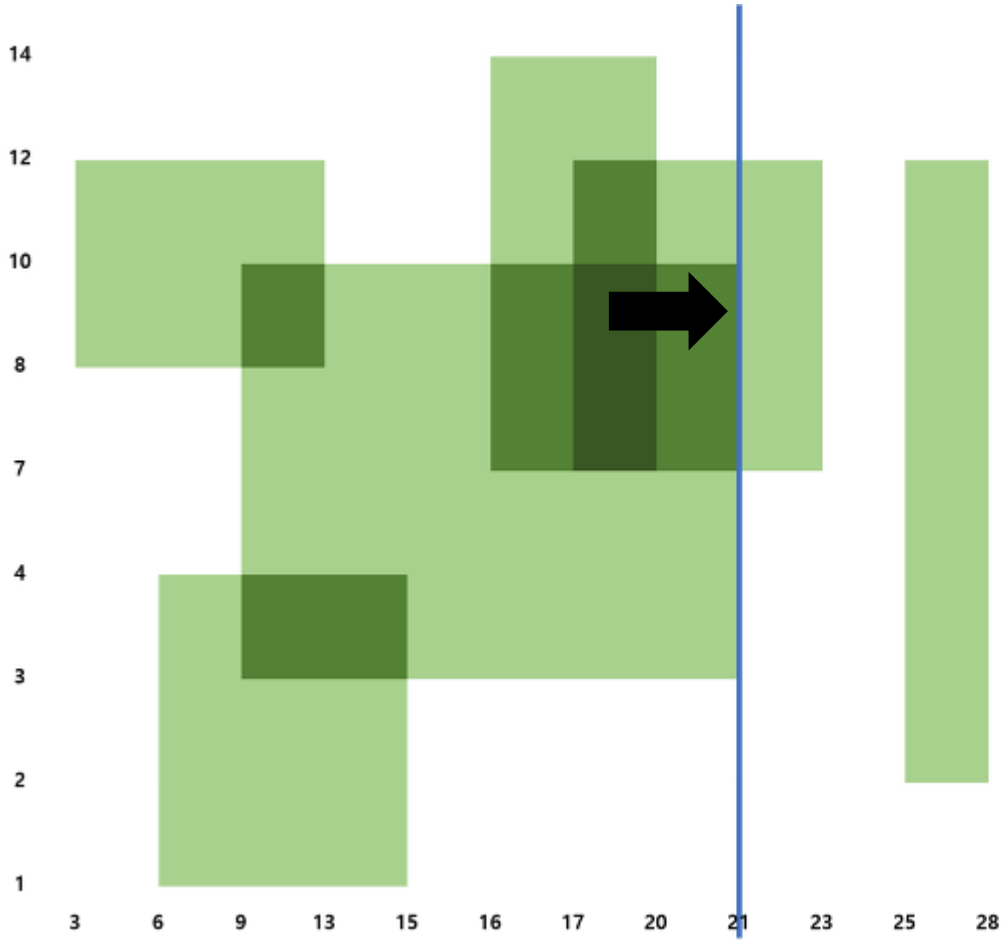
4번 과정)
노드의 cnt값 > 0인 경우, 자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우, 해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

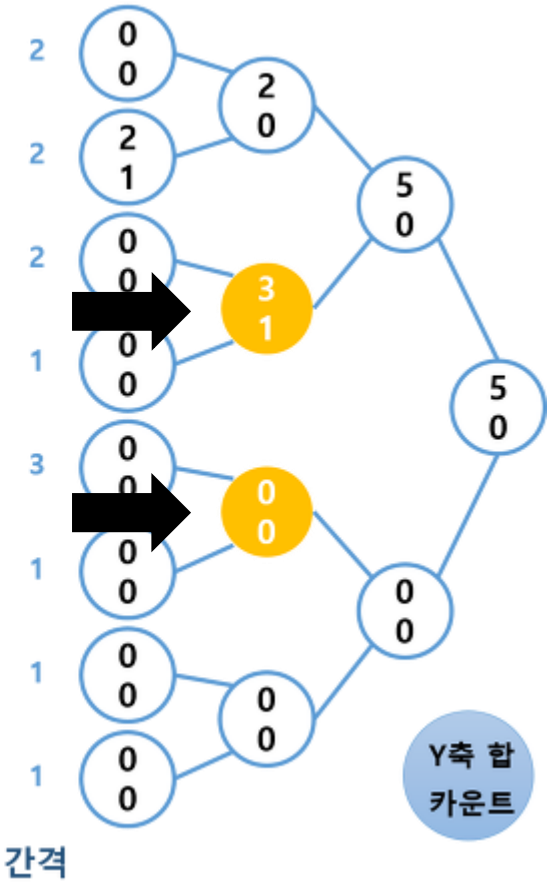
Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 종료선
(21, 3~10)

이 때 넓이는 (21-20)*9 =9





3번 과정)
3~10을 나타내는 노드의 cnt를 1감소시킵니다.
(종료선이기 때문)

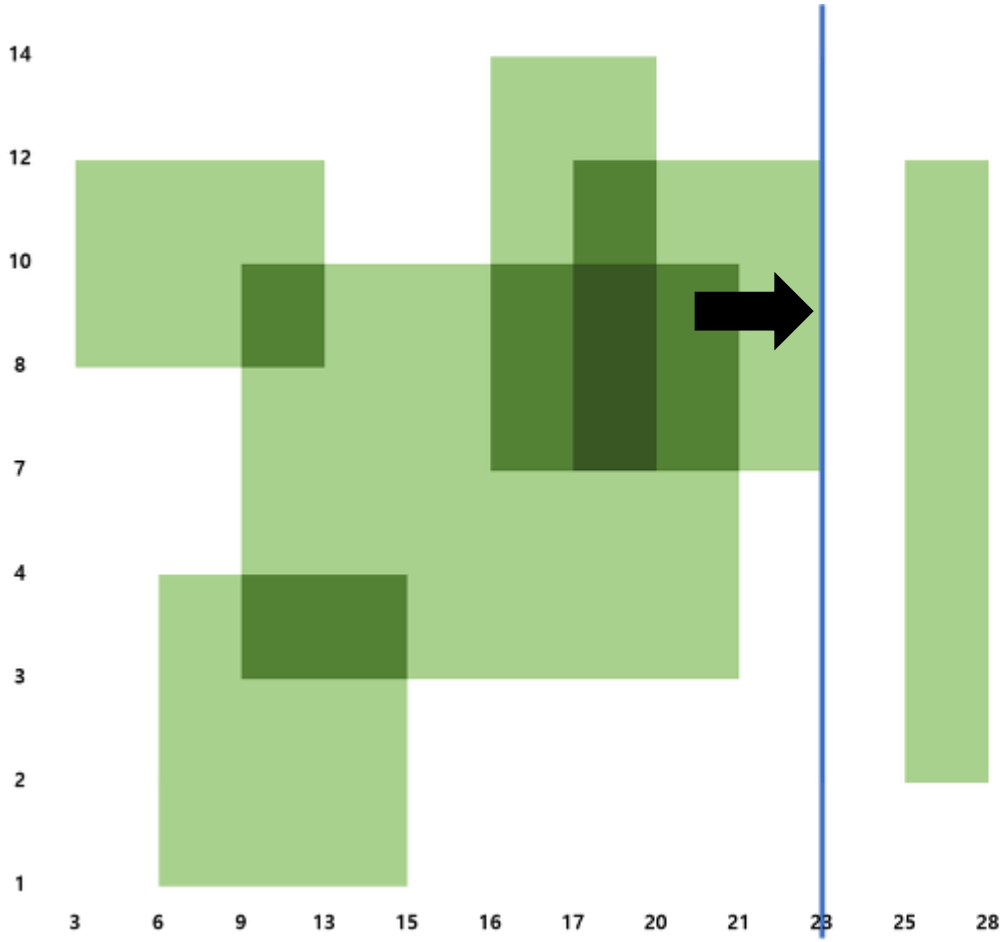
4번 과정)
노드의 cnt값 > 0인 경우,
자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우,
해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

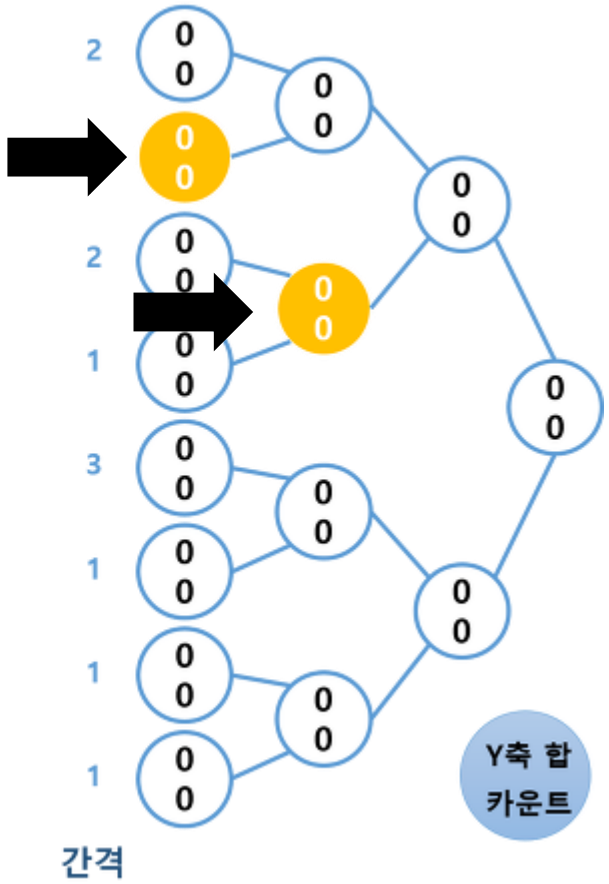
Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 종료선
(23, 7~12)

이 때 넓이는 (23-21)*5
=10





3번 과정)
7~12를 나타내는 노드의 cnt를 1감소시킵니다.
(종료선이기 때문)

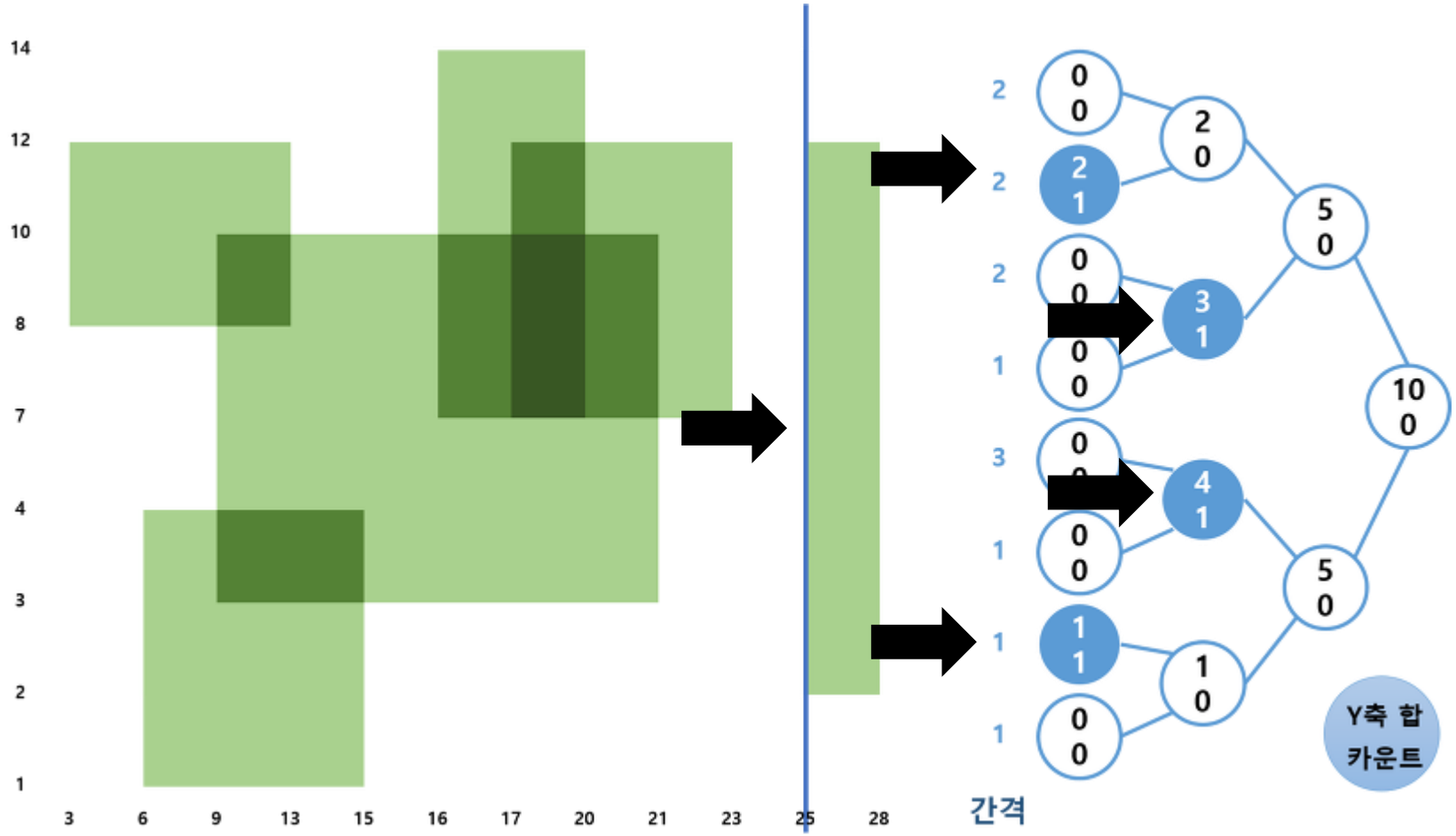
4번 과정)
노드의 cnt값 > 0인 경우,
자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우,
해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 시작선
(25, 2~12)

이 때 넓이는 (25-23)*0 =0



3번 과정)
2~12를 나타내는 노드의 cnt를 1증가시킵니다.
(시작선이기 때문)

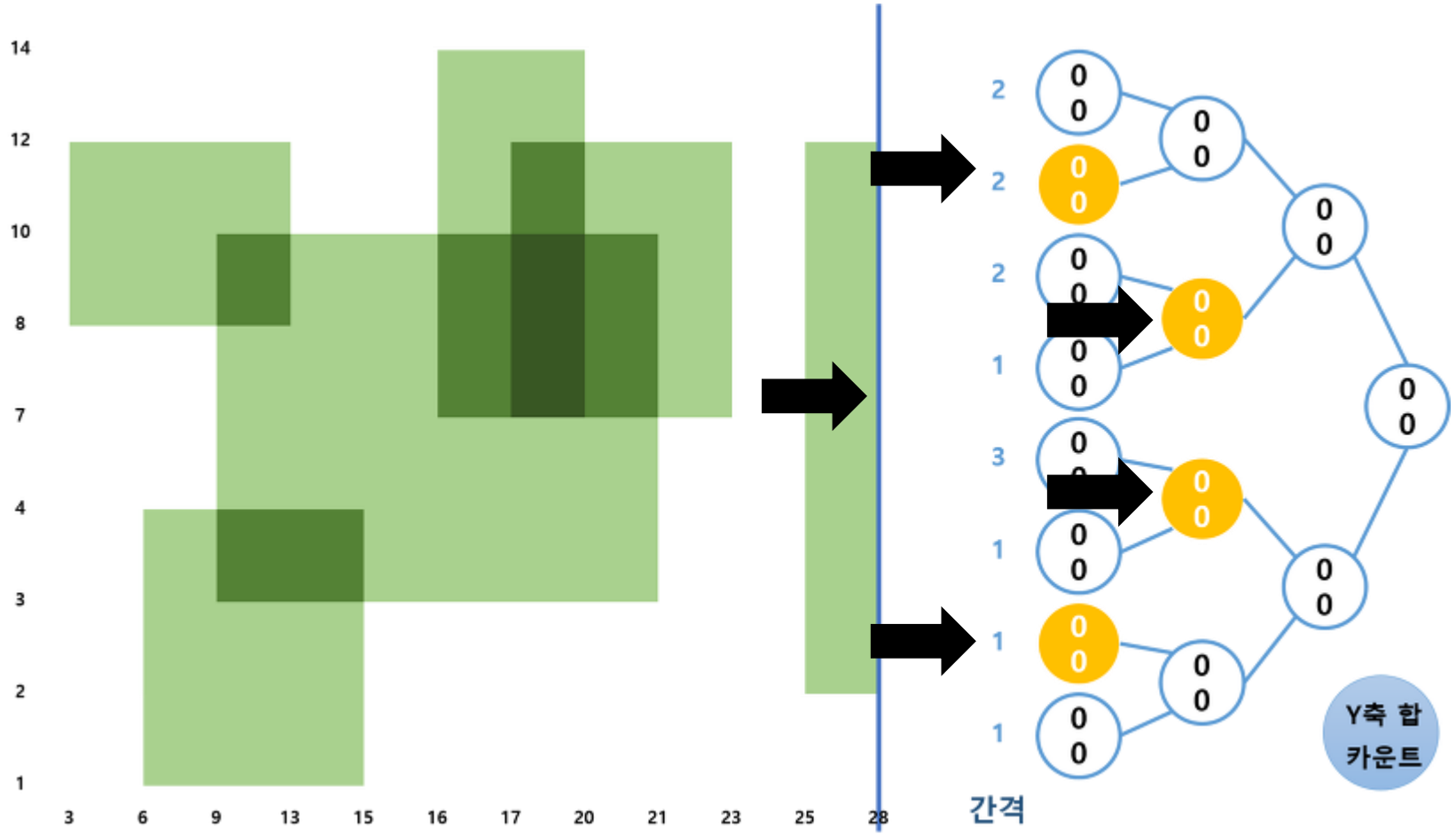
4번 과정)
노드의 cnt값 > 0인 경우,
자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우,
해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

Plane Sweeping

현재 선과 이전 선의 x차이	갱신 이전의 루트노드 sum

찾은 종료선
(28, 2~12)

이 때 넓이는 (28-25)*10
=30

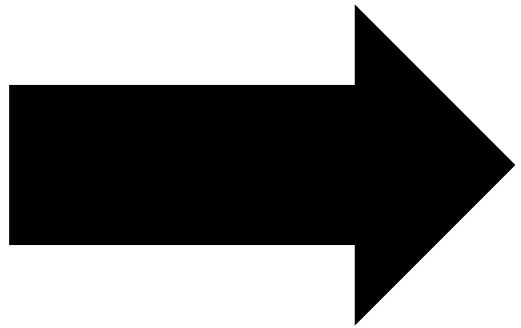


3번 과정)
2~12를 나타내는 노드의 cnt를 1감소시킵니다.
(종료선이기 때문)

4번 과정)
노드의 cnt값 > 0인 경우,
자식노드의 sum값을 더해 노드의 sum에 저장합니다.
노드의 cnt값이 ≥ 0인 경우,
해당하는 y축 구간의 길이를 sum에 저장합니다.
위의 과정을 루트노드까지 반복합니다.

Plane Sweeping

각 과정의 사각형 넓이
(6-3) * 4 = 12
(9-6) * 7 = 21
(13-9) * 11 = 44
(15-13) * 9 = 18
(16-15) * 7 = 7
(17-16) * 11 = 11
(20-17) * 11 = 33
(21-20) * 9 = 9
(23-21) * 5 = 10
(25-23) * 0 = 0
(28-25) * 10 = 30



사각형의 총 넓이
= (각 과정 넓이 합)
= 195

구현

```
void update_range(int start, int end, int node, int left, int right, int diff) {

    if (end < left || start > right) return; // 포함되지 않는 구간이면, 함수종료

    if (left <= start && end <= right) { // 포함되는 구간이면 노드의 cnt값 ++(diff 1)
        cnt[node] += diff;
    }
    else { // 일부 포함되는 구간이면 자식노드에서 update_range
        int mid = (start + end) / 2;
        update_range(start, mid, node * 2, left, right, diff);
        update_range(mid + 1, end, node * 2 + 1, left, right, diff);
    }

    if (cnt[node] > 0) { // 4번과정 : cnt가 0보다 크면, 노드의 sum 갱신
        sum[node] = end - start + 1;
    }
    else if (start != end) { // 4번과정 : cnt가 0이면, 자식노드의 합
        sum[node] = sum[node * 2] + sum[node * 2 + 1];
    }
    else { // sum 초기화
        sum[node] = 0;
    }
}
```

변수 설명

start : Seg Tree 노드가 가리키는 시작구간

end : Seg Tree 노드가 가리키는 끝 구간

node : Seg Tree의 node 번호

left : 변경 구간의 시작

right : 변경 구간의 끝

diff : 변화량 (1)

cnt : Tree의 cnt값 저장

sum : Tree의 sum값 저장

BOJ 2563 색종이 ~~(수정/제출)~~

색종이

Plane Sweeping을 이용하고 풀기

BOJ 3392 화성지도

화성지도

Plane Sweeping을 이용하고 풀기
확장된 색종이

추가문제

BOJ 12844 XOR
BOJ 2669 직사각형 네 개의 합집합의 면적 구하기
BOJ 2672 여러 직사각형의 전체 면적 구하기
BOJ 2601 도서실 카펫
BOJ 2820 자동차 공장

감사합니다!

끝