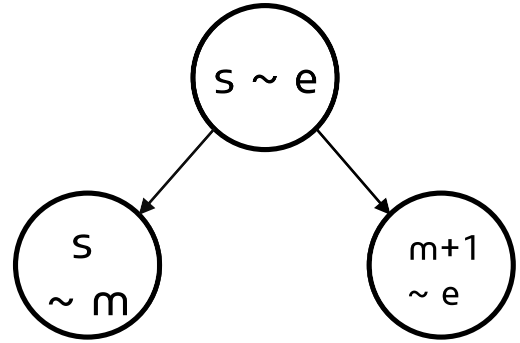
세그먼트 트리(Segment Tree)

Gaussian

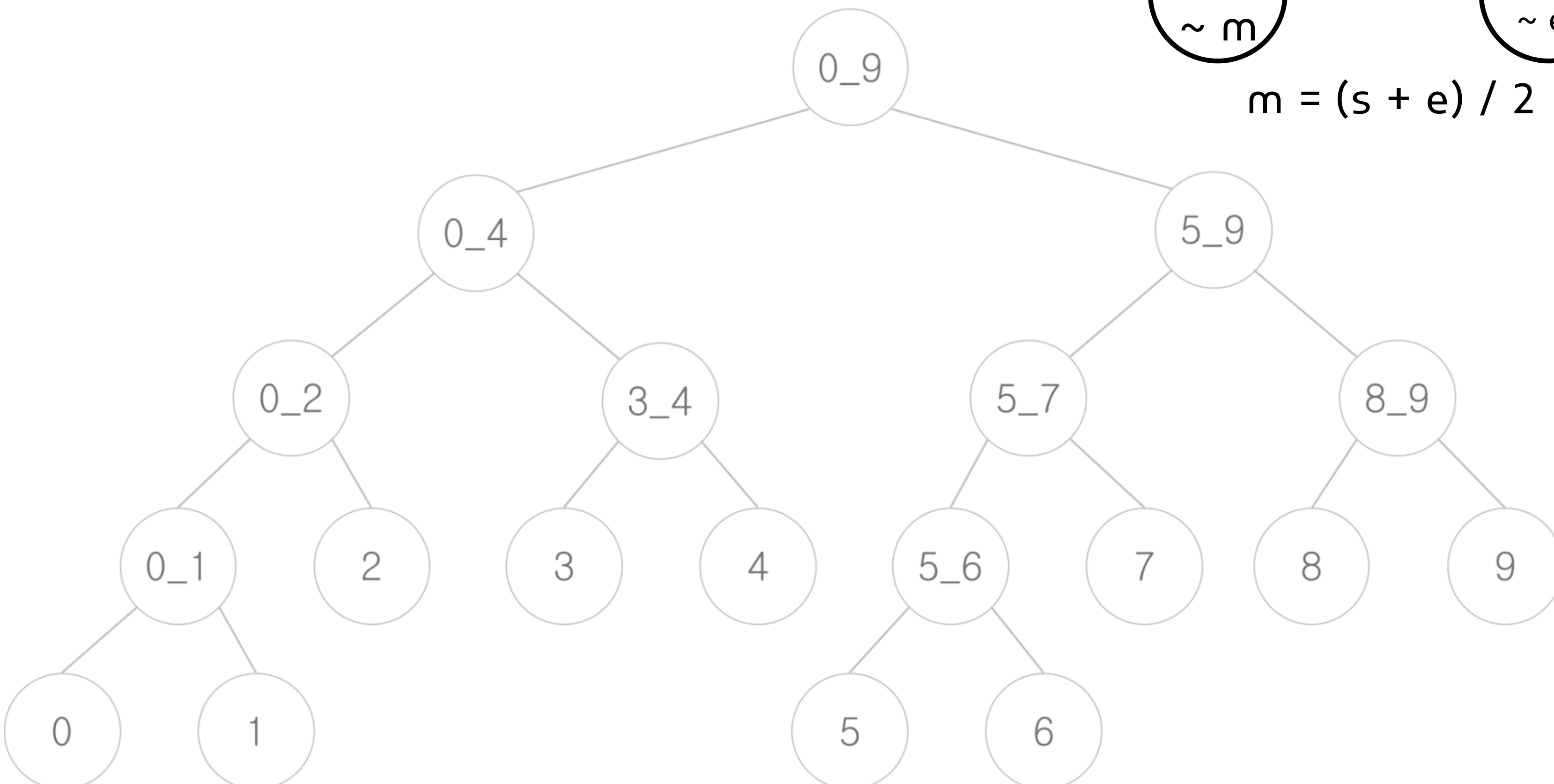
세그먼트 트리

Range Minimum Query

- 세그먼트 트리는 다음과 같은 형식입니다.



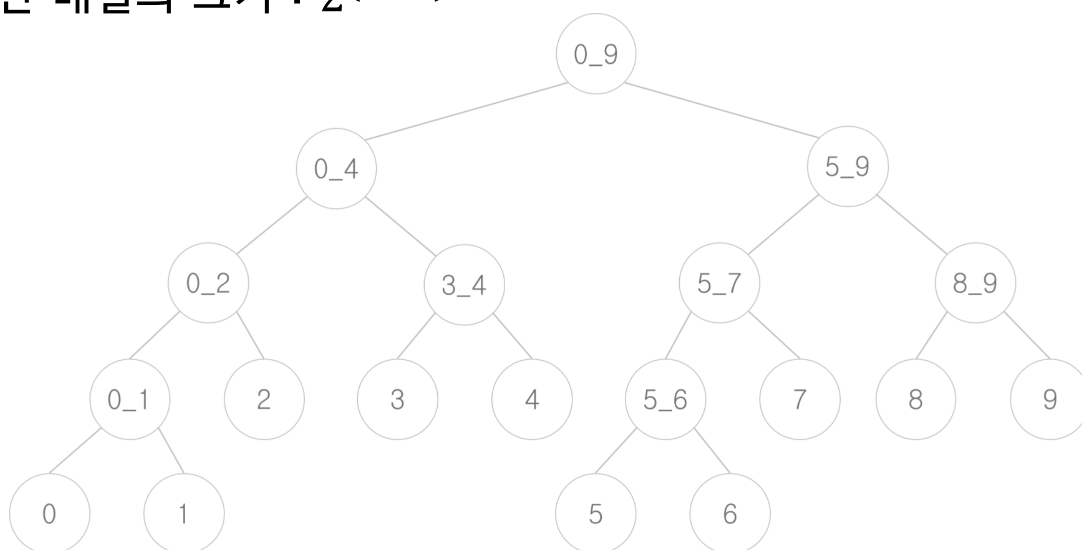
$m = (s + e) / 2$



세그먼트 트리

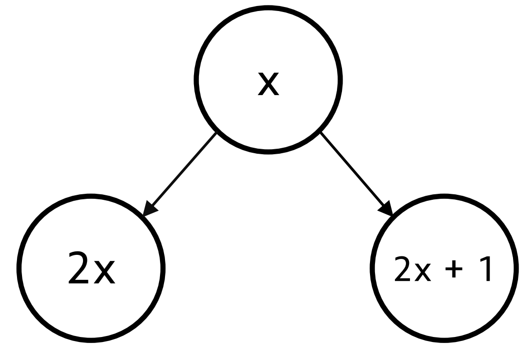
Range Minimum Query

- 각 노드는 구간을 나타냅니다.
- 0 ~ 9는 0 ~9 까지의 최소값을 저장하고 있습니다. 0 ~ 1은 0 ~ 1까지의 최솟값
- 노드는 3가지 인수를 통하여 정보를 저장합니다.
  ① 저장된 칸의 번호
  ② 구간 : 시작
  ③ 구간 : 끝
- N이 2의 제곱 꼴인 경우 Full Binary Tree 이므로 높이 H = $[lgN]$ 입니다.
  이 때 필요한 배열의 크기 : $2^{(H+1)}$



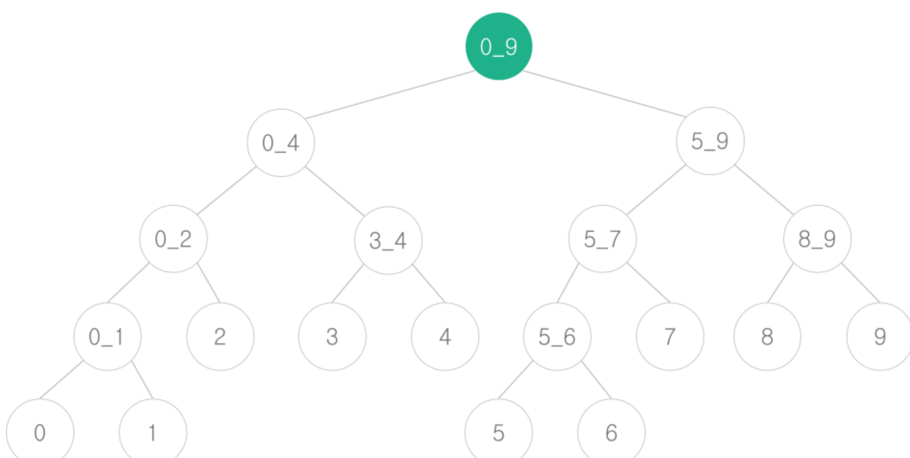
세그먼트 트리

Range Minimum Query



- start == end 인 경우에는 리프 노드
- node의 왼쪽 자식 : 2*node, 오른쪽 자식 : 2*node + 1
- 어떤 노드가 [start, end]를 담당한다면
  왼쪽 자식 : [start, (start + end)/2], 오른쪽 자식 : [(start + end)/2 +1, end]
- ※ 노드의 범위가 쿼리의 범위에 완전히 포함? 완전히 미포함 되는가?
  ① 노드의 범위[start, end]가 쿼리의 [i, j] 범위의 부분집합 이라면 구간의 최솟값을 리턴
  ② 노드의 범위[start, end]가 쿼리의 [i, j] 범위와 배타적 이라면 불가능한 값 리턴
  ③ 그 외의 경우 왼쪽/오른쪽 노드를 나눈 후 분할하여 구간의 최솟값 확인

- 0 ~ 9 최솟값 구하기 : root node 구간에 완전히 포함되어 최솟값 리턴

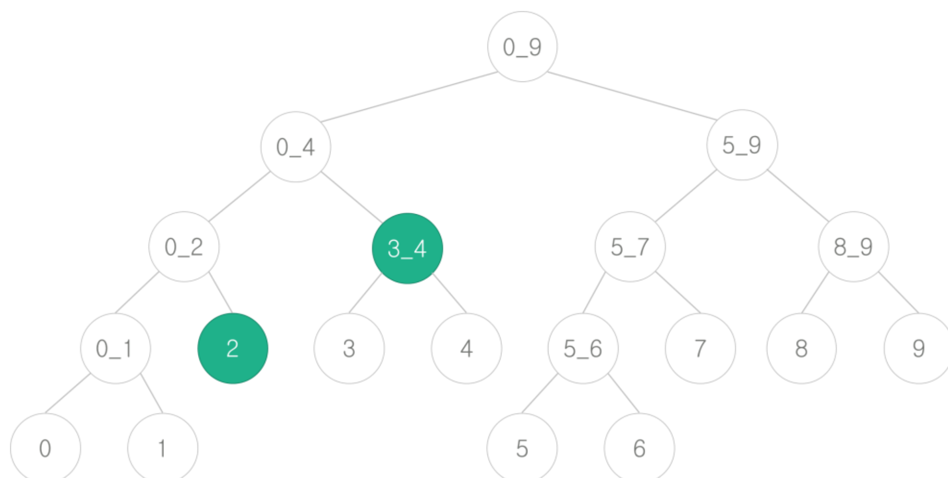


Gaussian

세그먼트 트리

Range Minimum Query

- 구간 [2, 4] 범위의 최솟값 구하기
  ① 노드 [0, 9]가 쿼리 [2, 4]에 부분집합/배타적 경우가 아니므로 왼쪽/오른쪽 자식으로 분할 한다.
  ② 노드 [5, 9]는 쿼리 [2, 4]에 배타적이므로 불가능한 값 리턴
  ③ 노드 [0, 4]는 쿼리 [2, 4]에 부분집합/배타적 경우가 아니므로 왼쪽/오른쪽 자식으로 분할 한다.
  ④ 노드 [3, 4]는 쿼리 [2, 4]에 부분집합 이므로 구간 [3, 4]의 최솟값을 리턴 한다.
  ⑤ 노드 [2]는 쿼리 [2, 4]에 부분집합 이므로 구간 [2]의 최솟값을 리턴 한다.
  ⑥ 노드 [0, 1]은 쿼리 [2, 4]에 배타적이므로 불가능한 값 리턴



세그먼트 트리

Range Minimum Query

```
vi A, tree;
void init(int node, int start, int end) {
        if (start == end) {
                tree[node] = A[start];
        }
        else {
                init(2*node, start, (start + end) / 2);
                init(2*node + 1, (start + end) / 2 + 1, end);
                tree[node] = min(tree[2*node], tree[2*node + 1]);
        }
}
```

세그먼트 트리 초기화

```
int query(int node, int start, int end, int i, int j) {
        if (j < start || end < i) return -1;
        if (i <= start && end <= j) return tree[node];

        int m1 = query(2 * node, start, (start + end) / 2, i, j);
        int m2 = query(2 * node + 1, (start + end) / 2 + 1, end, i, j);
        if (m1 == -1) return m2;
        else if (m2 == -1) return m1;
        else return min(m1, m2);
}
```

Minimum Query 함수

세그먼트 트리

Range Minimum Query

```
void update(int node, int start, int end, int i, int new_val) {
        if (i < start || end < i) return;
        tree[node] = min(tree[node], new_val);
        if (start != end) {
                update(2 * node, start, (start + end) / 2, i, new_val);
                update(2 * node + 1, (start + end) / 2 + 1, end, i, new_val);
        }
}
```

세그먼트 트리 업데이트

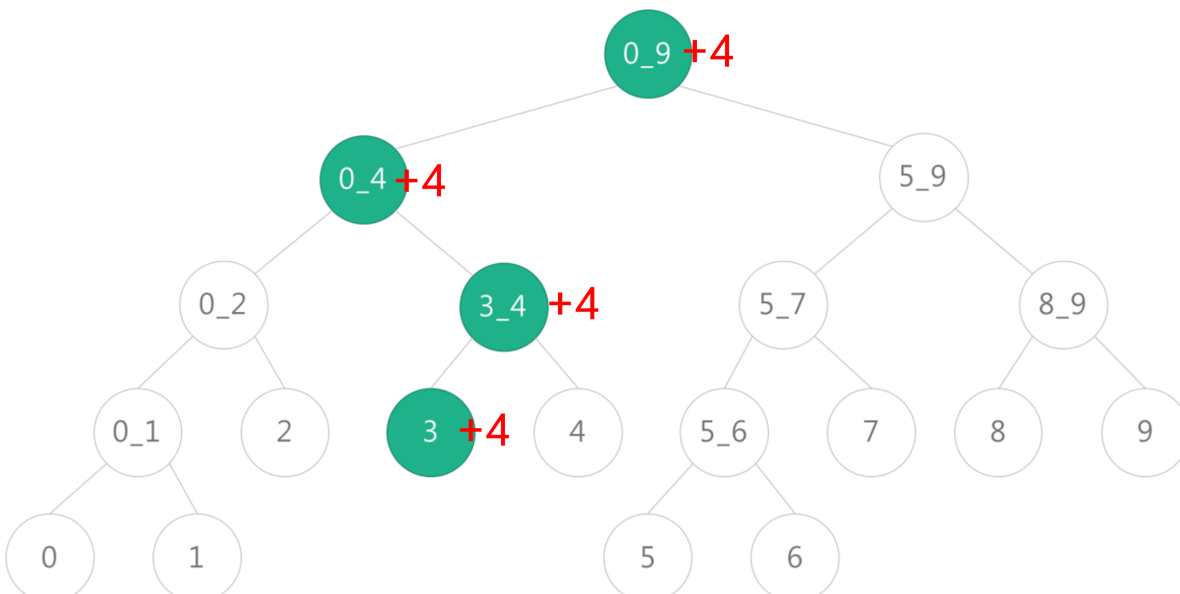
최소값

https://www.acmicpc.net/problem/10868

세그먼트 트리

Range Sum Query

- 구간의 합을 구할 때 누적 합을 이용하면 O(1)에 구할 수 있지만 수정이 필요한 경우 누적 합은 O(N)의 시간이 드는 반면 세그먼트 트리는 O(lgN)에 수정이 가능하다.
- 구간의 최솟값이 아니고 합을 구하는 경우 min 대신 +를 하면 되고 결과를 return으로 바꾼다.
- 구간의 합을 구하는 경우 중간에 수 변경이 필요할 때 세그먼트 트리를 사용한다.
- 3 → 7로 변경할 때 : 변경 값(+4) 만큼 더해준다.



Gaussian

세그먼트 트리

Range Sum Query

- RMQ와 비교 시 수정 된 코드 확인

```
ll init(int node, int start, int end) {
	if (start == end) {
		return tree[node] = A[start];
	}
	else {
		ll m1 = init(2 * node, start, (start + end) / 2);
		ll m2 = init(2*node + 1, (start + end) / 2 + 1, end);
		return tree[node] = m1 + m2;
	}
}
```

세그먼트 트리 초기화

```
ll query(int node, int start, int end, int i, int j) {
	if (j < start || end < i) return 0;
	if (i <= start && end <= j) return tree[node];

	ll m1 = query(2 * node, start, (start + end) / 2, i, j);
	ll m2 = query(2 * node + 1, (start + end) / 2 + 1, end, i, j);
	return m1 + m2;
}
```

구간 합 Query

Gaussian

세그먼트 트리

Range Sum Query

- update 함수 사용 시 기존 배열과 수정값의 차이를 매개변수로 넘겨 트리 update 수행

```
void update(int node, int start, int end, int i, ll diff) {
        if (i < start || end < i) return;
        tree[node] = tree[node] + diff;
        if (start != end) {
                update(2 * node, start, (start + end) / 2, i, diff);
                update(2 * node + 1, (start + end) / 2 + 1, end, i, diff);
        }
}
```

구간 합 구하기

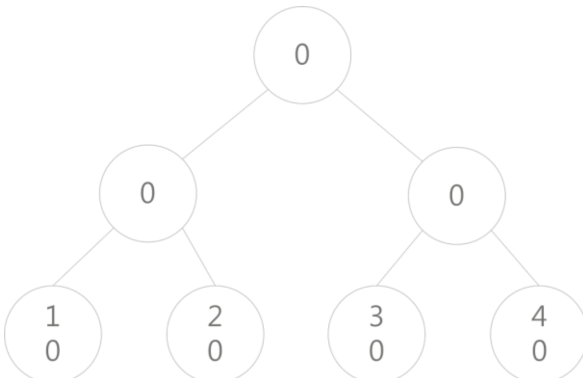
https://www.acmicpc.net/problem/2042

- 세그먼트 트리를 사용하여 구간 합을 구한다.

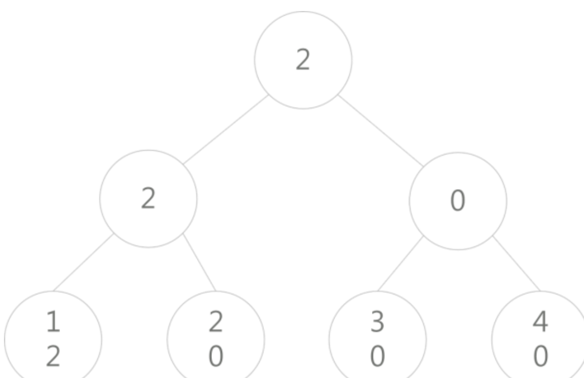
세그먼트 트리 – K번째 수 구하기

Segment Tree

- 세그먼트 트리의 초기 상태가 다음과 같다고 가정해 본다.



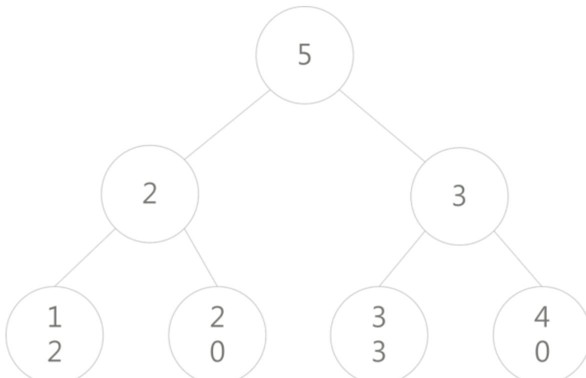
- 1에 +2를 한다.



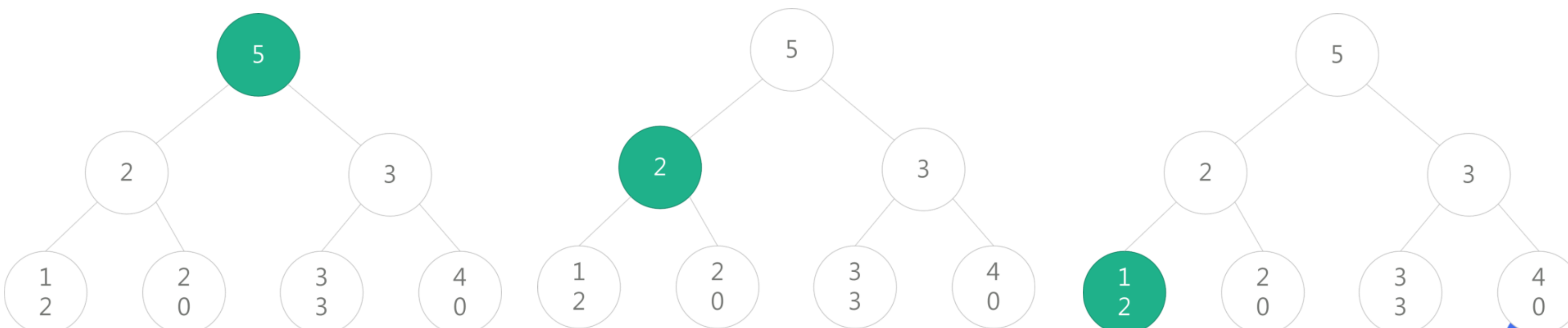
세그먼트 트리 – K번째 수 구하기

Segment Tree

- 3에 +3을 한다.



- 2번째 값을 -1을 한다 : 2번째 값을 찾아야 한다.



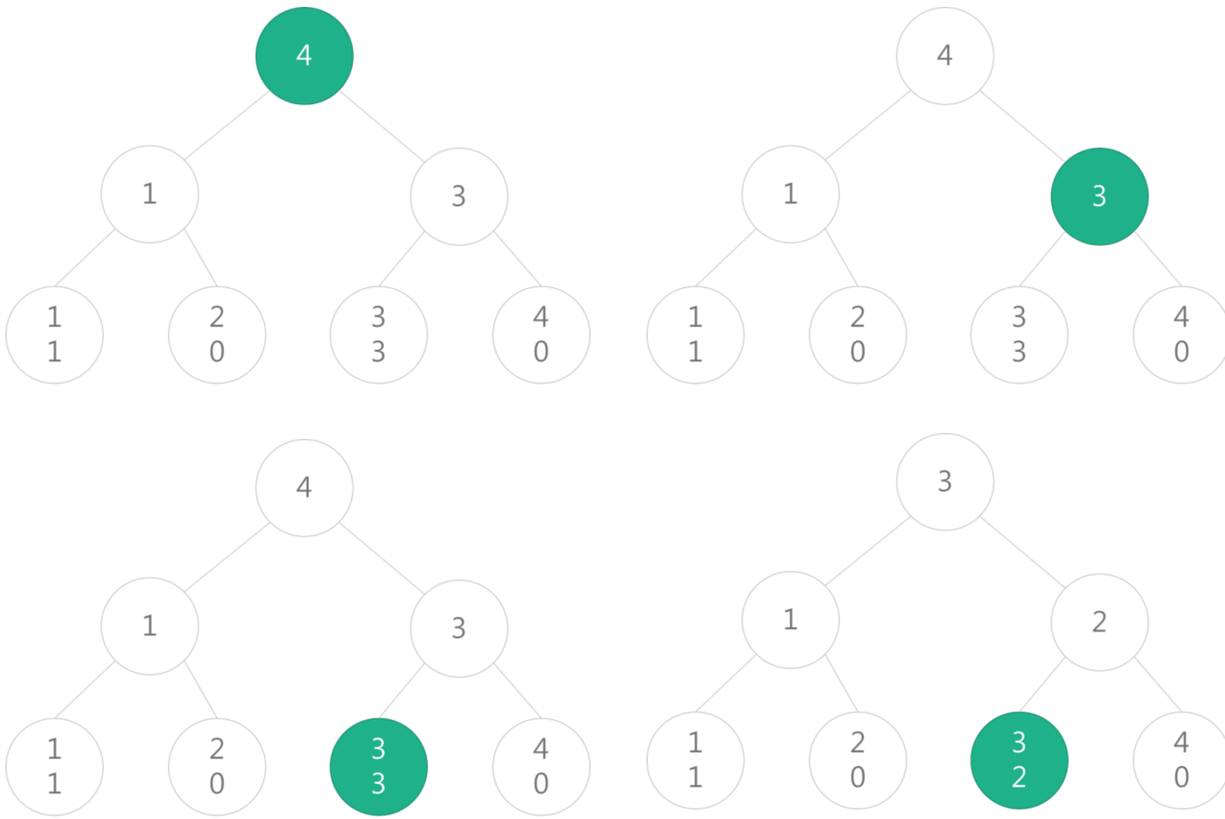
세그먼트 트리 – K번째 수 구하기

Segment Tree

- 다시 2번째 값을 -1한다.

  2번째 값은 오른쪽에 있다. 따라서 왼쪽 자식 트리의 값 만큼 빼주어야 한다.

  오른쪽 자식 트리에서 2 – tree[2*node] = 2 – tree[2] = 2 – 1 = 1 번째 값을 찾으면 된다.



세그먼트 트리 – K번째 수 구하기

Segment Tree

```
void update(vi& tree, int node, int start, int end, int i, int diff) {
        if (i < start || i > end) return;
        tree[node] += diff;
        if (start != end) {
                update(tree, 2 * node, start, (start + end) / 2, i, diff);
                update(tree, 2 * node + 1, (start + end) / 2 + 1, end, i, diff);
        }
}

int kth(vi& tree, int node, int start, int end, int k) {
        if (start == end) {
                return start;
        }
        else {
                if (k <= tree[2 * node]) {
                        return kth(tree, 2 * node, start, (start + end) / 2, k);
                }
                else {
                        return kth(tree, 2 * node + 1, (start + end) / 2 + 1, end, k - tree[2*node]);
                }
        }
}
```

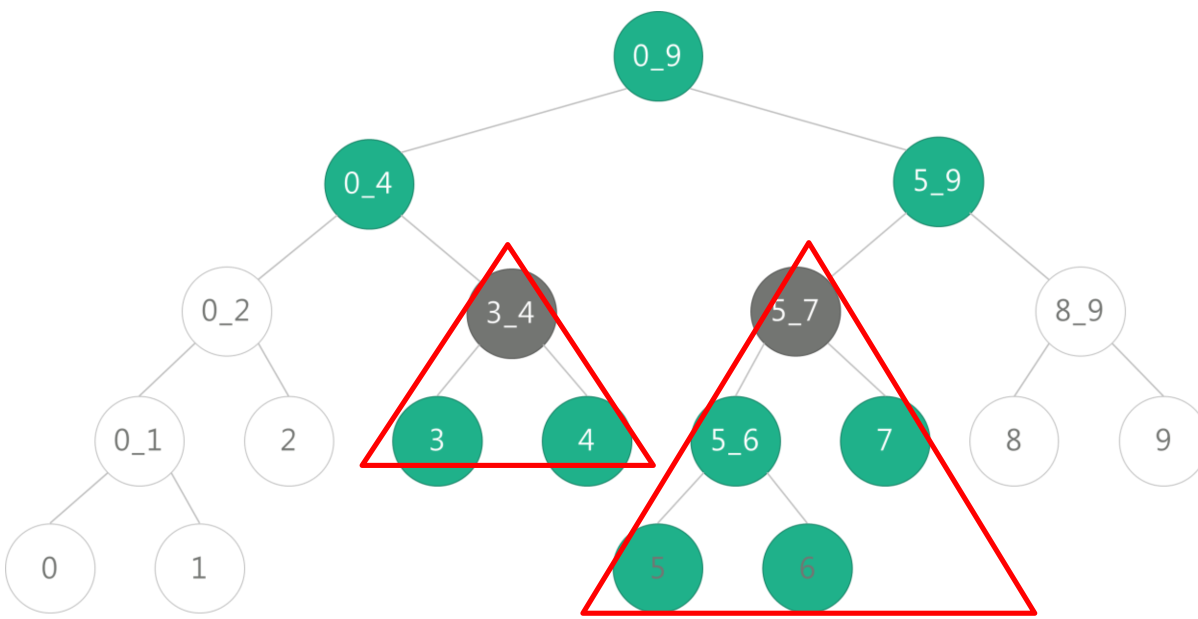
Lazy Propagation

Lazy Propagation

- 세그먼트 트리의 구간 업데이트를 효율적으로 하는 방법에 대하여 알아본다.
- 아래와 같이 구간의 업데이트가 필요할 때, 3~4 서브와 5~7의 서브는 업데이트할 필요가 없다.
- 서브 트리는 나중에 노드를 방문할 때, 업데이트를 진행하면 된다. (lazy 배열에 정보 저장)

  ex) ① 3~4를 나타내는 노드의 lazy에 10이 저장되어 있으면 3번째 수와 4번째 수에 10을 더해야 하나, 나중에 10을 더하겠다는 의미입니다.

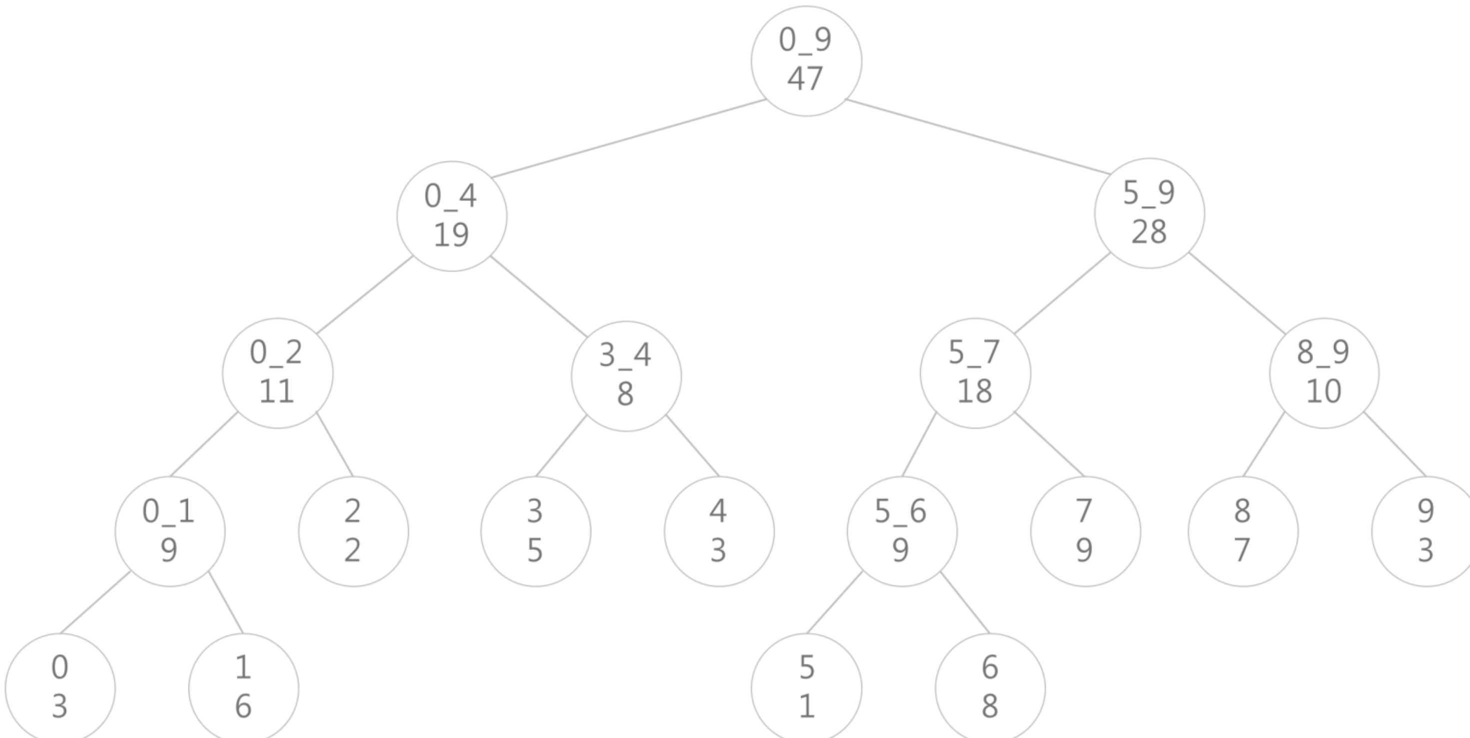
  ② 5~7을 나타내는 노드의 lazy에 20이 저장되어 있으면 5, 6, 7번째 수에 20을 더해야 하나 나중에 더하겠다는 의미 입니다.



Lazy Propagation

Lazy Propagation

- 노드에 범위 / 저장하고 있는 값이 다음과 같은 세그먼트 트리가 있다고 가정



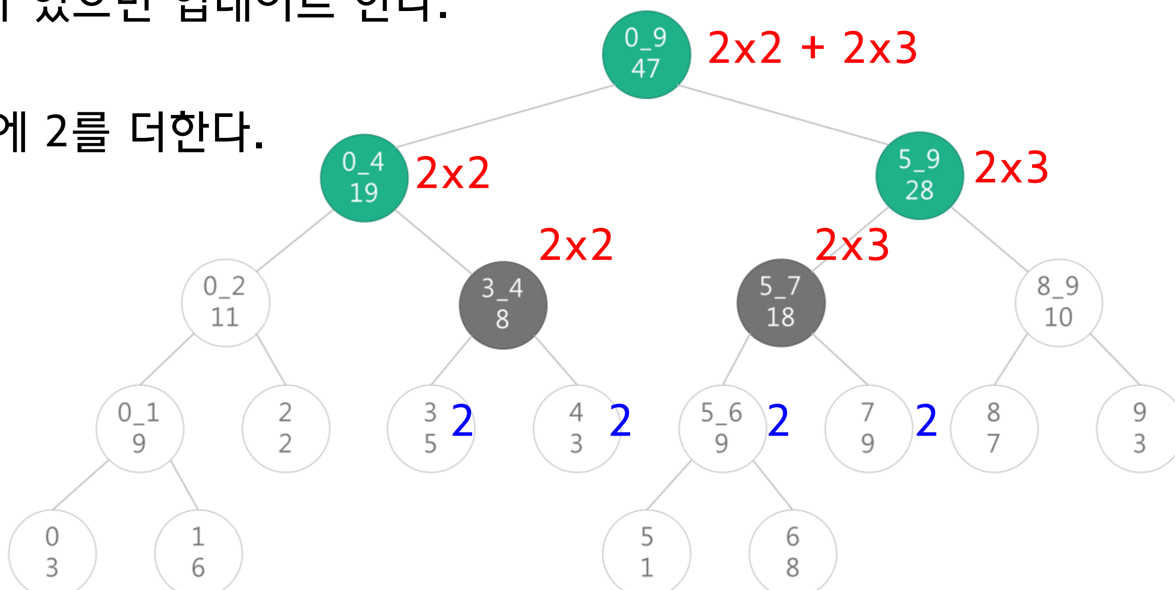
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
3	6	2	5	3	1	8	9	7	3

Lazy Propagation

Lazy Propagation

- ※ 노드의 범위가 쿼리의 범위에 완전히 포함? 완전히 미포함 되는가?
  ① 노드의 범위[start, end]가 쿼리의 [i, j] 범위의 부분집합 이라면 다음 값 만큼 노드에 더한다.
  : (update 크기) x (구간의 원소 개수)
  ② 노드의 범위[start, end]가 쿼리의 [i, j] 범위와 배타적 이라면 리턴 한다.
  ③ 그 외의 경우 왼쪽/오른쪽 노드를 나눈 후 분할하여 구간 확인
  ④ Lazy가 있으면 업데이트 한다.

3 ~ 7번째 수에 2를 더한다.



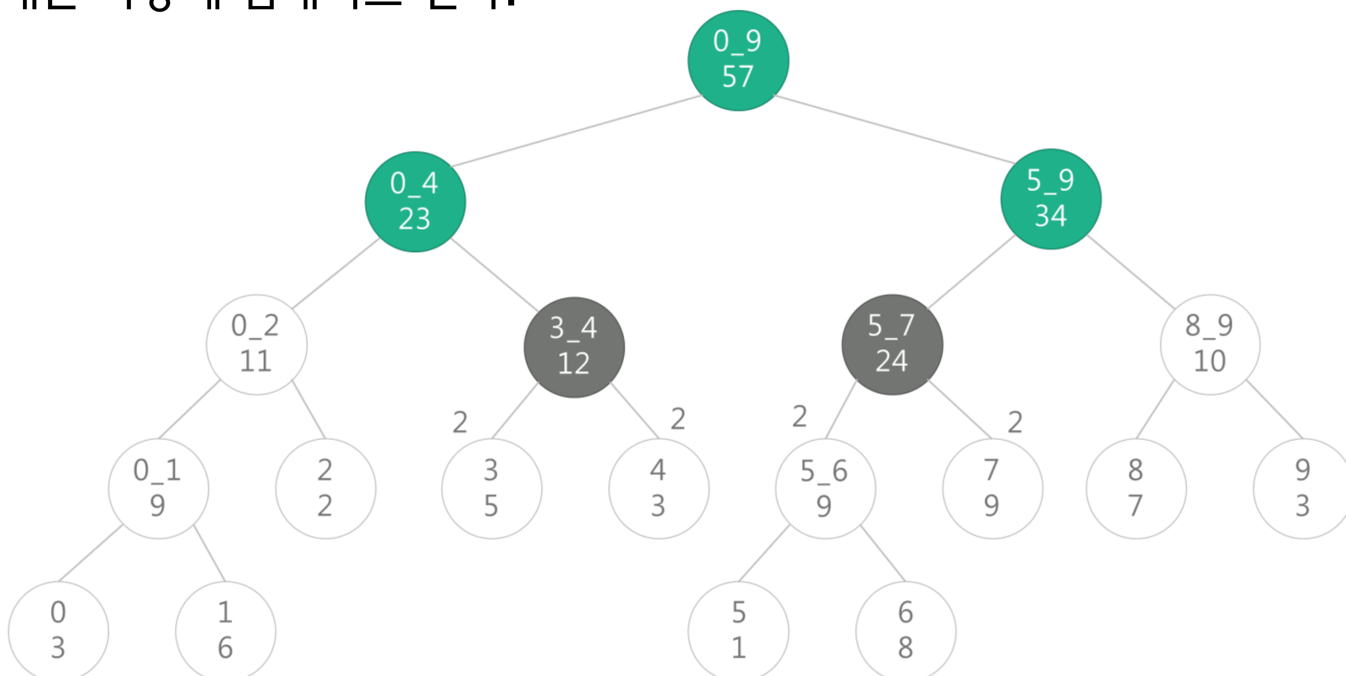
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
3	6	2	5	3	1	8	9	7	3

Lazy Propagation

Lazy Propagation

- 3 ~ 7번째 수에 2를 더한다.
- 초록색 : 3 ~ 7 중 일부 범위에 포함됨

  검정색 : 3 ~ 7 에 완전히 포함됨 → (update 크기 : 2) x (구간의 원소 개수) 만큼 더해준다.
- 검정 아래는 나중에 업데이트 한다.

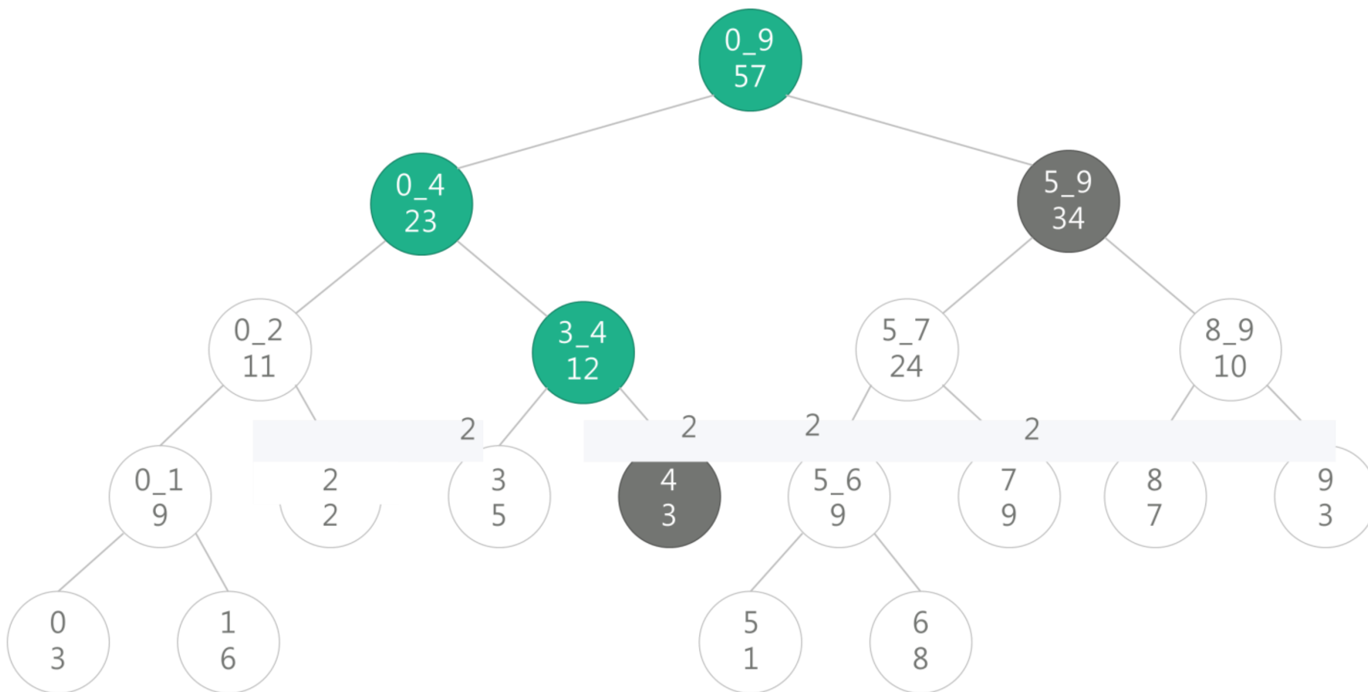


A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
3	6	2	7	5	3	10	11	7	3

Lazy Propagation

Lazy Propagation

- 4~9에 1을 더합니다.
- 이 때 해당하는 범위는 아래와 같고 검은색 노드 아래는 나중에 업데이트를 합니다.

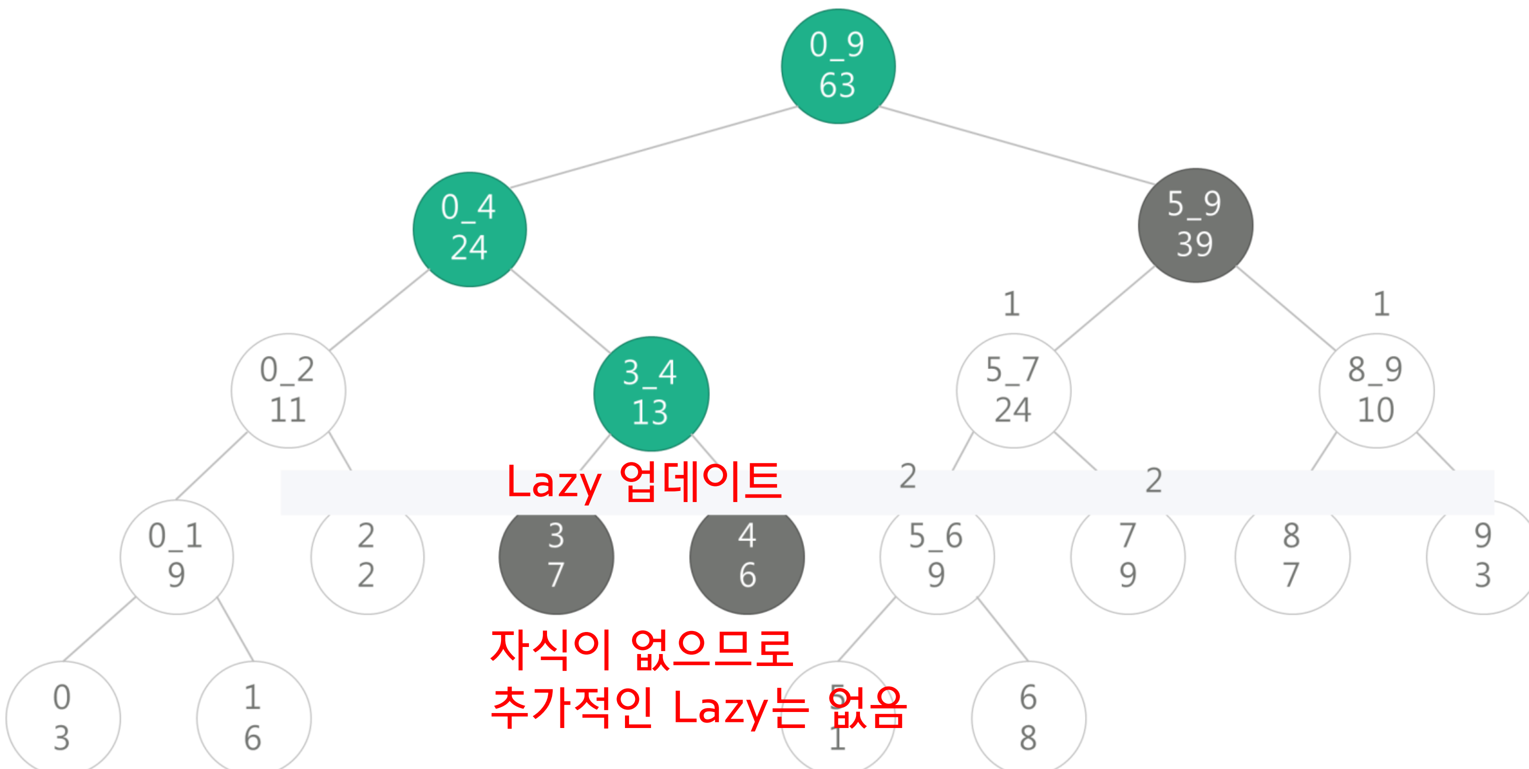


A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
3	6	2	7	5	3	10	11	7	3

Lazy Propagation

Lazy Propagation

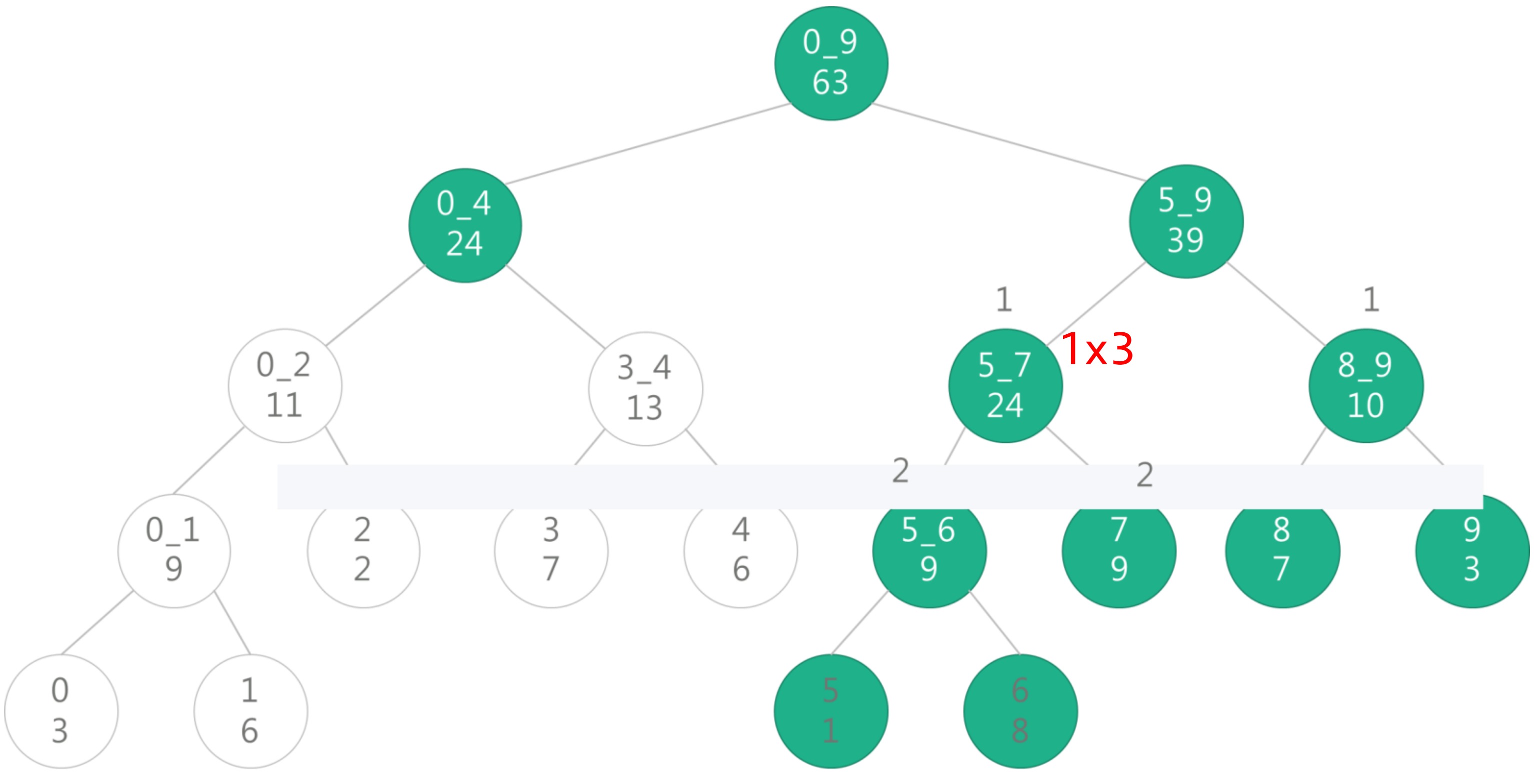
- Lazy 값이 있으면 업데이트를 한다.
- 자식이 없으면 Lazy 추가 없이 업데이트 값 조상으로 갱신한다.



Lazy Propagation

Lazy Propagation

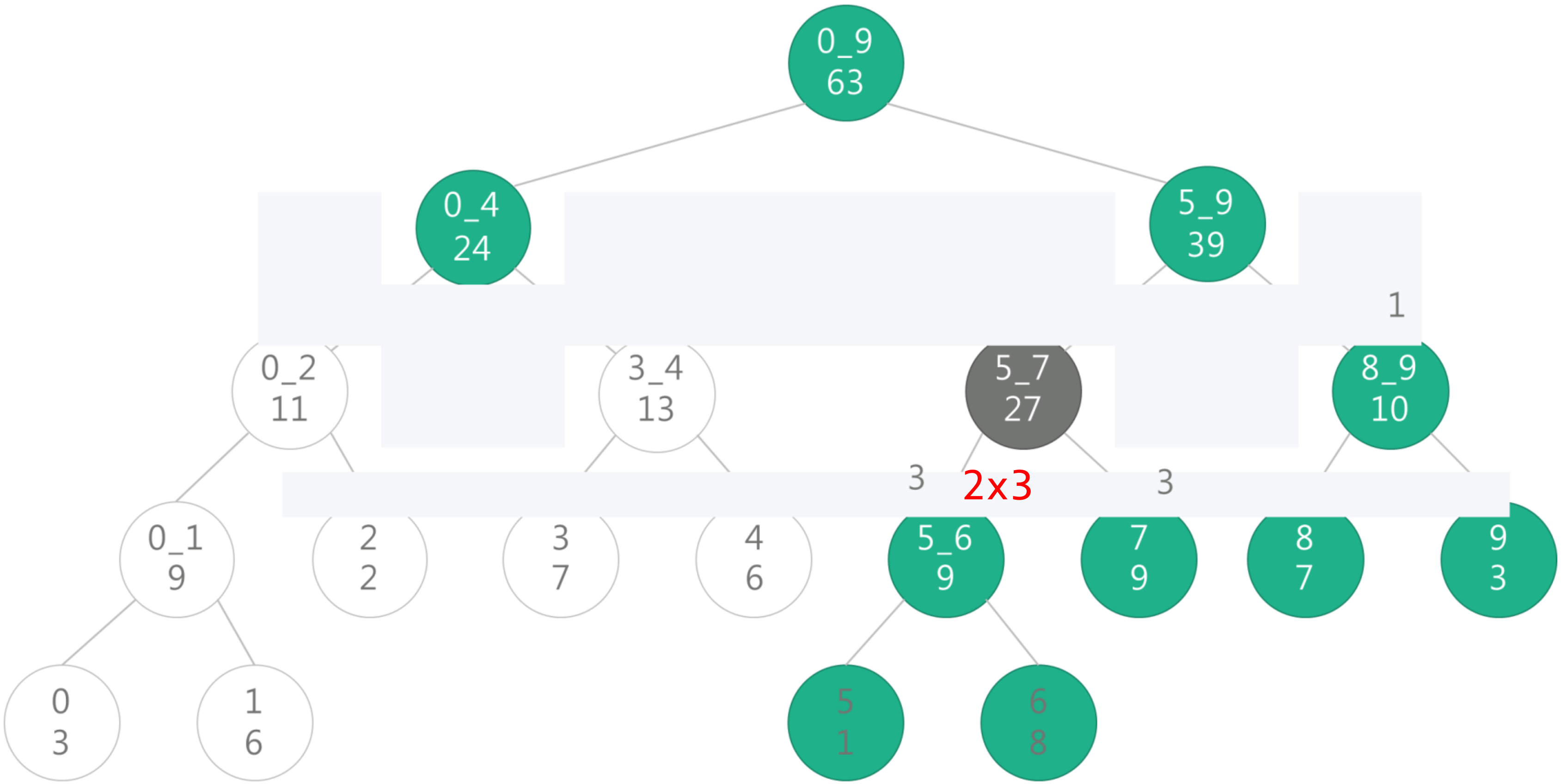
- 6 ~ 8 까지 합을 구해 본다.



Lazy Propagation

Lazy Propagation

- 5~7을 방문하였을 때 lazy가 있기 때문에 업데이트를 하고 자식에게 물려준다.

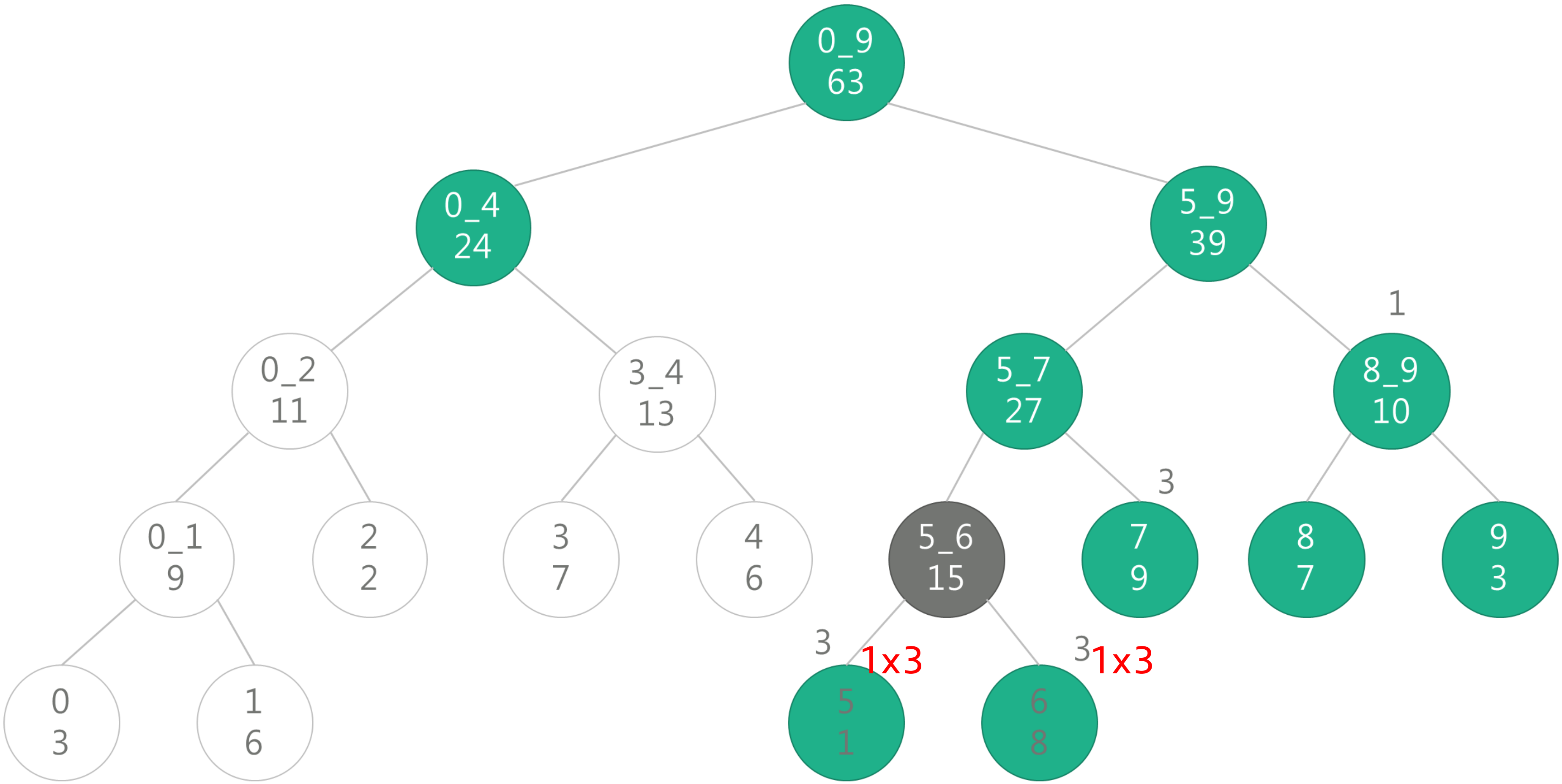


Gaussian

Lazy Propagation

Lazy Propagation

- 5~6을 방문하였을 때 lazy가 있기 때문에 업데이트를 하고 자식에게 물려준다.

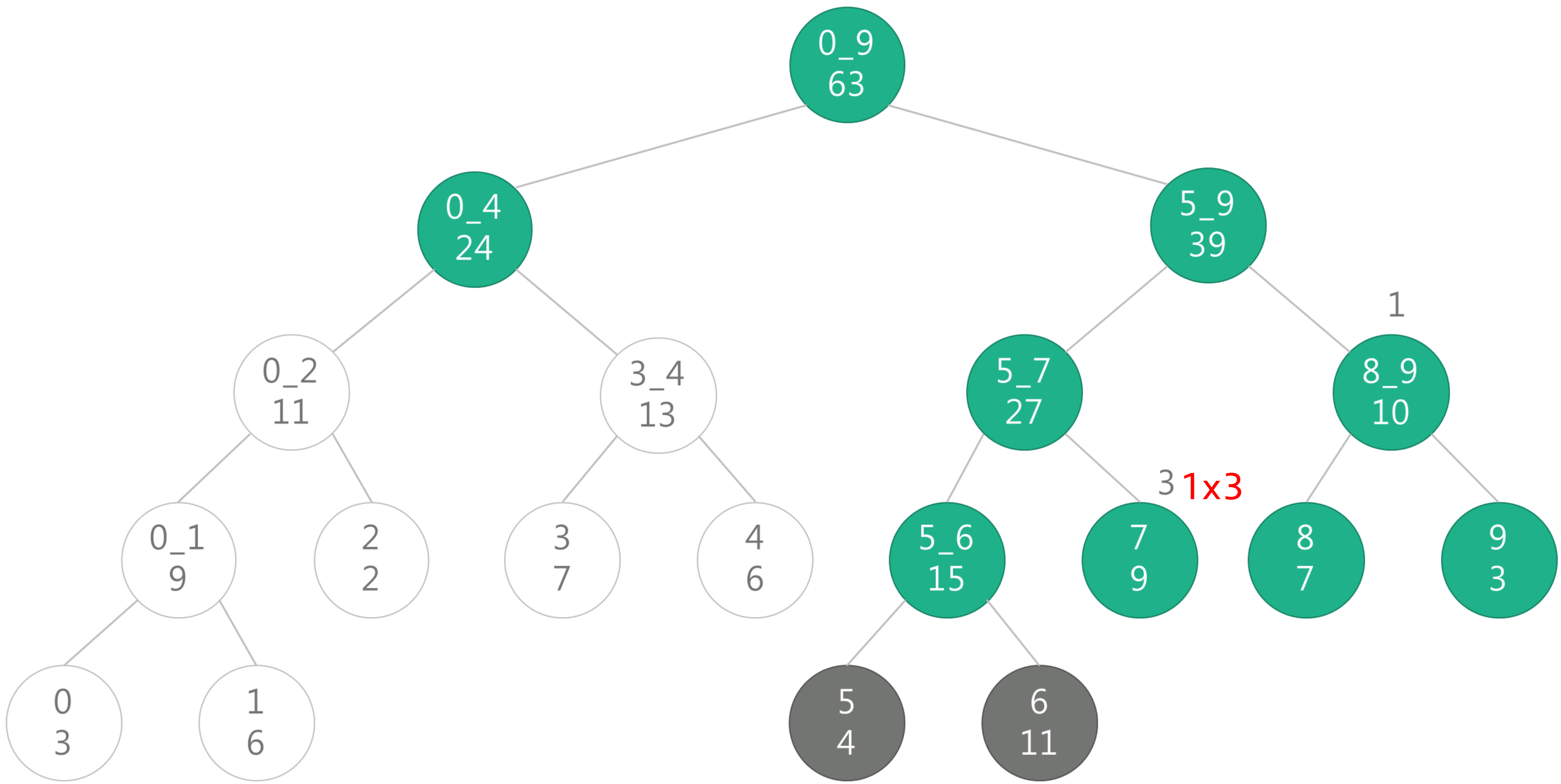


Gaussian

Lazy Propagation

Lazy Propagation

- 5와 6을 방문하였을 때 lazy가 있기 때문에 업데이트를 한다.

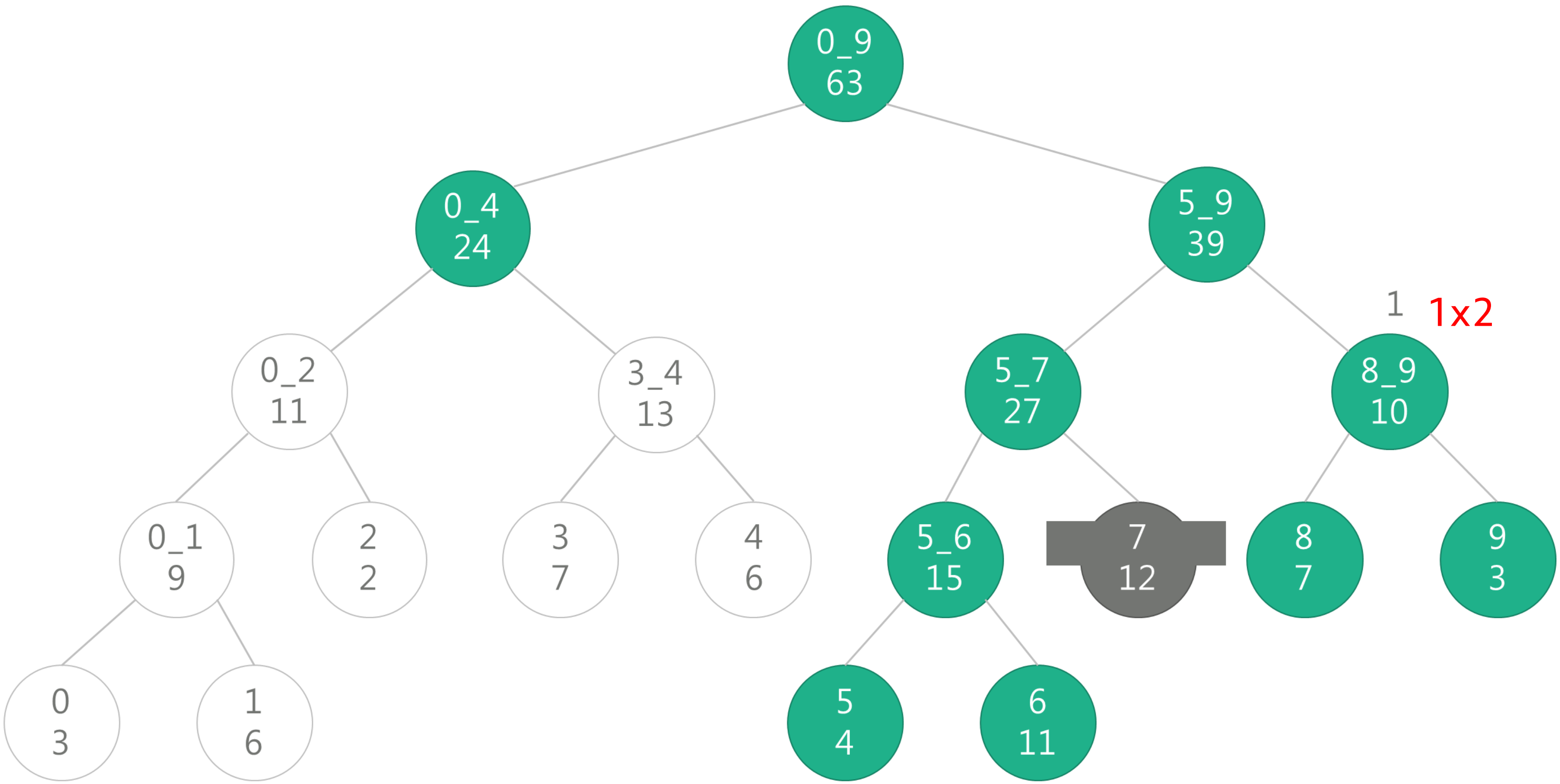


Gaussian

Lazy Propagation

Lazy Propagation

- 7을 방문하였을 때 lazy가 있기 때문에 업데이트를 한다.
- 6~8에 포함되기 때문에 리턴

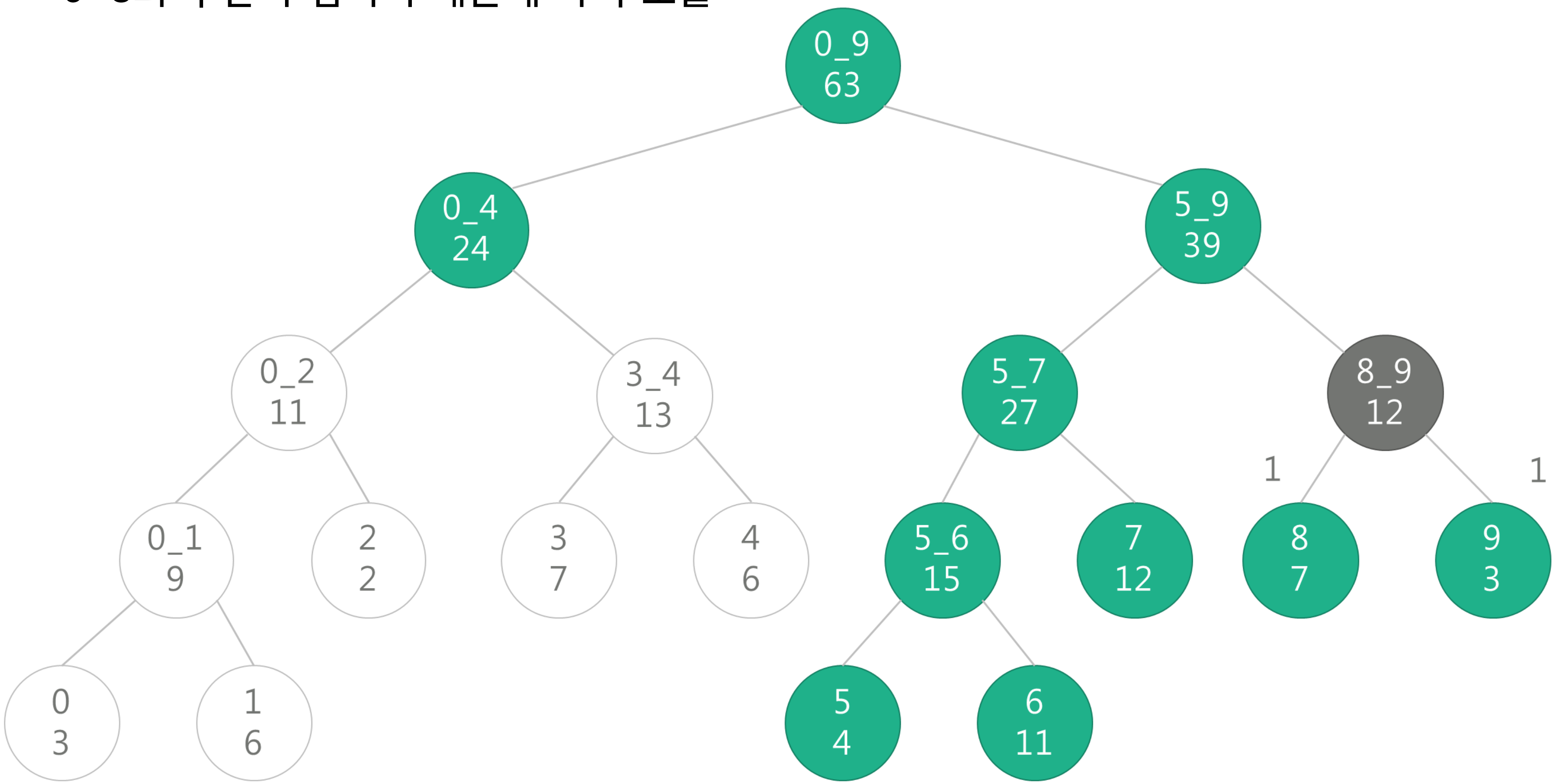


Gaussian

Lazy Propagation

Lazy Propagation

- 8~9를 방문했을 때 lazy가 있기 때문에 업데이트를 한다.
- 6~8과 구간이 겹치기 때문에 자식 호출

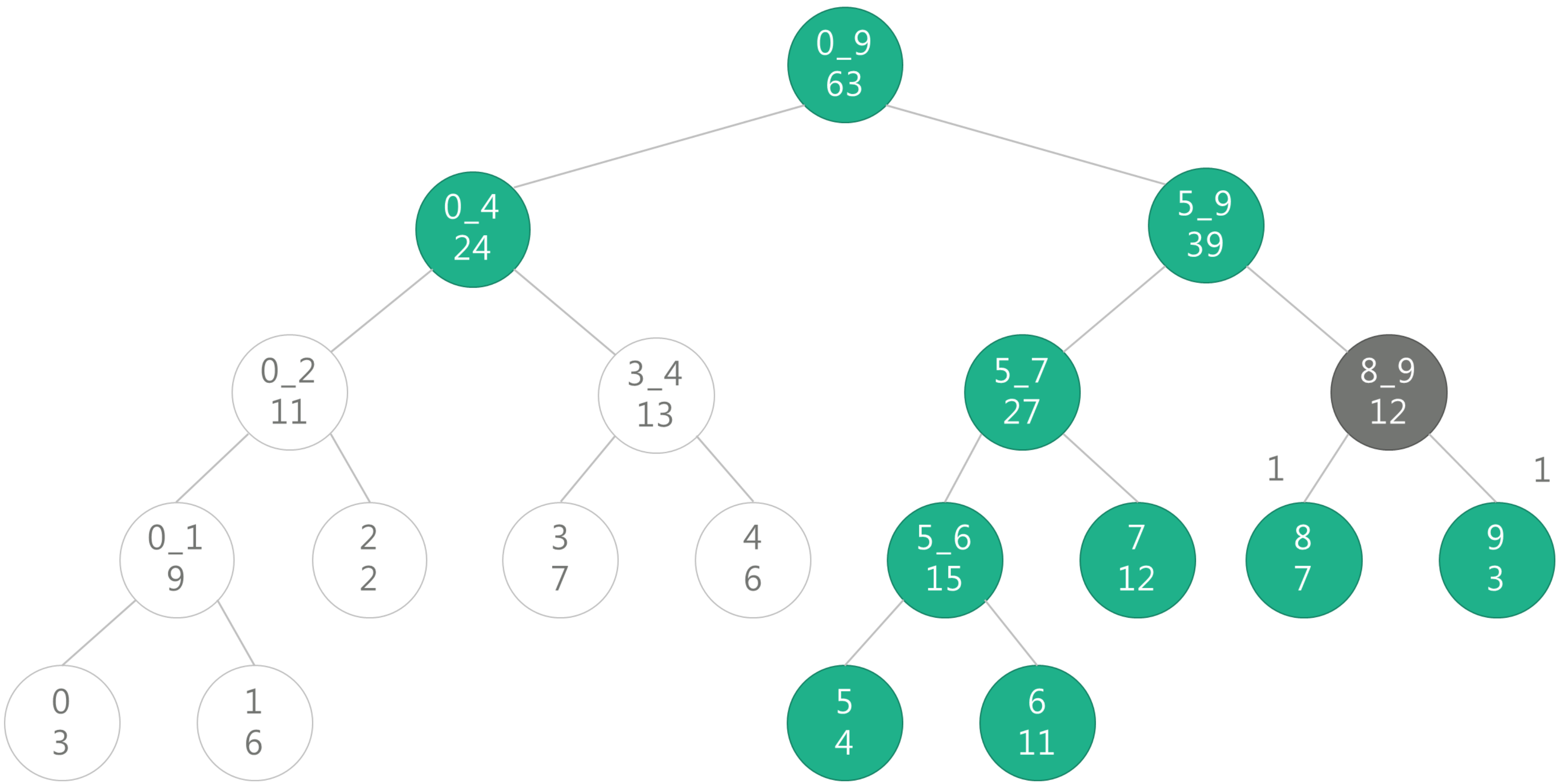


Gaussian

Lazy Propagation

Lazy Propagation

- 8은 포함되기 때문에 리턴, 9은 포함 안된다.



Gaussian

Lazy Propagation

Lazy Propagation

```
void update_lazy(int node, int start, int end) {
        if (lazy[node] != 0) {
                tree[node] += (end - start + 1)*lazy[node];
                // leaf가 아니면
                if (start != end) {
                        lazy[node * 2] += lazy[node];
                        lazy[node * 2 + 1] += lazy[node];
                }
                lazy[node] = 0;
        }
}
```