
# Binary Indexed Tree & Segment Tree

2018 Alkor 중급반 세미나 - 손민철

# 겁나는 쿼리 문제..


14427번 | 제출 | 맞은 사람 | 숏코딩 | 풀이 | 풀이 작성 | 풀이 요청 | 재채점/수정 | 채점 현황 | 내 소스 | 강의 | 질문 검색

## 수열과 쿼리 15 성공

| 시간 제한 | 메모리 제한 | 제출 | 정답 | 맞은 사람 | 정답 비율 |
|---|---|---|---|---|---|
| 2 초 | 512 MB | 81 | 68 | 59 | 89.394% |

### 문제

길이가 N인 수열 $A_1$, $A_2$, ..., $A_N$이 주어진다. 이때, 다음 쿼리를 수행하는 프로그램을 작성하시오.

- `1 i v` : $A_i$를 v로 바꾼다.
- `2` : 수열에서 가장 크기가 작은 값의 인덱스를 출력한다. 크기가 작은 값이 여러개인 경우에는 인덱스가 작은 것을 출력한다.

수열의 인덱스는 1부터 시작한다.

쿼리 문제를 만나면 겁이 난다 8ㅅ8..
하지만 Binary Indexed Tree & Segment Tree와 함께라면 기초적인 쿼리 문제는 더 이상 두려워하지 않아도 괜찮다!!

# 워밍업 (1/2)


## 구간 합 구하기 4 성공 icpc.me/11659

| 시간 제한 | 메모리 제한 | 제출 | 정답 | 맞은 사람 | 정답 비율 |
|---|---|---|---|---|---|
| 1 초 | 256 MB | 5312 | 2862 | 2226 | 53.833% |

### 문제

수 N개가 주어졌을 때, i번째 수부터 j번째 수까지 합을 구하는 프로그램을 작성하시오.

### 입력

첫째 줄에 수의 개수 N (1 ≤ N ≤ 100,000), 합을 구해야 하는 횟수 M (1 ≤ M ≤ 100,000)이 주어진다. 둘째 줄에는 N개의 수가 주어진다. 수는 1,000보다 작거나 같은 자연수이다. 셋째 줄부터 M개의 줄에는 합을 구해야 하는 구간 i와 j가 주어진다.

### 출력

총 M개의 줄에 입력으로 주어진 i번째 수부터 j번째 수까지 합을 출력한다.

$arr$ : 1 5 2 4 10

$D[0] = 0$
$D[1] = 1$
$D[2] = 6$
$D[3] = 8$
$D[4] = 12$
$D[5] = 22$

$D[i] = \sum_{k=1\ to\ i} arr[i]$ 으로 두면 각 쿼리를 $O(1)$ 로 해결 가능 (Prefix sum)

# 워밍업 (2/2)


## 구간 합 구하기 성공

icpc.me/2042

| 시간 제한 | 메모리 제한 | 제출 | 정답 | 맞은 사람 | 정답 비율 |
|---|---|---|---|---|---|
| 2 초 | 256 MB | 18316 | 6073 | 3105 | 29.529% |

### 문제

어떤 N개의 수가 주어져 있다. 그런데 중간에 수의 변경이 빈번히 일어나고 그 중간에 어떤 부분의 합을 구하려 한다. 만약에 1,2,3,4,5 라는 수가 있고, 3번째 수를 6으로 바꾸고 2번째부터 5번째까지 합을 구하라고 한다면 17을 출력하면 되는 것이다. 그리고 그 상태에서 다섯 번째 수를 2로 바꾸고 3번째부터 5번째까지 합을 구하라고 한다면 12가 될 것이다.

### 입력

첫째 줄에 수의 개수 N(1 ≤ N ≤ 1,000,000)과 M(1 ≤ M ≤ 10,000), K(1 ≤ K ≤ 10,000) 가 주어진다. M은 수의 변경이 일어나는 회수이고, K는 구간의 합을 구하는 회수이다. 그리고 둘째 줄부터 N+1번째 줄까지 N개의 수가 주어진다. 그리고 N+2번째 줄부터 N+M+K+1번째 줄까지 세 개의 정수 a, b, c가 주어지는데, a가 1인 경우 b번째 수를 c로 바꾸고 a가 2인 경우에는 b번째 수부터 c번째 수까지의 합을 구하여 출력하면 된다.

a가 1인 경우 c는 long long 범위를 넘지 않는다.

### 출력

첫째 줄부터 K줄에 걸쳐 구한 구간의 합을 출력한다. 단, 정답은 long long 범위를 넘지 않는다.

$arr$ : 1 5 2 4 10

$D[0] = 0$
$D[1] = 1$
$D[2] = 6$
$D[3] = 8$
$D[4] = 12$
$D[5] = 22$

으로 계산을 다 해두었는데
$arr[i] = 3$ 으로 변경된다면,
prefix sum을 최대 N개 바꾸므로
Update가 $O(N)$ 이 걸린다.

원소가 바뀌고 나면 Prefix sum을 구한 것이 아무 의미 없어진다. 이 문제를 어썸하게 해결할 수 있는 Binary Indexed Tree를 이제 배워보자!

# Binary Indexed Tree의 연산


Binary Indexed Tree에서 제공하는 연산

- **ADD(x)** : arr[1]부터 arr[x]까지의 합을 반환, $O(lgN)$
- **UPDATE(x, val)** : arr[x]의 값을 val만큼 변경, $O(lgN)$

다시 앞의 문제를 생각해보면, 특정  위치의 수를 변경하는 것과 특정  구간의 합을 구하는 것을 모두 $O(lgN)$ 에 수행할 수 있게 되어 시간 내에 풀이가 가능하다.

# Binary Indexed Tree의 구현 (1/3)


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BIT[16] | | | | | | | | | | | | | | | | | | | | |
| BIT[8] | | | | | | | | | | | | | | | | | | | | |
| BIT[4] | | | | | | | | BIT[12] | | | | | | | | BIT[20] | | | | |
| BIT[2] | | | | BIT[6] | | | | BIT[10] | | | | BIT[14] | | | | BIT[18] | | | | |
| 1 | | 3 | | 5 | | 7 | | | | 11 | | 13 | | 15 | | 17 | | 19 | | 21 |

위와 같이 BIT라는 배열을 만들어놓았다고 가정해보자.

- ADD(21) : $BIT[16] + BIT[20] + BIT[21]$
- UPDATE(11, val) : $BIT[11], BIT[12], BIT[16]$ 변경

뭔가 시간복잡도가 많이 줄어들 느낌이다!!

# Binary Indexed Tree의 구현 (2/3)

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BIT[16] | | | | | | | | | | | | | | | | | | | | |
| BIT[8] | | | | | | | | | | | | | | | | | | | | |
| BIT[4] | | | | | | | | BIT[12] | | | | | | | | BIT[20] | | | | |
| BIT[2] | | | | BIT[6] | | | | BIT[10] | | | | BIT[14] | | | | BIT[18] | | | | |
| 1 | | 3 | | 5 | | 7 | | | | 11 | | 13 | | 15 | | 17 | | 19 | | 21 |

**BIT**$[x]$는 어디서부터 어디까지의 합을 가지고 있는걸까?

BIT$[20 = 10100_{(2)}]$ = arr[17]+arr[18]+arr[19]+arr[20]
BIT$[10 = 1010_{(2)}]$ = arr[9]+arr[10]
BIT$[8 = 1000_{(2)}]$ = arr[1]+arr[2] +arr[3] +arr[4] +arr[5] +arr[6] +arr[7] +arr[8]

규칙이 보이나요?

# Binary Indexed Tree의 구현 (3/3)


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BIT[16] | | | | | | | | | | | | | | | | | | | | |
| BIT[8] | | | | | | | | | | | | | | | | | | | | |
| BIT[4] | | | | | | | | BIT[12] | | | | | | | | BIT[20] | | | | |
| BIT[2] | | | | BIT[6] | | | | BIT[10] | | | | BIT[14] | | | | BIT[18] | | | | |
| 1 | | 3 | | 5 | | 7 | | | | 11 | | 13 | | 15 | | 17 | | 19 | | 21 |

**BIT**$[x]$는 어디서부터 어디까지의 합을 가지고 있는걸까?

BIT$[20 = 10100_{(2)}]$ = arr[17]+arr[18]+arr[19]+arr[20]
BIT$[10 = 1010_{(2)}]$ = arr[9]+arr[10]
BIT$[8 = 1000_{(2)}]$ = arr[1]+arr[2] +arr[3] +arr[4] +arr[5] +arr[6] +arr[7] +arr[8]

마지막 1이 나타내는 값만큼의 수의 합을 저장한다.

# Binary Indexed Tree의 구현 - ADD (1/3)


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BIT[16] | | | | | | | | | | | | | | | | | | | | |
| BIT[8] | | | | | | | | | | | | | | | | | | | | |
| BIT[4] | | | | | | | | BIT[12] | | | | | | | | BIT[20] | | | | |
| BIT[2] | | | | BIT[6] | | | | BIT[10] | | | | BIT[14] | | | | BIT[18] | | | | |
| 1 | | 3 | | 5 | | 7 | | | | 11 | | 13 | | 15 | | 17 | | 19 | | 21 |

$BIT[21 = 10101_{(2)}] = BIT[21=10101_{(2)}]+BIT[20=10100_{(2)}]+BIT[16=10000_{(2)}]$

$BIT[15 = 1111_{(2)}] = BIT[15=1111_{(2)}]+BIT[14=1110_{(2)}]+BIT[12=1100_{(2)}] +BIT[8=1000_{(2)}]$

뭔가 느낌이 오나요?

# Binary Indexed Tree의 구현 - ADD (2/3)


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BIT[16] | | | | | | | | | | | | | | | | | | | | |
| BIT[8] | | | | | | | | | | | | | | | | | | | | |
| BIT[4] | | | | | | | | BIT[12] | | | | | | | | BIT[20] | | | | |
| BIT[2] | | | | BIT[6] | | | | BIT[10] | | | | BIT[14] | | | | BIT[18] | | | | |
| 1 | | 3 | | 5 | | 7 | | | | 11 | | 13 | | 15 | | 17 | | 19 | | 21 |

$BIT[21 = 10101_{(2)}] = BIT[21=10101_{(2)}]+BIT[20=10100_{(2)}]+BIT[16=10000_{(2)}]$

$BIT[15 = 1111_{(2)}] = BIT[15=1111_{(2)}]+BIT[14=1110_{(2)}]+BIT[12=1100_{(2)}] +BIT[8=1000_{(2)}]$

마지막 1을 계속 빼나가면서 더하면 된다!

# Binary Indexed Tree의 구현 - ADD (3/3)


마지막 1을 계속 빼나가면서 더하면 된다!

```
int sum(int i){
  int ret = 0;
  while(i){
    ret += BIT[i];
    i -= (i & -i);
  }
  return ret;
}
```

-i가 ~i + 1이므로 i와 -i는 최초로 1이 등장하는 지점의 bit만 같게 된다.

i = 11011000
-i = 00101000

# Binary Indexed Tree의 구현 - UPDATE (1/3)


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BIT[16] | | | | | | | | | | | | | | | | | | | | |
| BIT[8] | | | | | | | | | | | | | | | | | | | | |
| BIT[4] | | | | | | | | BIT[12] | | | | | | | | BIT[20] | | | | |
| BIT[2] | | | | BIT[6] | | | | BIT[10] | | | | BIT[14] | | | | BIT[18] | | | | |
| 1 | | 3 | | 5 | | 7 | | | | 11 | | 13 | | 15 | | 17 | | 19 | | 21 |

arr[13= 1101 (2)]을 변경 : BIT[13 = 1101 (2)], BIT[14=1110(2)], BIT[16=10000(2)]을 변경
arr[17= 10001 (2)]을 변경 : BIT[17 = 10001 (2)], BIT[18=10010(2)], BIT[20=10100(2)]을 변경

뭔가 느낌이 오나요?

# Binary Indexed Tree의 구현 - UPDATE (2/3)


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BIT[16] | | | | | | | | | | | | | | | | | | | | |
| BIT[8] | | | | | | | | | | | | | | | | | | | | |
| BIT[4] | | | | | | | | BIT[12] | | | | | | | | BIT[20] | | | | |
| BIT[2] | | | | BIT[6] | | | | BIT[10] | | | | BIT[14] | | | | BIT[18] | | | | |
| 1 | | 3 | | 5 | | 7 | | | | 11 | | 13 | | 15 | | 17 | | 19 | | 21 |

arr[13= 1101 (2)]을 변경 : BIT[13 = 1101 (2)], BIT[14=1110(2)], BIT[16=10000(2)]을 변경
arr[17= 10001 (2)]을 변경 : BIT[17 = 10001 (2)], BIT[18=10010(2)], BIT[20=10100(2)]을 변경

이전에 변경한 index에서 마지막 1을 더해주면 됩니다.
ex) 13+1 = 14, 14+2 = 16 // 17+1 = 18, 18+2 = 20

# Binary Indexed Tree의 구현 - UPDATE (3/3)


마지막 1을 계속 더해나가면서 갱신해주면 된다!

```
void update(int i, int val){
    while(i <= n){
        BIT[i] += val;
        i += (i & -i);
    }
}
```

# Binary Indexed Tree의 구현 - INIT


맨 처음 배열의 원소를 입력받았을 때 BIT 배열을 채우는 방법

- Prefix Sum을 이용하면 $O(N)$

- Update 함수를 이용하면 $O(NlgN)$

Update를 사용하는  방식이 시간복잡도가 안 좋지만 PS에서는 $O(N)$이나 $O(NlgN)$ 이나 도찐개찐이기 때문에 그냥 이미 만들어둔 Update함수를 가져다 써서 BIT 배열을 채우는 것을 추천한다.

# Binary Indexed Tree의 응용과 한계


Q. 1부터 x까지의 합이 아니라 a부터 b까지의 합을 구해야 하는 쿼리가 들어온다면?
A. sum(b)-sum(a-1)

Q. x번째 값을 val만큼 더하는 것이 아니라 x번째 값을 val로 변경하는 쿼리가 들어온다면?
A. arr[x] 값을 따로 저장하고 있으면 됨

Q. 1부터 x까지의 합이 아니라 1부터 x까지의 곱/GCD/최댓값/최솟값을 구해야 하는 쿼리가 들어온다면?
A. sum, update 함수를 변경하면 됨

Q. 1부터 x까지의 GCD/최댓값이 아니라 a부터 b까지의 GCD/최댓값을 구해야 하는 쿼리가 들어온다면?
A. 불가능ㅠㅠ Segment Tree가 필요하다.

# Segment Tree의 연산


Segment Tree에서 제공하는 연산

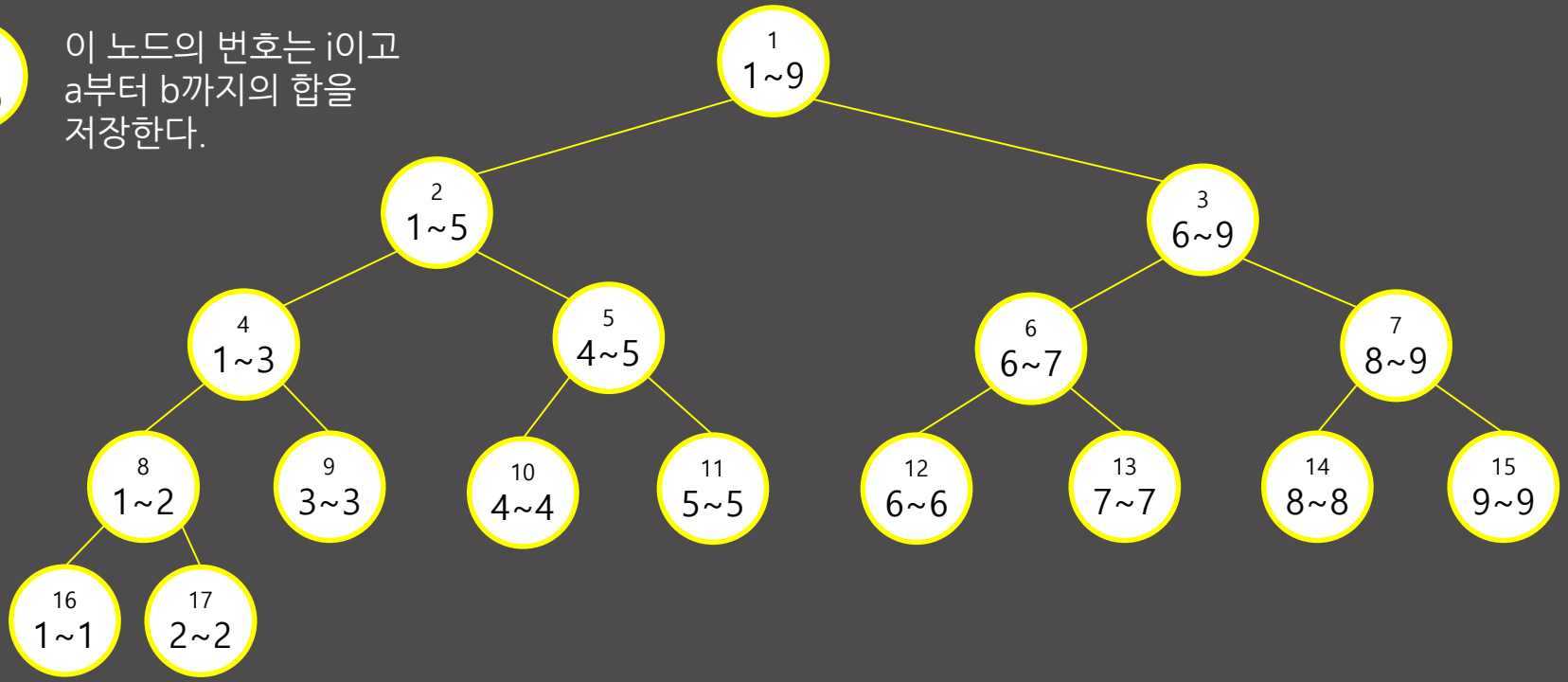
- ADD(a, b) : arr[a]부터 arr[b]까지의 합을 반환, $O(lgN)$
- UPDATE(x, val) : arr[x]의 값을 val만큼 변경, $O(lgN)$

BIT보다 더 좋아보인다!!

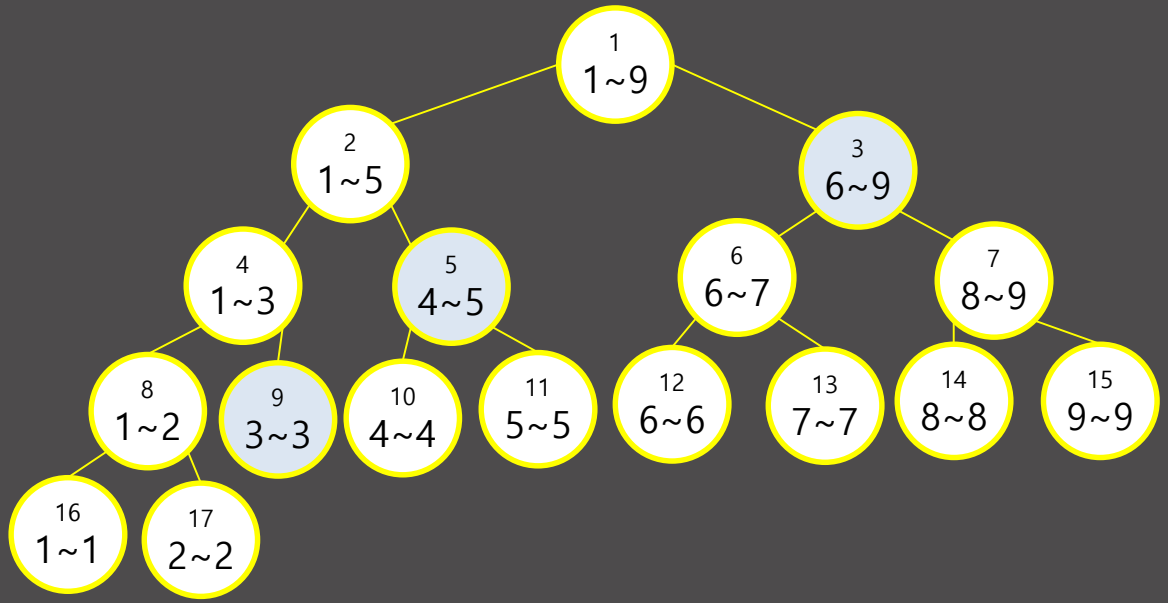
# Segment Tree의 구현


i
a~b

이 노드의 번호는 i이고
a부터 b까지의 합을
저장한다.



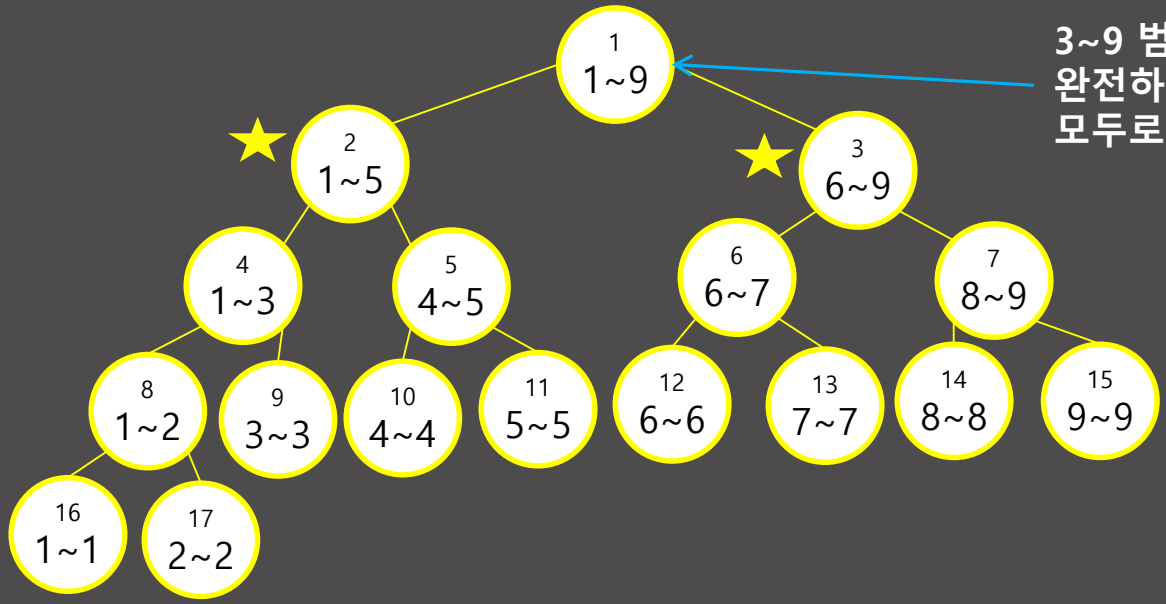
# Segment Tree의 구현 - ADD (1/9)


3부터 9의 합은 3,5,9번 노드의 합이다.

뭔가 구간을 적절하게 택하면 a부터 b까지의 합을 효율적으로 구할 수 있을 것 같은데 어떻게 구간을 택해야할까?

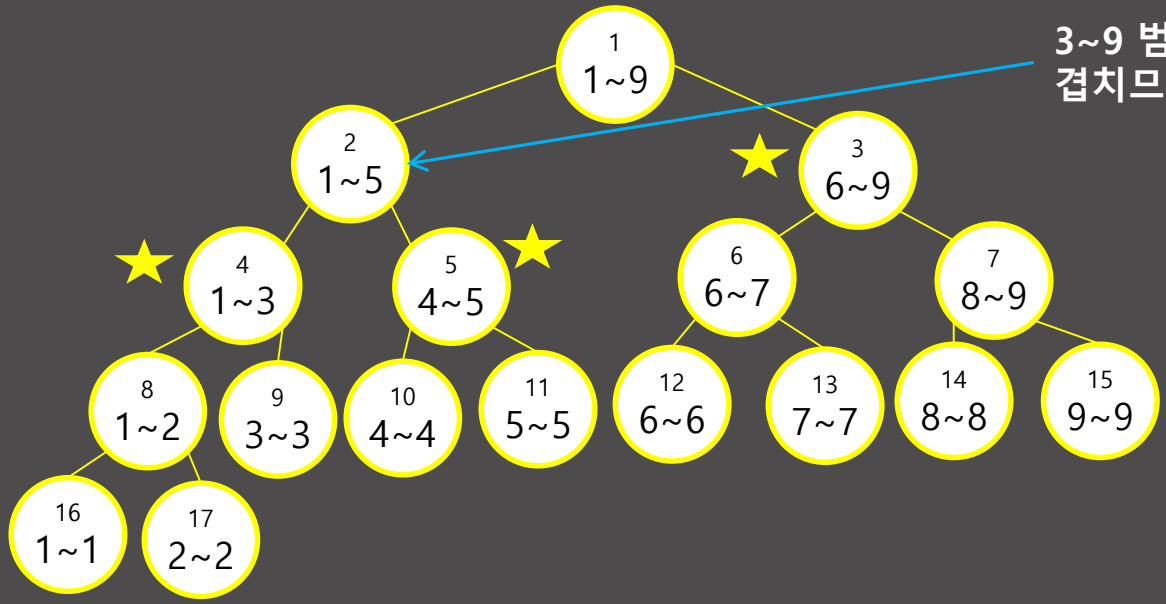
# Segment Tree의 구현 - ADD (2/9)




3~9 범위는 현재 노드의 범위(1~9)에 완전하게 포함되므로 왼쪽과 오른쪽 자식 모두로 들어간다.

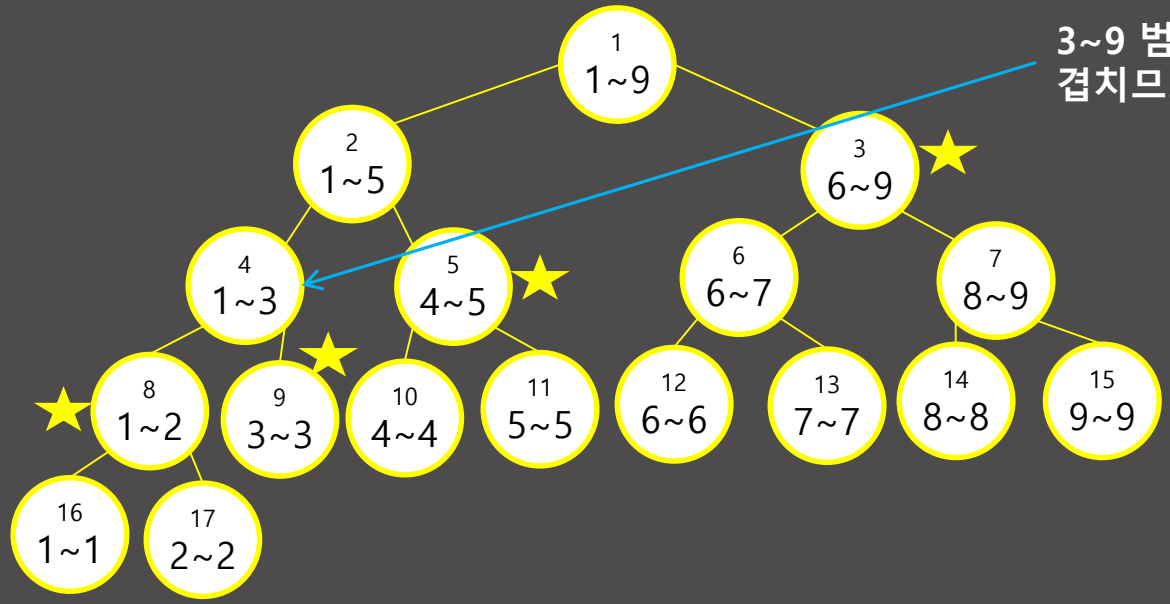
3부터 9까지의 합을 구하고 싶은 상황, root에서부터 top-down 형식으로 순회를 해보자.
별은 탐색을 해야하나 아직 완료하지 않은 노드를 의미.

# Segment Tree의 구현 - ADD (3/9)

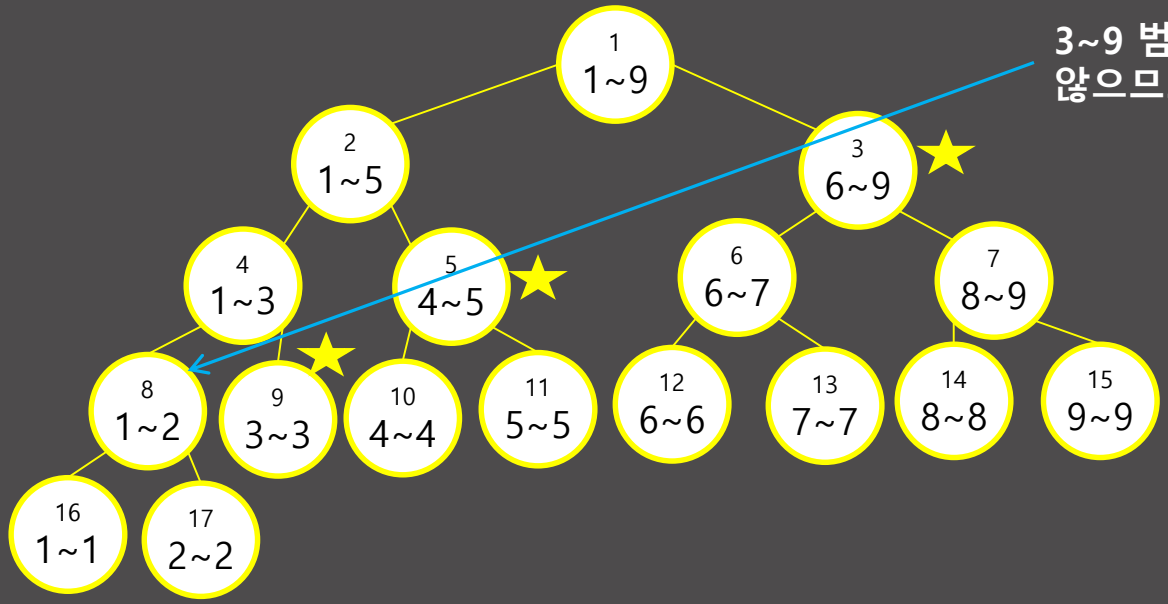

3~9 범위는 현재 노드의 범위(1~5)와 일부가 겹치므로 왼쪽, 오른쪽 자식 모두로 들어간다.

# Segment Tree의 구현 - ADD (4/9)



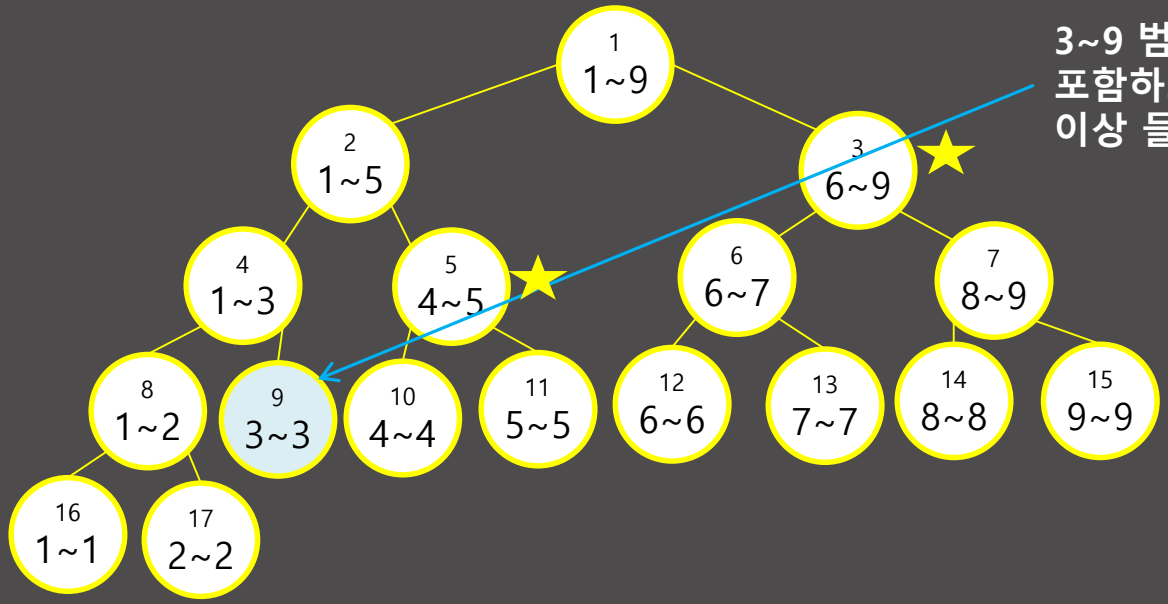

3~9 범위는 현재 노드의 범위(1~3)와 일부가 겹치므로 왼쪽, 오른쪽 자식 모두로 들어간다.

# Segment Tree의 구현 - ADD (5/9)

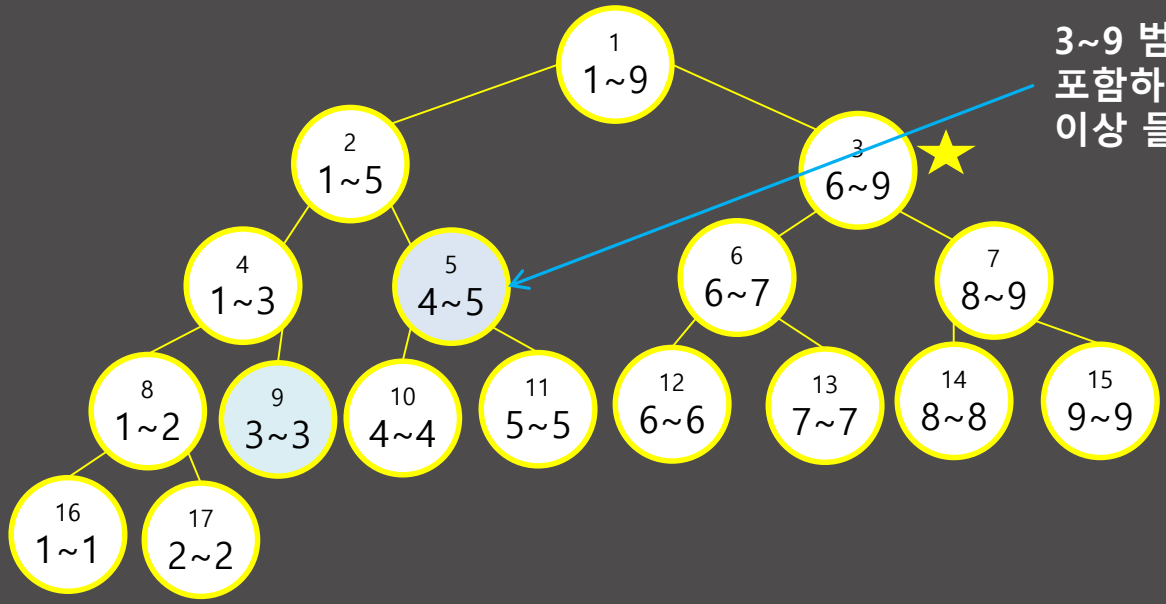

3~9 범위는 현재 노드의 범위(1~2)와 겹치지 않으므로 더 이상 들어가지 않는다.

# Segment Tree의 구현 - ADD (6/9)

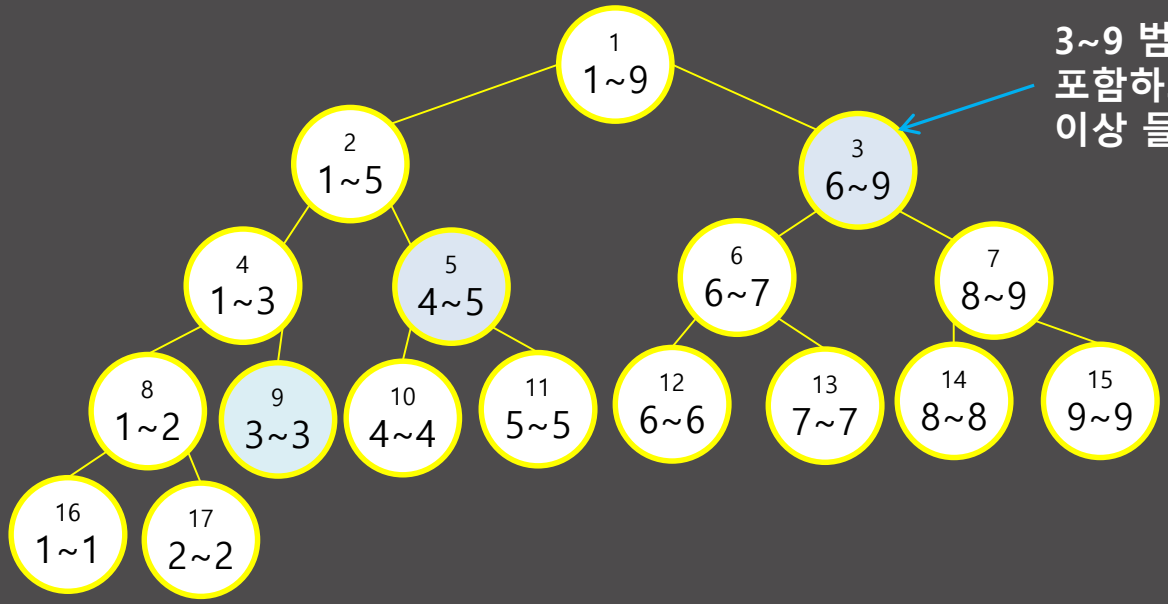

3~9 범위는 현재 노드의 범위(3~3)를 완전히 포함하므로 현재 노드의 값을 저장하고 더 이상 들어가지 않는다.

# Segment Tree의 구현 - ADD (7/9)


3~9 범위는 현재 노드의 범위(4~5)를 완전히 포함하므로 현재 노드의 값을 저장하고 더 이상 들어가지 않는다.

# Segment Tree의 구현 - ADD (8/9)



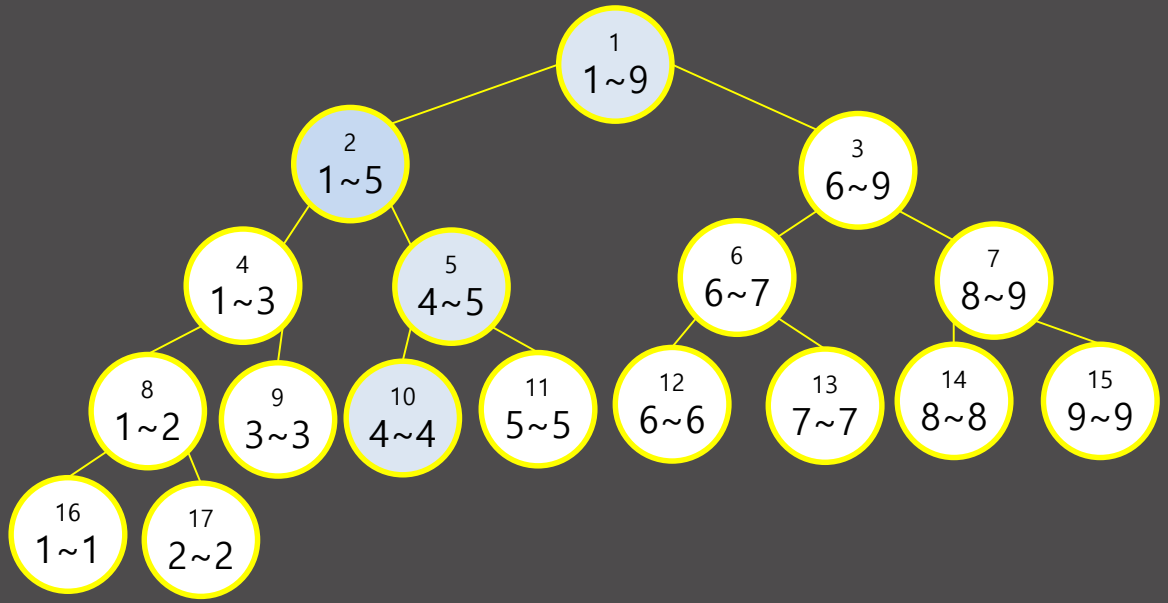

3~9 범위는 현재 노드의 범위(6~9)를 완전히 포함하므로 현재 노드의 값을 저장하고 더 이상 들어가지 않는다.

# Segment Tree의 구현 - ADD (9/9)


```
// idx번째 node가 나타내는 구간이 st에서 en이고 l부터 r까지의 합을 구하고 싶다.
int sum(int idx, int st, int en, int l, int r){
  if(l > en or r < st) return 0;
  if(l <= st and en <= r) return seg[idx];
  return sum(2*idx, st, (st+en)/2, l, r) + sum(2*idx+1, (st+en)/2+1, en, l, r);
}
```

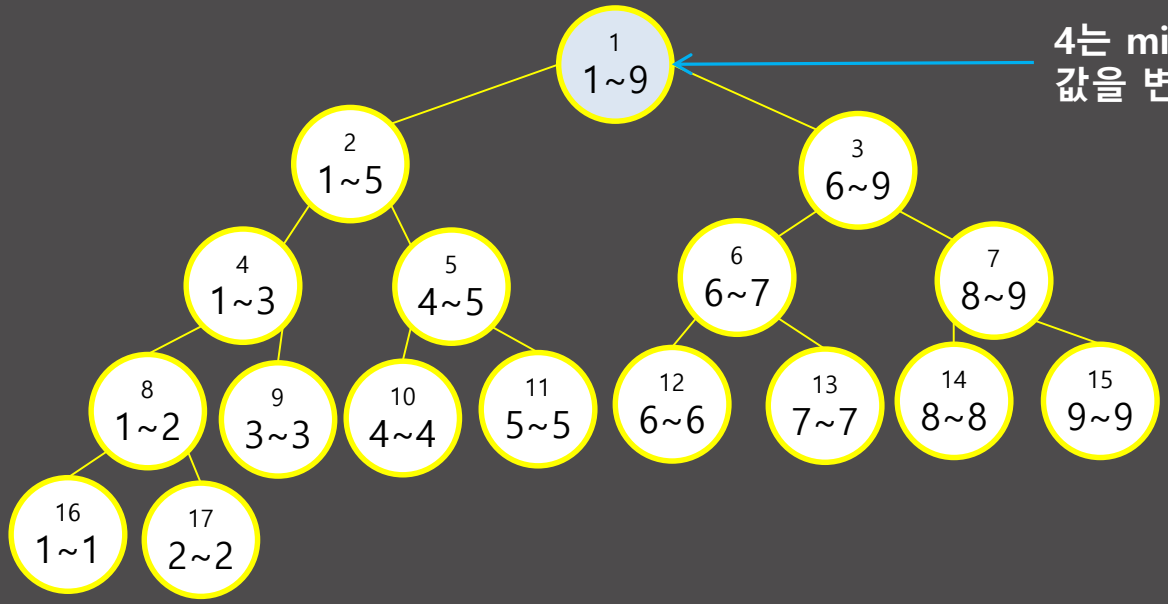
처음 보면 헷갈리는게 당연하다. 직접 Tree를 그려보거나 sum 함수가 어떻게 출력되는지를 보면서 이해해보자.

# Segment Tree의 구현 - UPDATE (1/6)



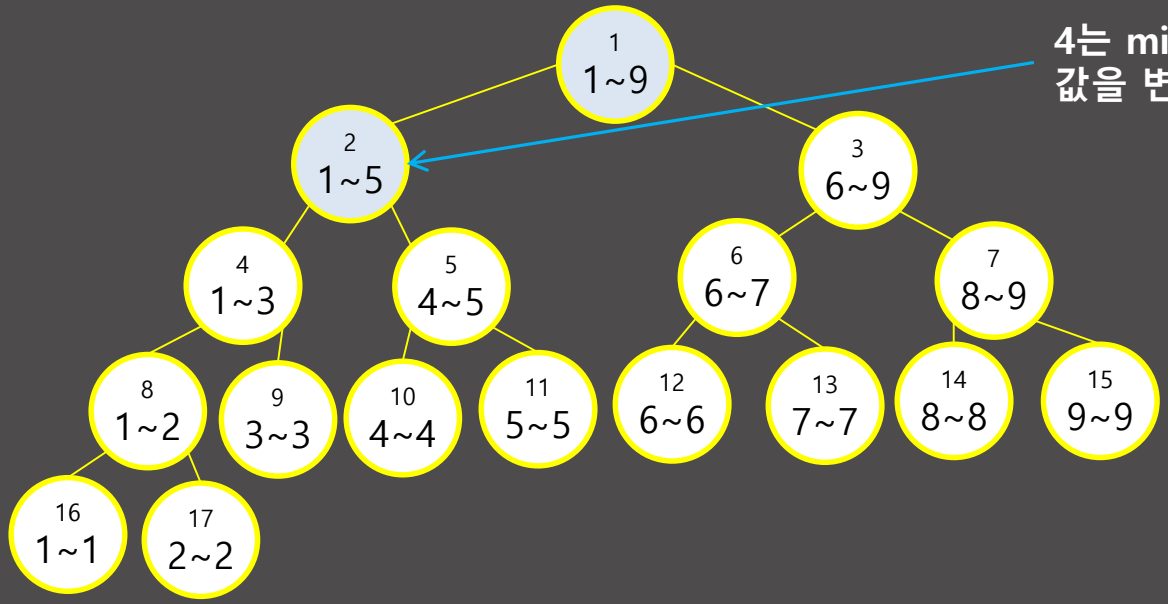

4번째 원소를 update한다면 해당 원소가 들어있는 모든 노드의 값을 변경해줘야 한다. 마찬가지로 root에서 재귀적으로 순회를 돌면서 처리한다.

# Segment Tree의 구현 - UPDATE (2/6)



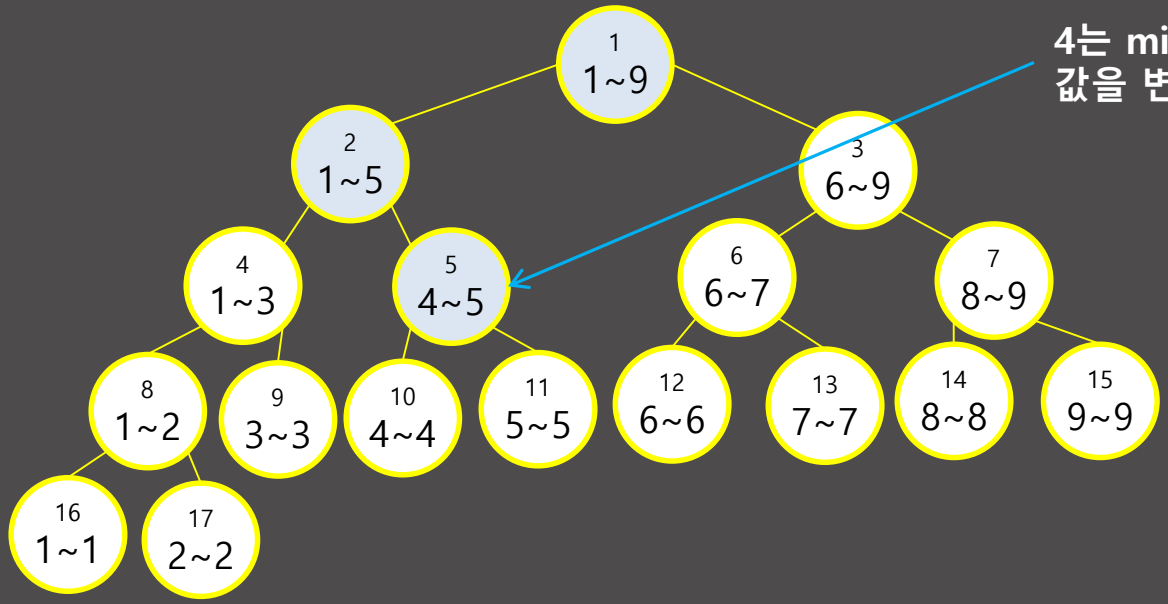

4는 mid(=5) 이하이므로 왼쪽 자식에 속한다.
값을 변경하고 왼쪽으로 이동

# Segment Tree의 구현 - UPDATE (3/6)

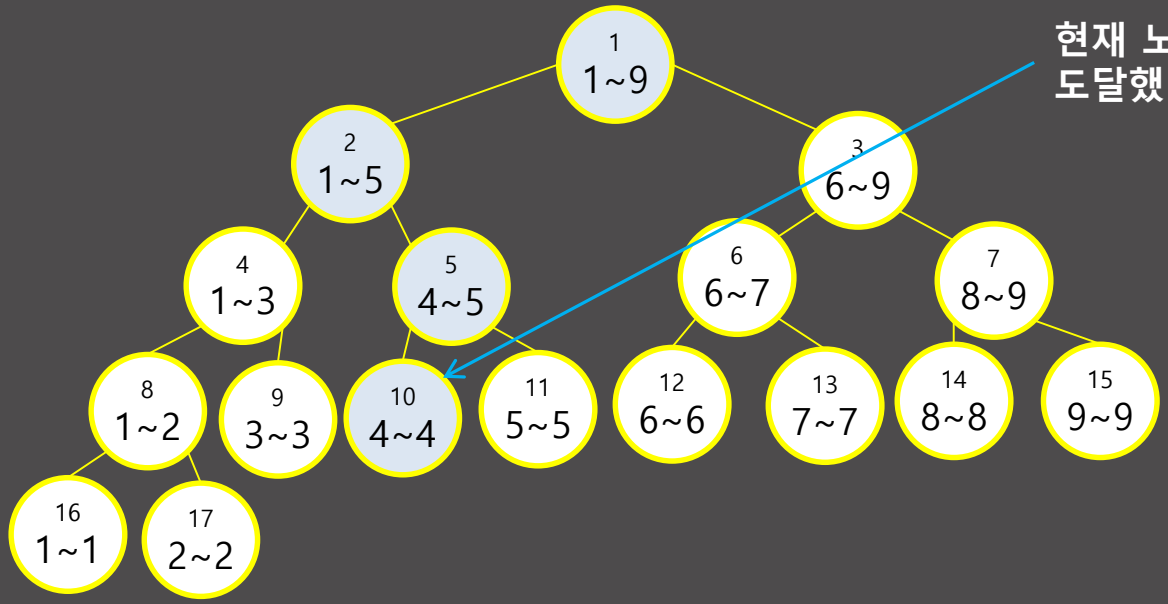

4는 mid(=3) 초과이므로 오른쪽 자식에 속한다.
값을 변경하고 오른쪽으로 이동

# Segment Tree의 구현 - UPDATE (4/6)


4는 mid(=4) 이하이므로 왼쪽 자식에 속한다.
값을 변경하고 왼쪽으로 이동

# Segment Tree의 구현 - UPDATE (5/6)




현재 노드의 left와 right가 일치하므로 leaf에 도달했다. 값을 갱신하고 종료.

# Segment Tree의 구현 - UPDATE (6/6)


```
// idx번째 node가 나타내는 구간이 st에서 en이고 i번째 원소를 diff만큼 변화시키고 싶다.
void update(int idx, int st, int en, int i, int diff){
  if(st==en){
    seg[idx] += diff;
    return;
  }
  if(i <= (st+en)/2) update(2*idx,st,(st+en)/2,i,diff);
  else update(2*idx+1,(st+en)/2+1,en,i,diff);
}
```

# Segment Tree의 구현 - INIT


맨 처음 배열의 원소를 입력받았을 때 seg 배열을 채우는 방법

- Prefix Sum을 이용하면 $O(N)$
- Update 함수를 이용하면 $O(NlgN)$

BIT와 마찬가지로 Update를 사용하는  방식이 시간복잡도가 안 좋지만 PS에서는 $O(N)$이나 $O(NlgN)$ 이나 도찐개찐이기 때문에 그냥 이미 만들어둔 Update함수를 가져다 써서 seg 배열을 채우는 것을 추천한다.

# Segment Tree의 응용과 한계


Q. BIT에서 할 수 있는 모든걸 Segment Tree에서 할 수 있는 것 아닌가요?
A. ㅇㅇ. 그러나 BIT는 메모리/수행시간이 더 우수하다는 장점이 있음.(그러나 나는 Segment Tree만 씀)

Q. x번째 값을 val만큼 더하는 것이 아니라 x번째 값을 val로 변경하는 쿼리가 들어온다면?
A. arr[x] 값을 따로 저장하고 있으면 됨.

Q. x번째 값을 val만큼 더하는 것이 아니라 a부터 b번째 까지를 val만큼 더하는 쿼리가 들어온다면?
A.  Lazy Propagation이라는 테크닉을 알아야 한다. 이건 다음 기회에!