2023 algos study
고급1 week1

Lazy Segment Tree & BIT

이유진

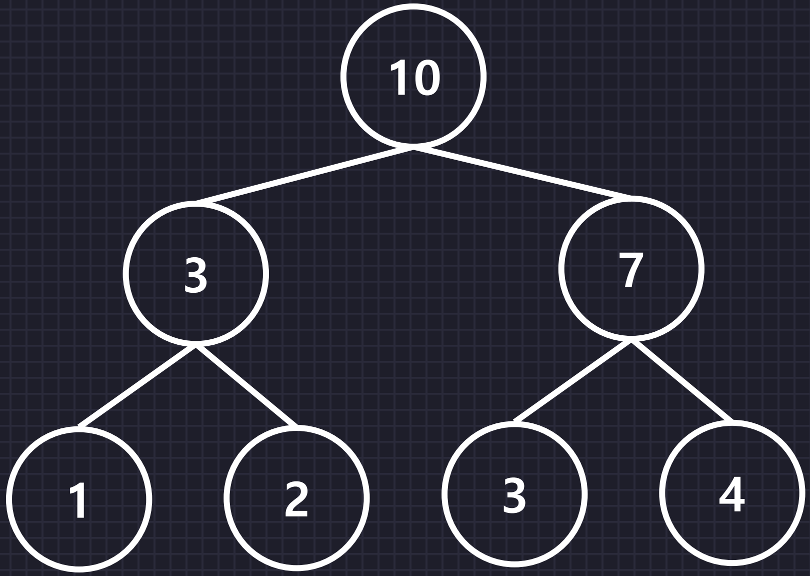
Segment Tree

- 이진 트리 형태
- 여러 개의 데이터가 존재할 때 특정 **구간의 연산**을 가장 빠르게 구할 수 있는 자료구조

1	2	3	4	5	6	7	8	9	10

데이터 변경 → O(1)

연산 → O(N)



데이터 변경 → O(logN)

연산 → O(logN)

트리의 높이 h = ceil(logN)

세그먼트 트리를 저장할 배열의 크기는 $2^{(h+1)}$면 충분

Segment Tree

- 특정 인덱스의 값 하나를 변경 → O(logN)
- 특정 구간의 값을 모두 변경? → O(NlogN)

- 연산이 조금만 바뀌면 시간이 매우 오래 걸리는 단점
- Lazy Propagation으로 해결!

Lazy Propagation

- 게으른 전파 ?
- 업데이트를 미룰 수 있을 때까지 미룸 → 업데이트 시간 최소화

Ex)

1. Index 1~5 구간의 값을 각각 +3
2. Index 6~8 구간의 값을 각각 +4
3. Index 9~10 구간의 값을 각각 +5

이 때 모든 노드들을 업데이트 해야 할까?

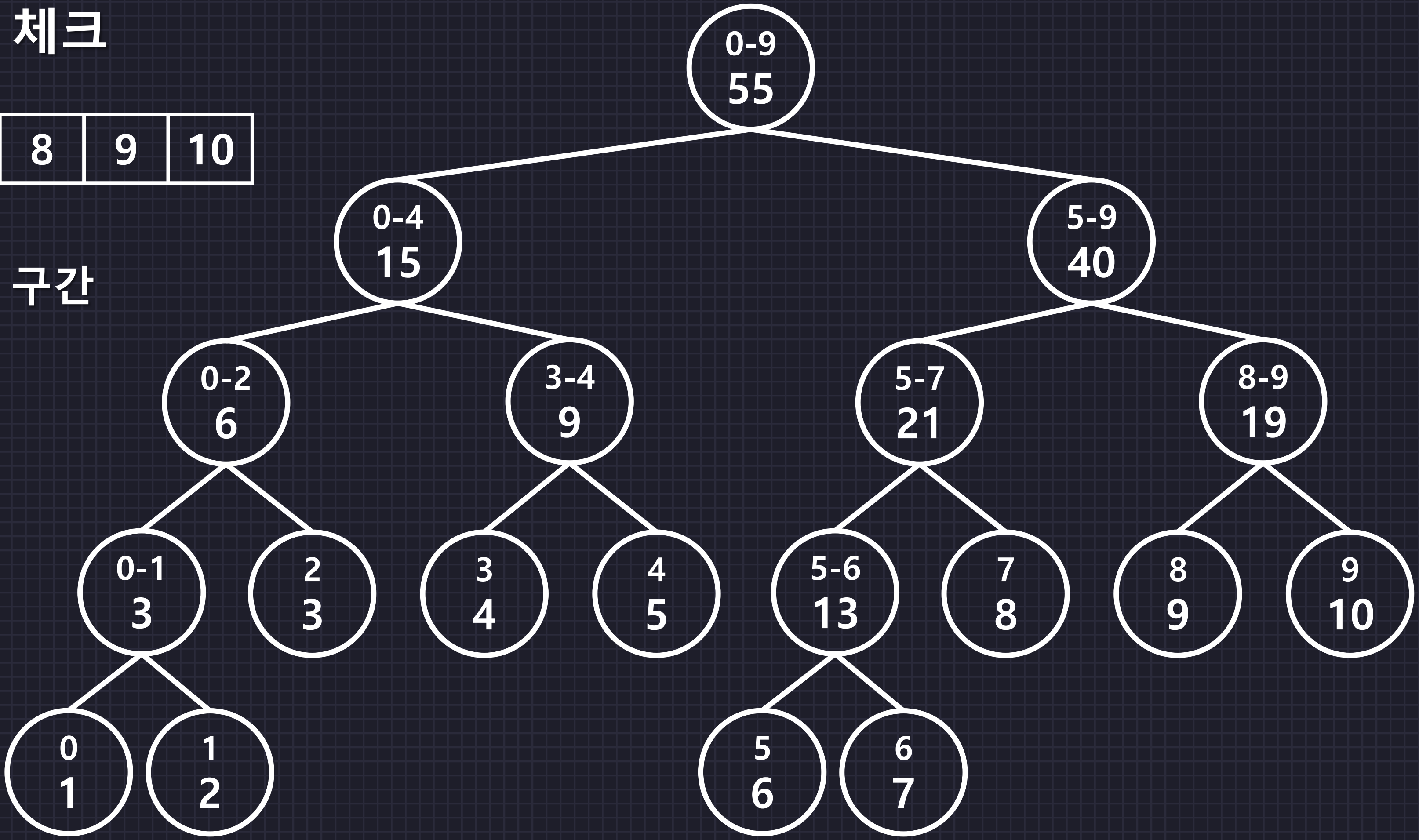
Lazy Propagation

- 세그먼트 트리와 같은 크기의 배열(혹은 벡터)이 하나 더 필요함

  → 업데이트의 필요 여부 체크

1	2	3	4	5	6	7	8	9	10

- 윗줄 : 해당 노드가 포함하는 구간
- 아랫줄 : 구간합의 결과



Lazy Propagation

구간에 대한 연산을 할 때는, 현재 우리가 탐색하는 범위가 찾고자 하는 구간과

1. 완전히 겹쳐지지 않는 경우 → 더이상의 탐색 / 업데이트 필요 x
2. 완전히 겹쳐지는 경우 → propagation 대상 !
3. 일부만 걸치는 경우 → 양 옆의 자식 노드까지 더 탐색

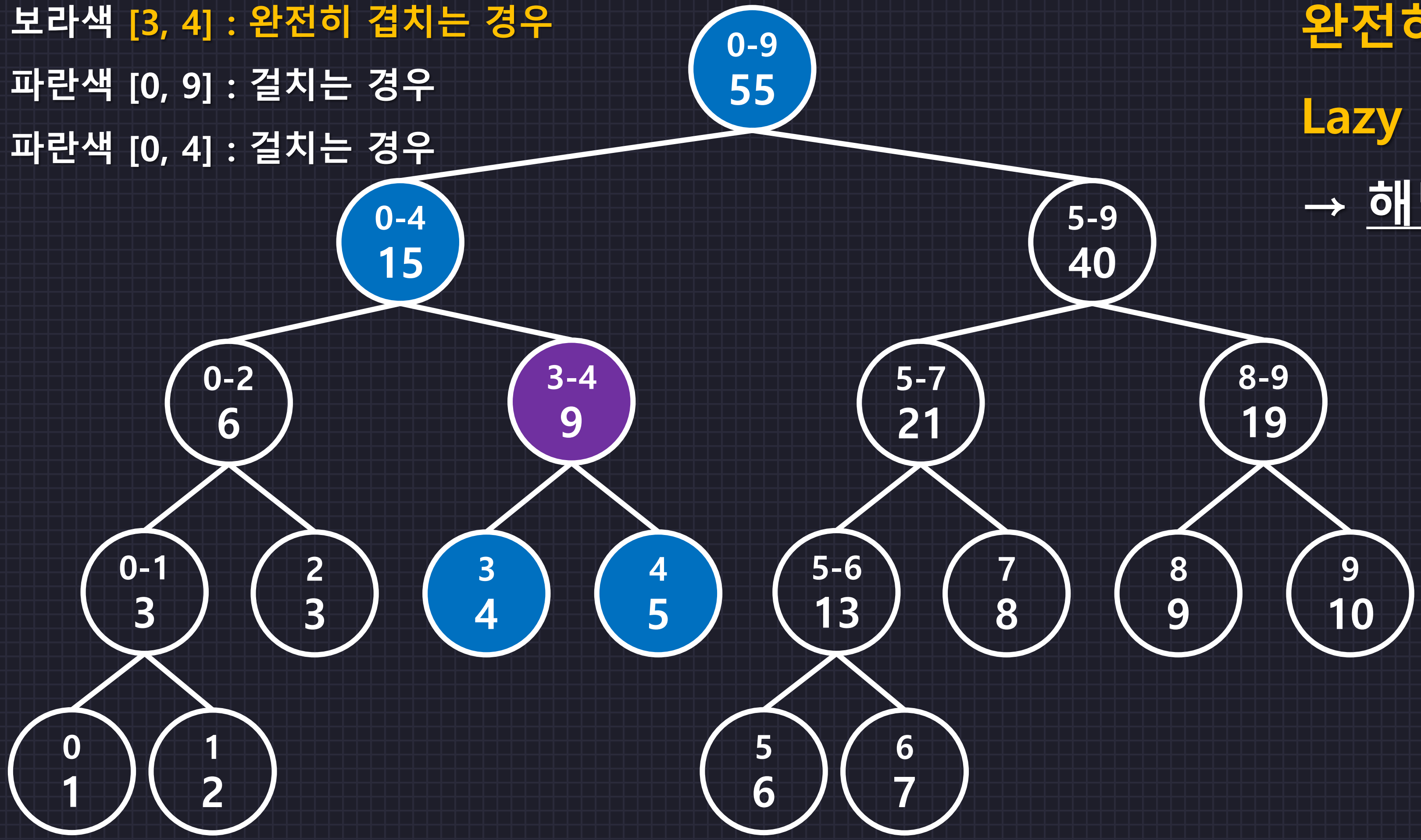
Lazy Propagation

Ex) index 3~4 구간의 값을 각각 +2

보라색 [3, 4] : 완전히 겹치는 경우

파란색 [0, 9] : 걸치는 경우

파란색 [0, 4] : 걸치는 경우



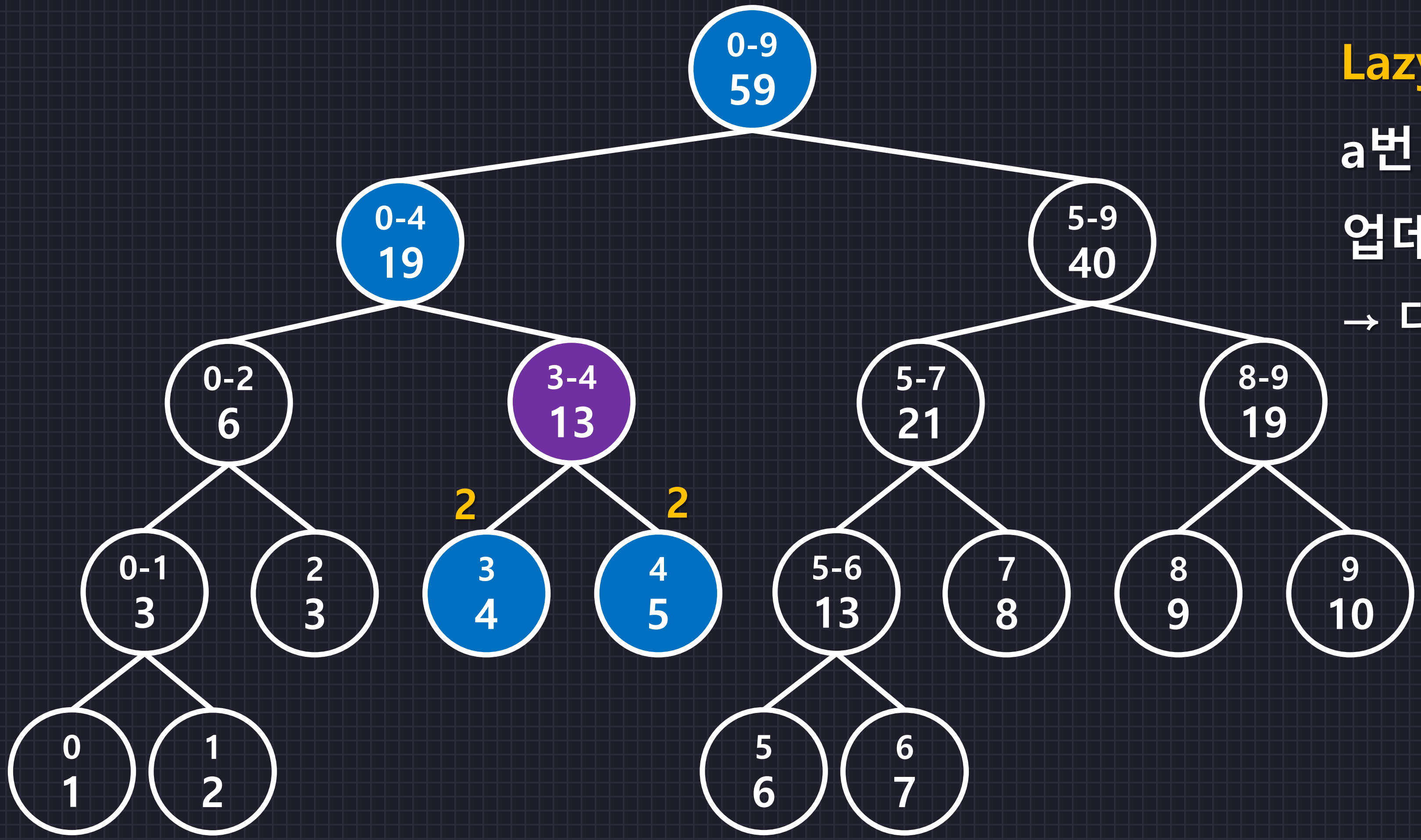
완전히 겹치는 구간 [3, 4]에 대해

Lazy propagation 진행!

→ <u>해당 노드까지만</u> 업데이트

Lazy Propagation

Ex) index 3~4 구간의 값을 각각 +2



Lazy[a] = b

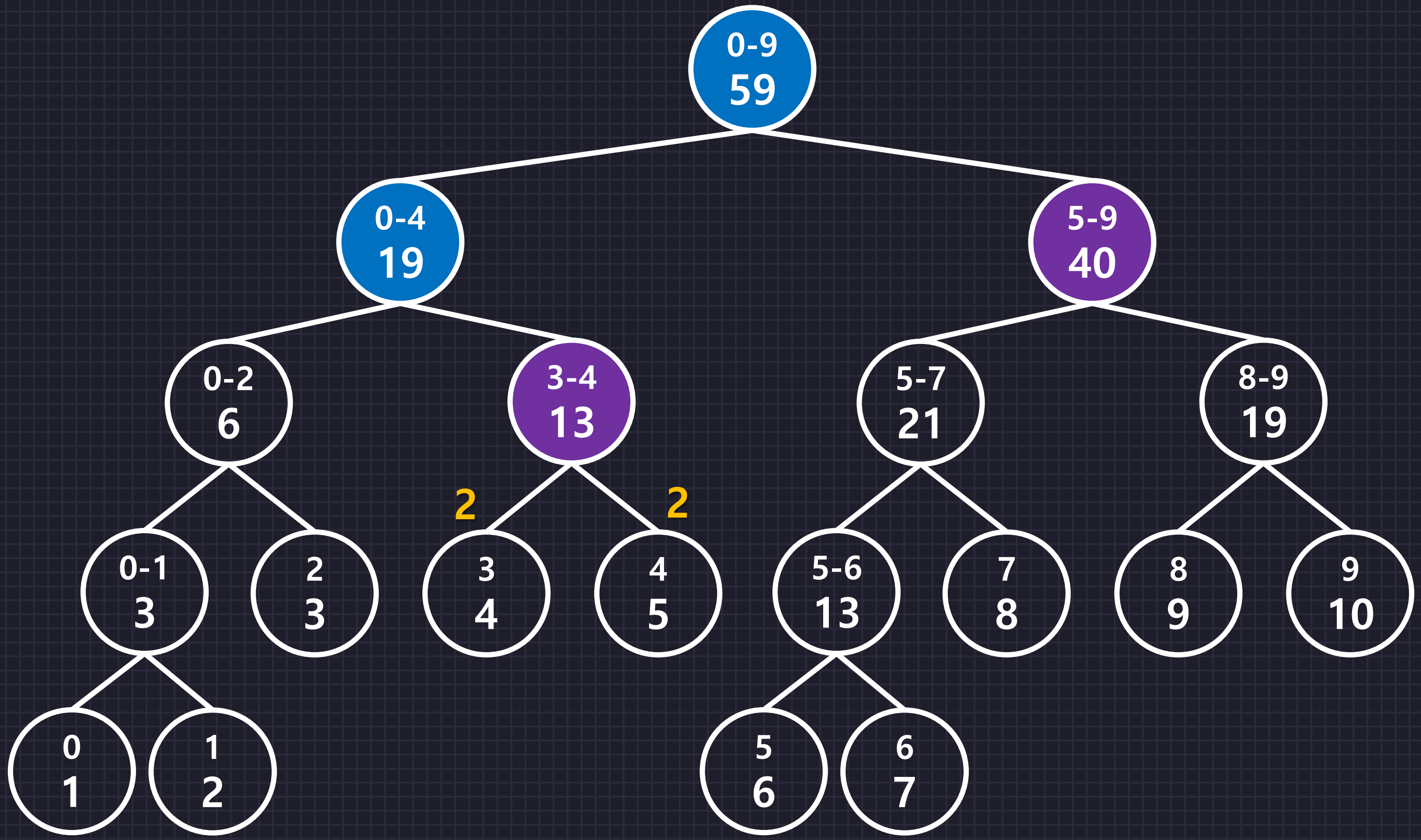
a번 노드는 사용 전 b만큼

업데이트 필요

→ 다음에 사용할 때 Lazy만 확인!

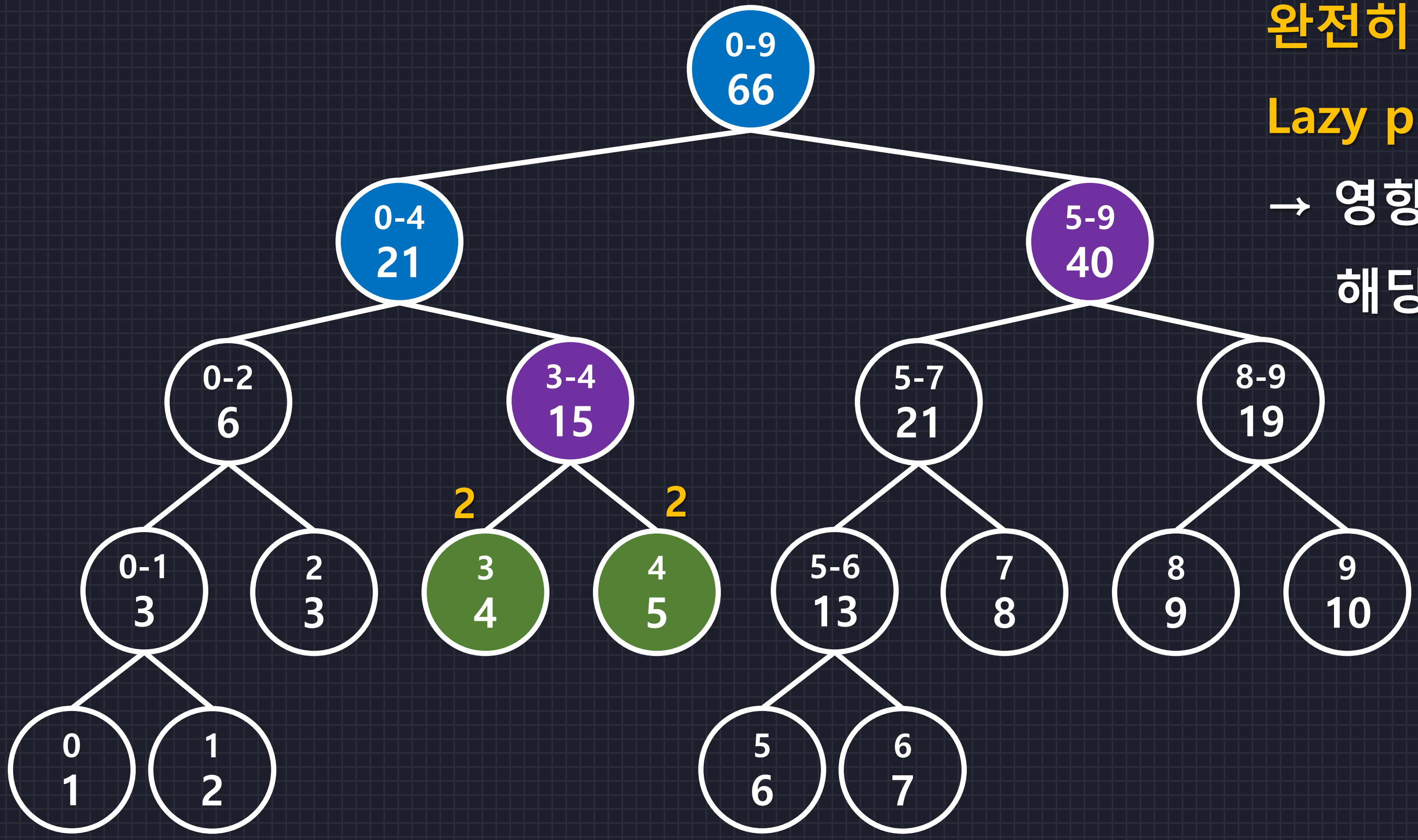
Lazy Propagation

Ex) index 3~9 구간의 값을 각각 +1



Lazy Propagation

Ex) index 3~9 구간의 값을 각각 +1



완전히 겹치는 구간 [3, 4]에 대해

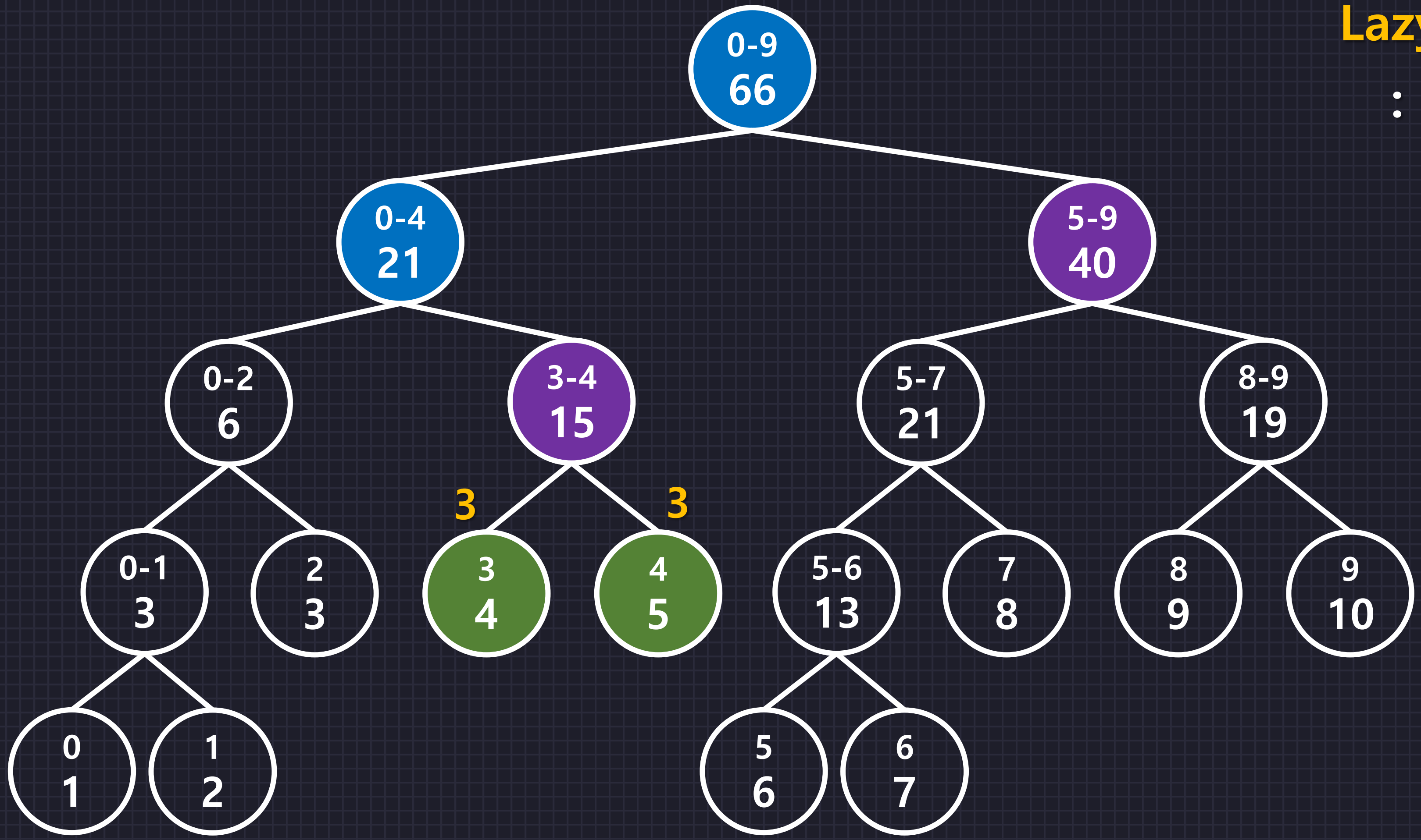
Lazy propagation 진행!

→ 영향을 받는 노드들 업데이트

해당 노드 아래로는 업데이트x

Lazy Propagation

Ex) index 3~9 구간의 값을 각각 +1

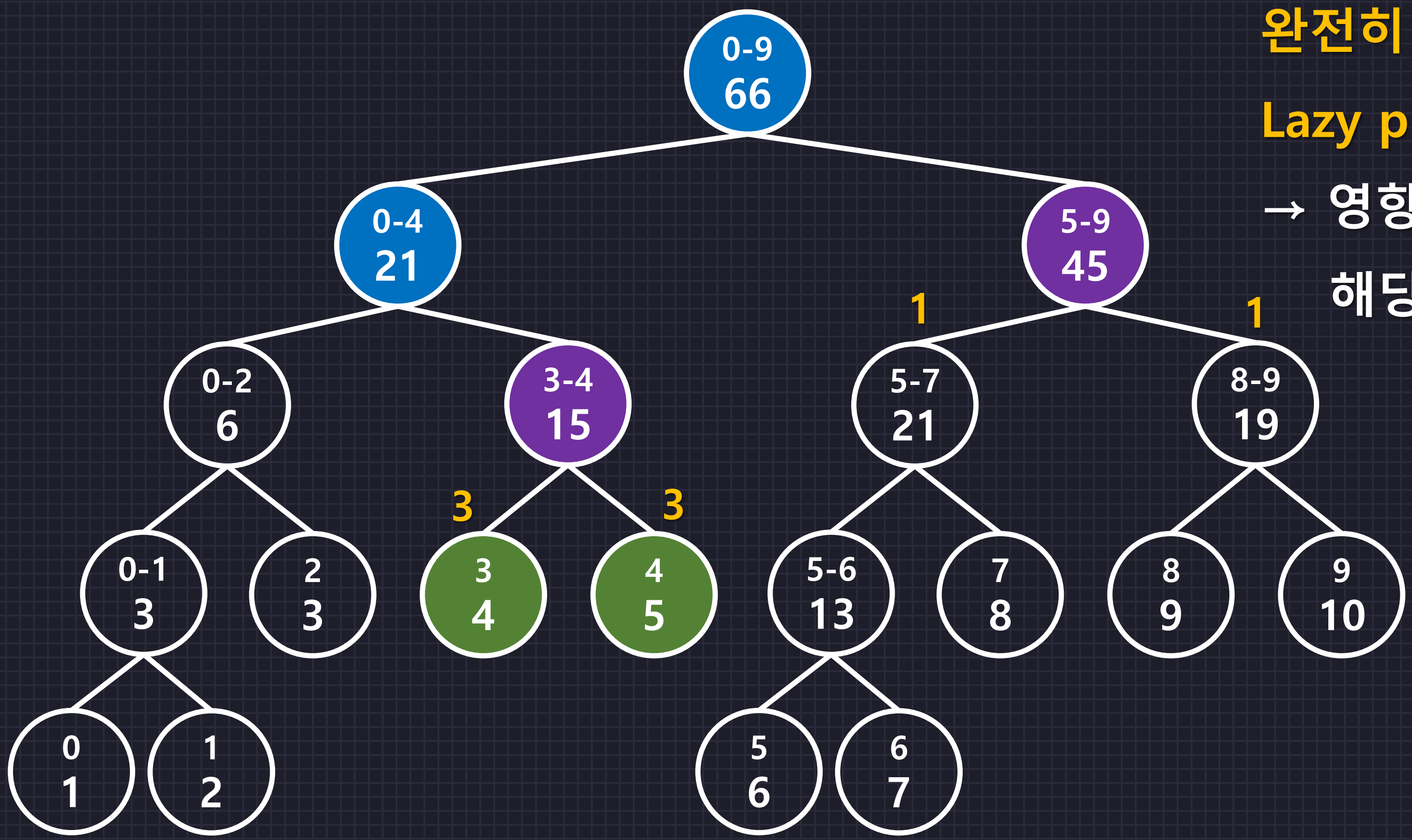


Lazy값 업데이트

: 그냥 더하면 됨!

Lazy Propagation

Ex) index 3~9 구간의 값을 각각 +1



완전히 겹치는 구간 [5, 9]에 대해

Lazy propagation 진행!

→ 영향을 받는 노드들 업데이트

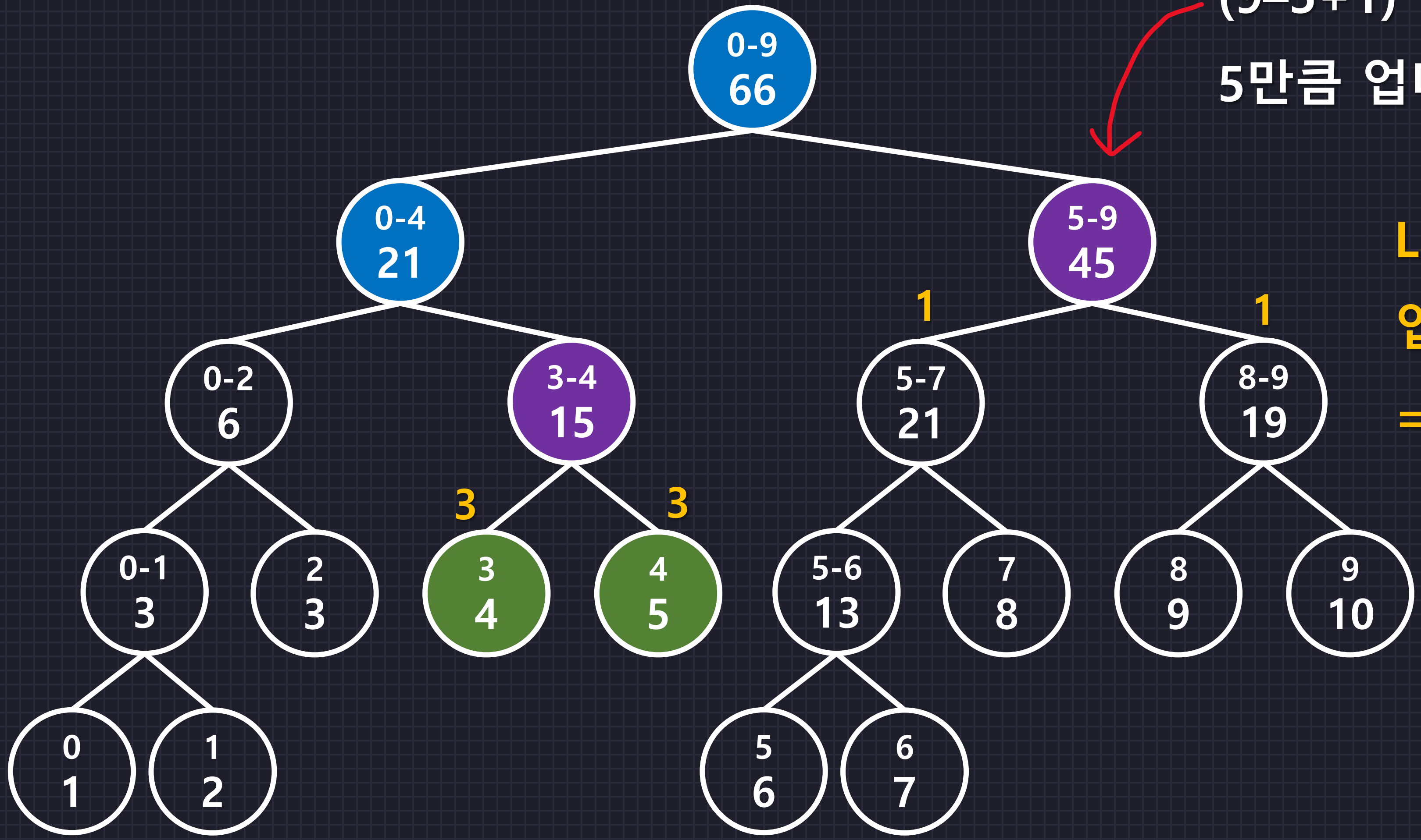
해당 노드 아래로는 업데이트x

Lazy Propagation

Ex) index 3~9 구간의 값을 각각 +1

(9–5+1) * 1 = 5

5만큼 업데이트됨



Lazy propagation으로

업데이트 할 값

= (End - Start + 1) * Value

Lazy Propagation

```
int Make_SegmentTree(int Node, int Start, int End)
{
    if (Start == End) return SegmentTree[Node] = Arr[Start];

    int Mid = (Start + End) / 2;
    int Left_Result = Make_SegmentTree(Node * 2, Start, Mid);
    int Right_Result = Make_SegmentTree(Node * 2 + 1, Mid + 1, End);
    SegmentTree[Node] = Left_Result + Right_Result;

    return SegmentTree[Node];
}

int main(void)
{
    int Tree_Height = (int)ceil(log2(N));
    int Tree_Size = (1 << (Tree_Height + 1));
    SegmentTree.resize(Tree_Size);
    Make_SegmentTree(1, 0, N - 1);
}
```

Lazy Propagation

```
int Sum(int Node, int Start, int End, int Left, int Right)
{
    if (Left > End || Right < Start) return 0;
    if (Left <= Start && End <= Right) return SegmentTree[Node];

    int Mid = (Start + End) / 2;
    int Left_Result = Sum(Node * 2, Start, Mid, Left, Right);
    int Right_Result = Sum(Node * 2 + 1, Mid + 1, End, Left, Right);
    return Left_Result + Right_Result;
}
```

```
void Update_SegmentTree(int Node, int Start, int End, int Index, int Diff)
{
    if (Index < Start || Index > End) return;
    SegmentTree[Node] = SegmentTree[Node] + Diff;

    if (Start != End)
    {
        int Mid = (Start + End) / 2;
        Update_SegmentTree(Node * 2, Start, Mid, Index, Diff);
        Update_SegmentTree(Node * 2 + 1, Mid + 1, End, Index, Diff);
    }
}
```

BIT ; Binary Indexed Tree (Fenwick Tree)

- 여러 개의 데이터가 존재할 때 누적합에 대한 연산을 위한 자료구조
- 비트를 이용한 연산 → 코드가 간결함!
- 어떤 수 x를 이진수로 나타냈을 때, 마지막 1의 위치

$3 = 11_2$ → L[3] = 1

$10 = 1010_2$ → L[10] = 2

$12 = 1100_2$ → L[12] = 4

A(x)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A(x)	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	10000
L[x]	1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16

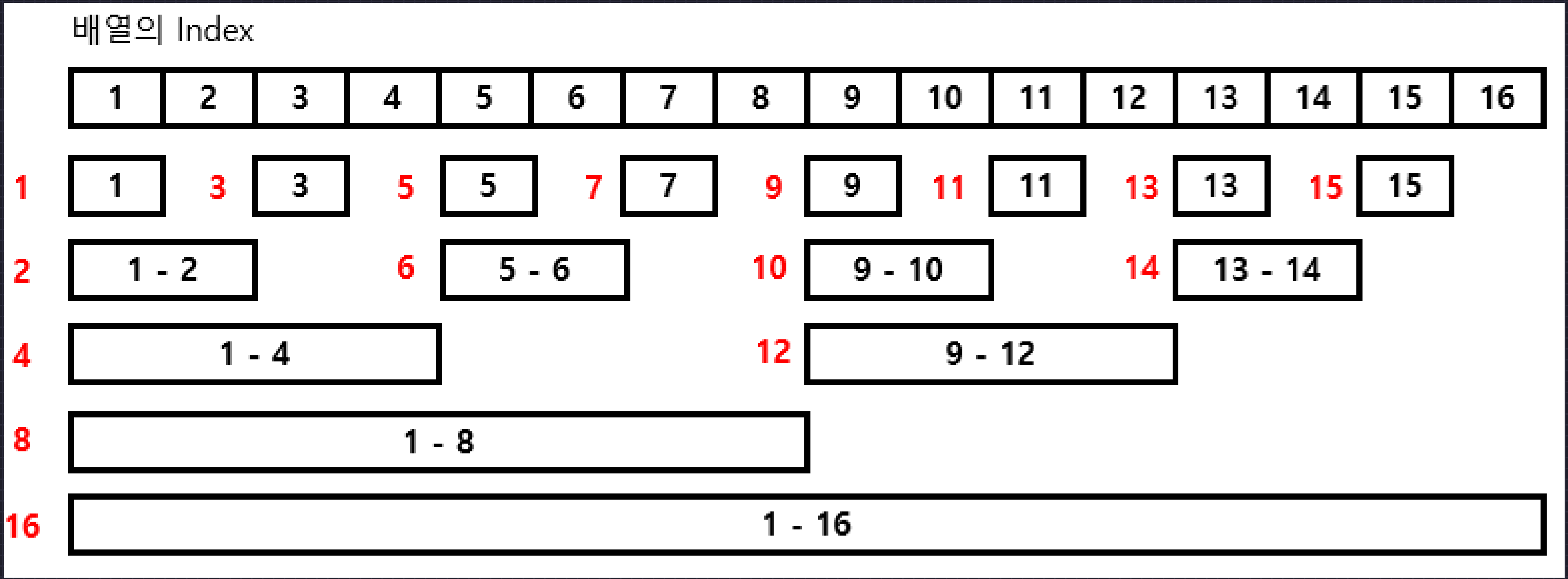
BIT ; Binary Indexed Tree (Fenwick Tree)

A(x)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A(x)	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	10000
L[x]	1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16

- 홀수번째의 L[x]값은 모두 1

BIT ; Binary Indexed Tree (Fenwick Tree)

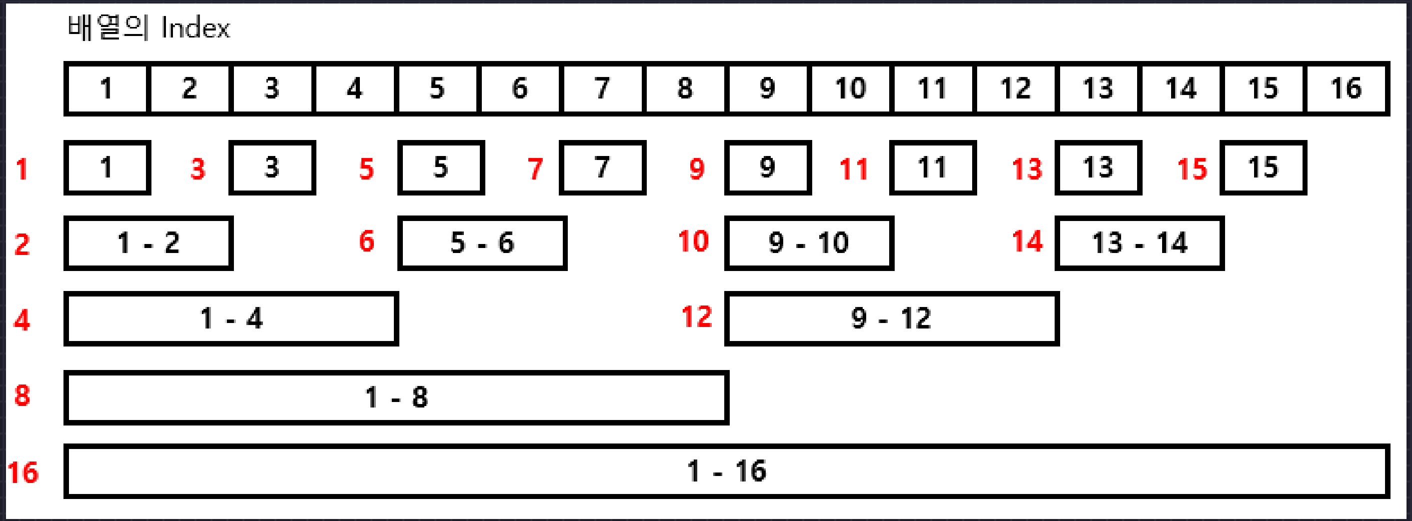
- 배열 아래는 모두 fenwick tree, 빨간 숫자는 fenwick tree의 인덱스



- Tree[홀수] = x (배열의 값을 그대로)

BIT ; Binary Indexed Tree (Fenwick Tree)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	10000
1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16



x=12인 경우

A(x) = 12

L[x] = 4

Tree[x] = L(12)+L(11)+L(10)+L(9)

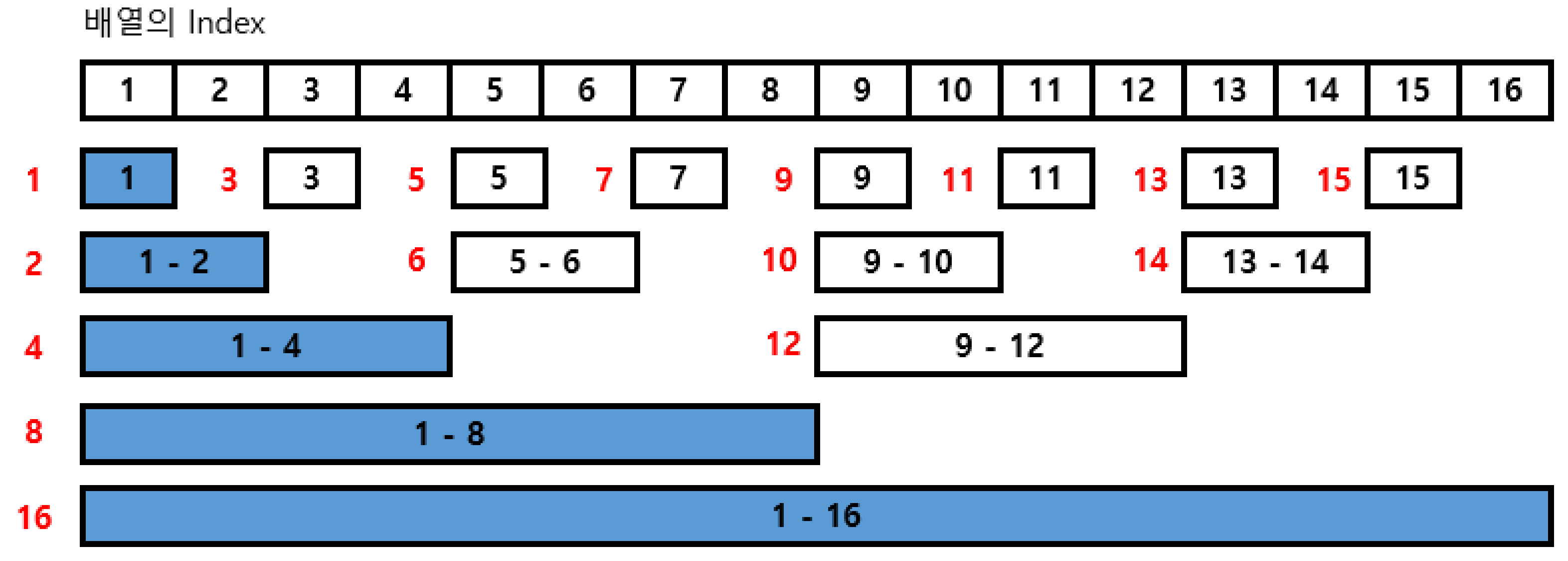
x가 홀수일 때 L[x] = 1, Tree[x] = x

x가 짝수일 때 L[x] = “1이 존재하는 최하위 비트값”,

Tree[x] = “x부터 앞으로 L[x]개의 합“

BIT ; Binary Indexed Tree (Fenwick Tree)

- A[1]의 값이 생성됨에 따라 변화가 생기는 펜윅트리

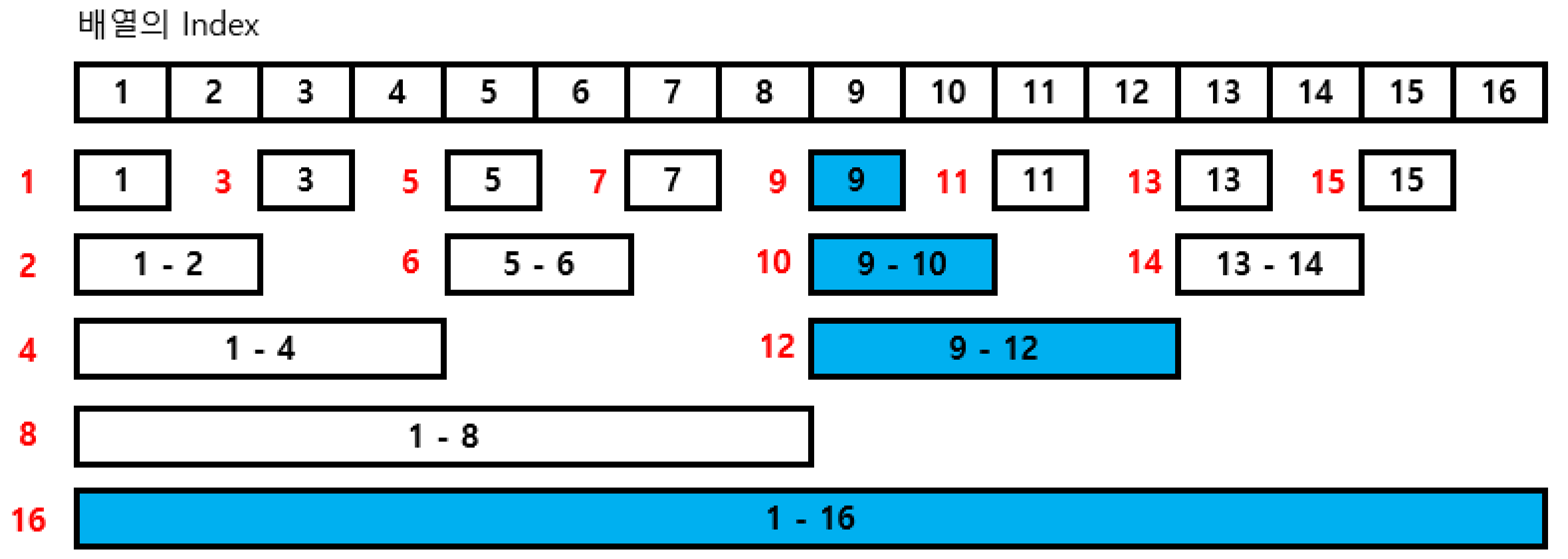


$1 = 0001_{(2)}$
$2 = 0010_{(2)}$
$4 = 0100_{(2)}$
$8 = 1000_{(2)}$
$16 = 10000_{(2)}$

→ 1이 존재하는 최하위 비트를 찾아, 거기에 1을 더하는 연산

BIT ; Binary Indexed Tree (Fenwick Tree)

- A[9]에 의해 변화가 생기는 펜윅트리



$9 = 1001_{(2)}$
$10 = 1010_{(2)}$
$12 = 1100_{(2)}$
$16 = 10000_{(2)}$

→ 1이 존재하는 최하위 비트를 찾아, 거기에 1을 더하는 연산

BIT ; Binary Indexed Tree (Fenwick Tree)

- 1이 나오는 최하위 비트를 찾는 방법?
- index번호 = index번호 + (index번호 & -index번호)

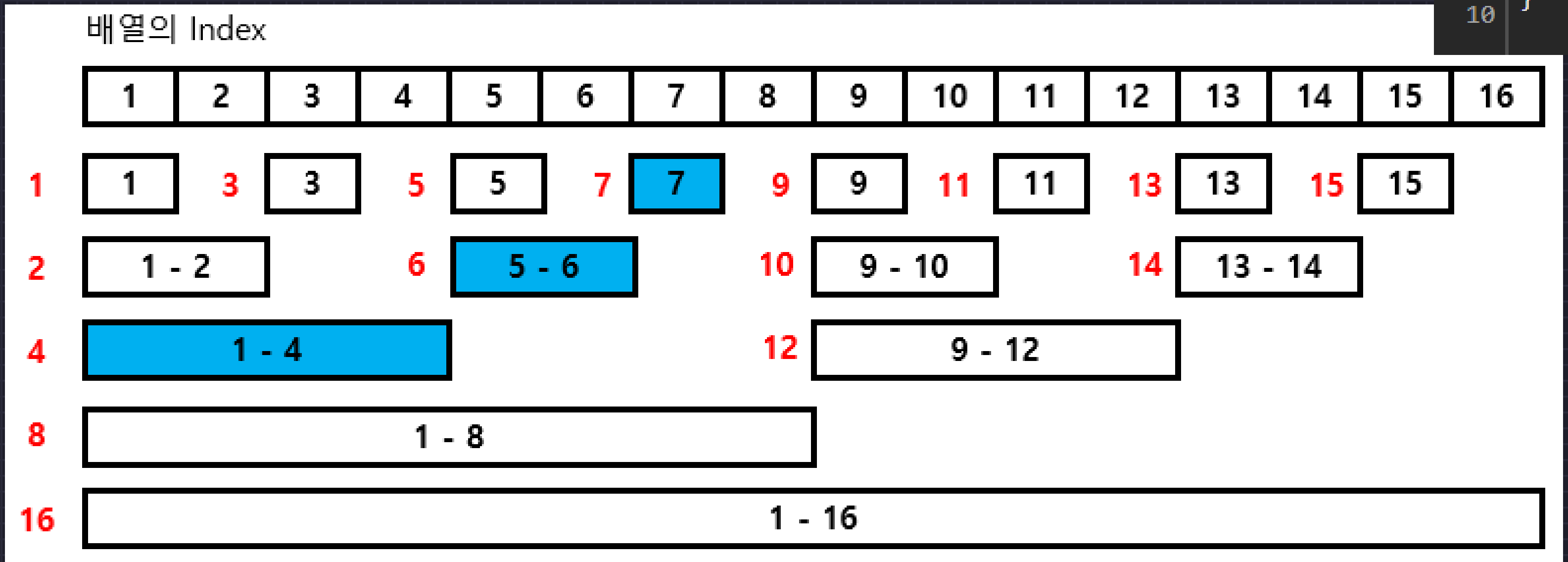
```
void Update(int Idx, int Value)
{
    while (Idx < Fenwick_Tree.size())
    {
        Fenwick_Tree[Idx] = Fenwick_Tree[Idx] + Value;
        Idx = Idx + (Idx & -Idx);
    }
}
```

```
void Make_PenwickTree()
{
    for (int i = 1; i <= N; i++)
    {
        Update(i, Arr[i]);
    }
}
```

BIT ; Binary Indexed Tree (Fenwick Tree)

```
int Sum(int Idx)
{
    ll Result = 0;
    while (Idx > 0)
    {
        Result = Result + Fenwick_Tree[Idx];
        Idx = Idx - (Idx & -Idx);
    }
    return Result;
}
```

- 구간에 대한 연산 : 1~7번 index의 합? (누적합)



- Tree[7]+Tree[6]+Tree[4]

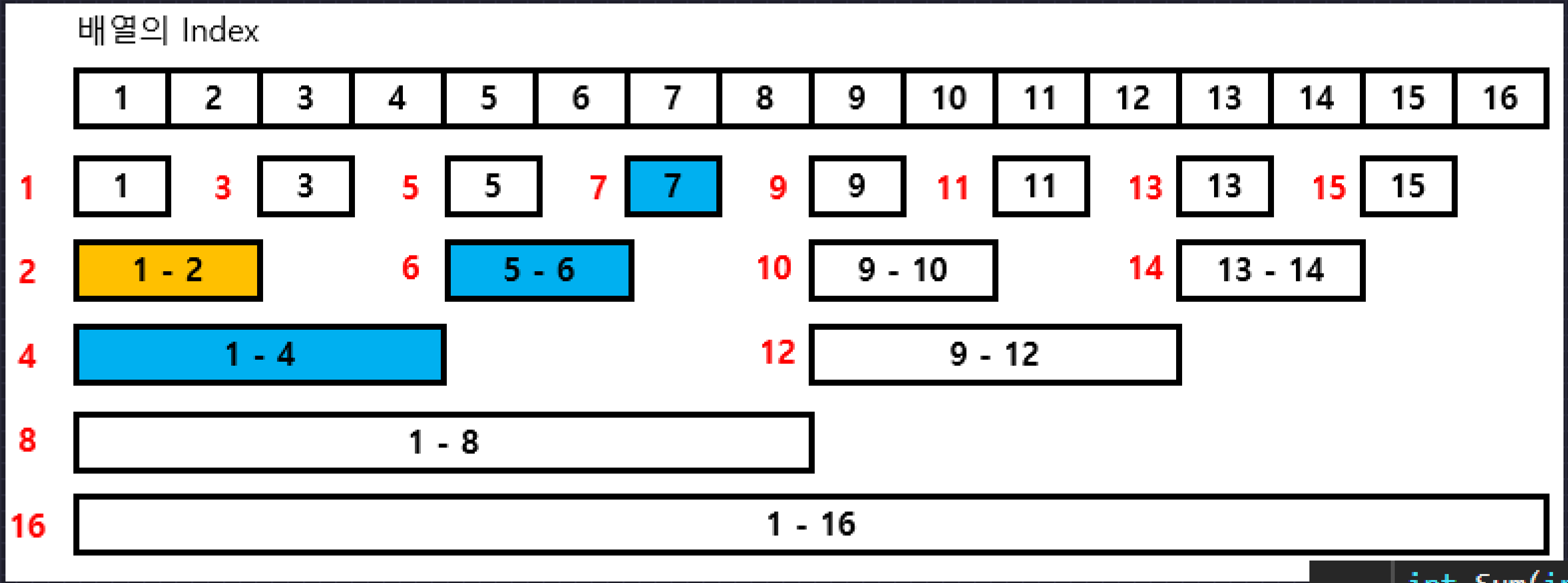
$7 = 0111_{(2)}$
$6 = 0110_{(2)}$
$4 = 0100_{(2)}$

해당 Index까지의 누적합
= 현재 Index번호 - (현재 Index번호 & -현재 Index번호)

→ 1이 존재하는 최하위비트를 찾아서, 해당 비트에 1을 빼는 연산

BIT ; Binary Indexed Tree (Fenwick Tree)

- 구간에 대한 연산 : 3~7번 index의 합? (구간합)



- Tree[4]+Tree[6]+Tree[7]-Tree[2]

```
int Sum(int Idx)
{
    ll Result = 0;
    while (Idx > 0)
    {
        Result = Result + Fenwick_Tree[Idx];
        Idx = Idx - (Idx & -Idx);
    }
    return Result;
}
```

BIT ; Binary Indexed Tree (Fenwick Tree)

누적합

- Boj3020 개똥벌레(G5) https://www.acmicpc.net/problem/3020

Segment tree

- Boj11658 구간 합 구하기(G1) https://www.acmicpc.net/problem/11658
- Boj11505 구간 곱 구하기(G1) https://www.acmicpc.net/problem/11505

Lazy propagation

- Boj10999 구간 합 구하기 2(P4) https://www.acmicpc.net/problem/10999
- Boj1395 스위치(P3) https://www.acmicpc.net/problem/1395

BIT

- Boj11658 구간 합 구하기 3(P5) https://www.acmicpc.net/problem/11658

Thank you.