

# 01주차: Segment tree

---

강사: 안상욱

# 챕터 0: Range Sum Problem

---

- Naive
- DP
- Segment Tree

BOJ 11659 구간 합 구하기 4

# 구간 합 구하기 4

Arr	1	3	5	4	2	6	7
DP	1	4	9	13	15	21	28

DP에 처음부터 지금까지의 누적 합을 저장한다

# Prefix Sum

- '구간합 구하기 4'의 해답이 Prefix Sum입니다.
- 전처리  $O(n)$ , 쿼리당 시간복잡도  $O(1)$
- 값이 바뀌지 않는다면 가장 빠르다.
- 값이 바뀐다면?

# 값이 바뀐다면?

Arr	1	3	5	4	2	6	7
DP	1	4	9	13	15	21	28

# 값이 바뀐다면?

Arr	1	5	5	4	2	6	7
DP	1	4	9	13	15	21	28

# 값이 바뀐다면?

Arr	1	5	5	4	2	6	7
DP	1	6	11	15	17	23	30

그 Index 이후의 모든 DP 값이 업데이트되어야한다!



# 챕터 1: Segment tree

---

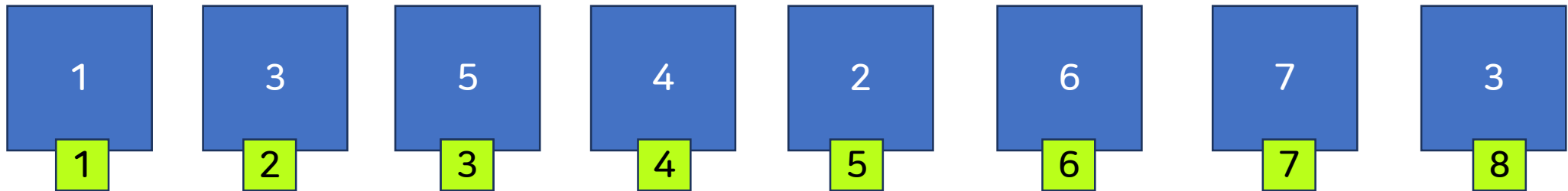
- 도입
- 구현
- 구간 합 말고는 안될까?

# 도입

Arr	1	3	5	4	2	6	7	3
-----	---	---	---	---	---	---	---	---

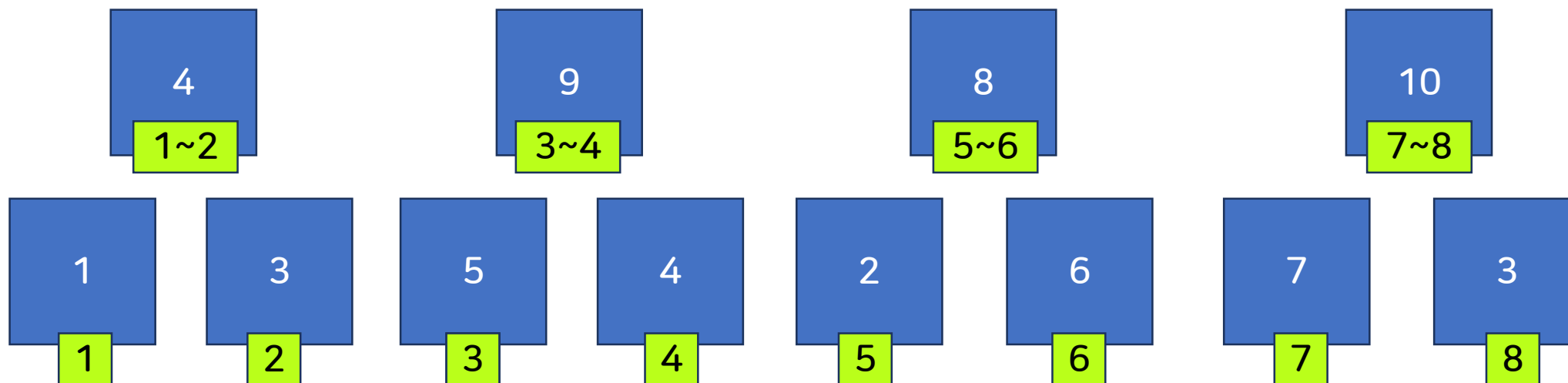
앞에서 이용한 방법 말고 구간합을 구하는 방법이 있을까?

# 도입



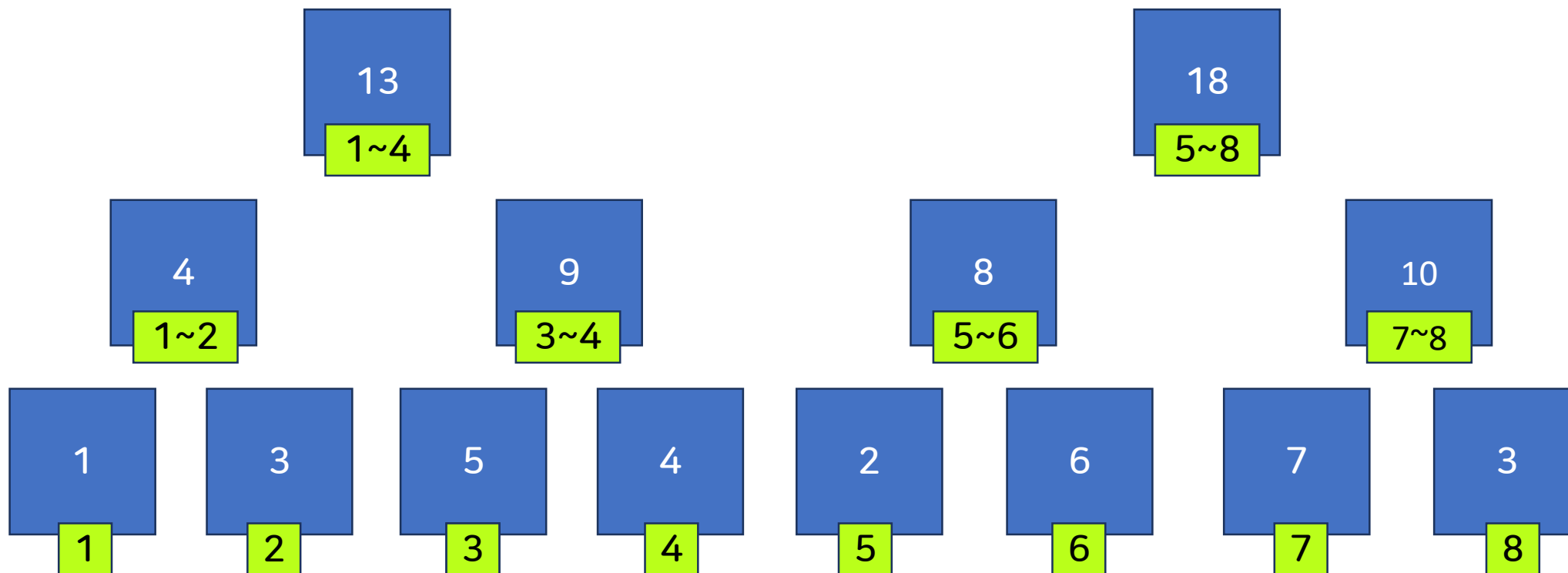
# 도입

2개씩 묶어서 더한다!



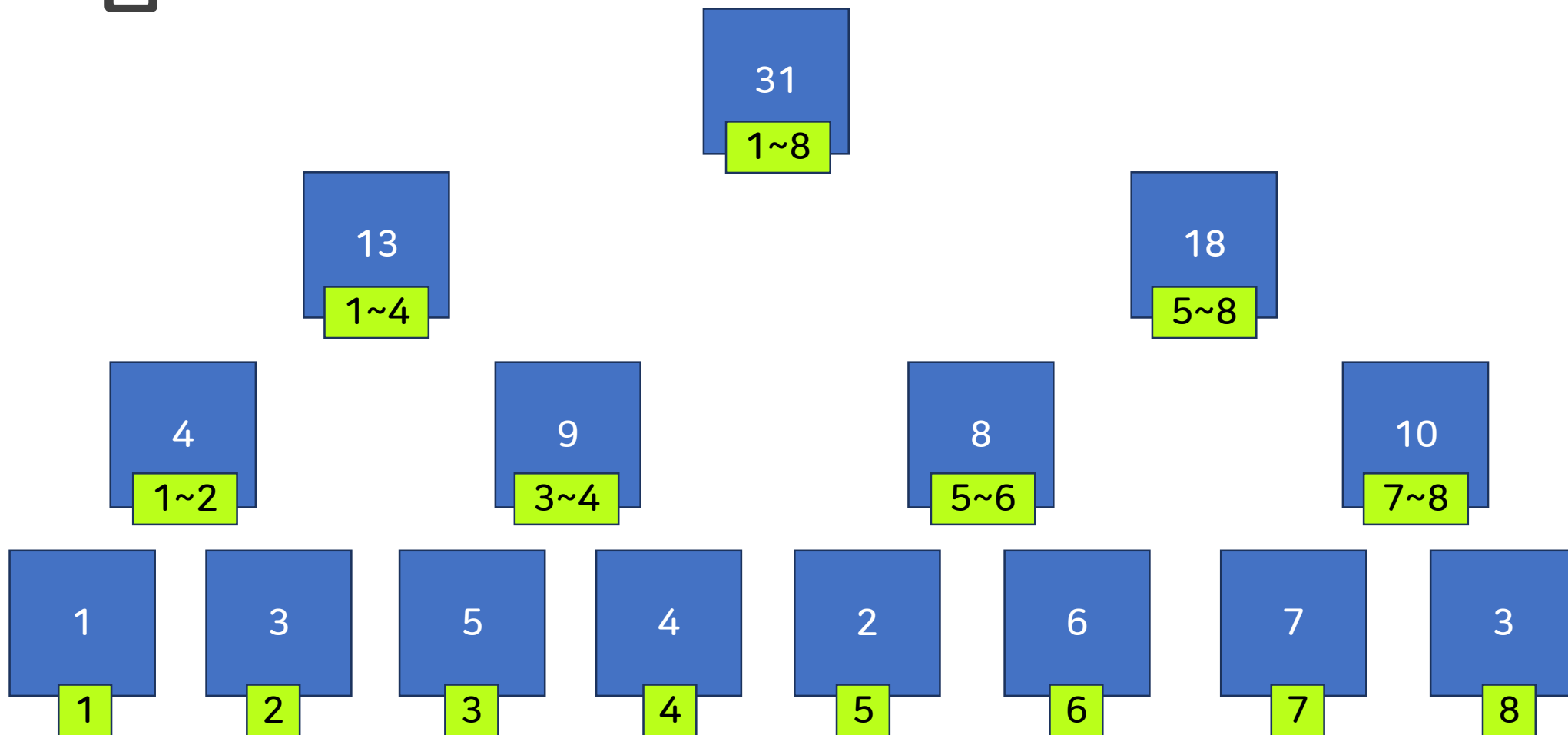
# 도입

2개씩 묶어서 더한다!



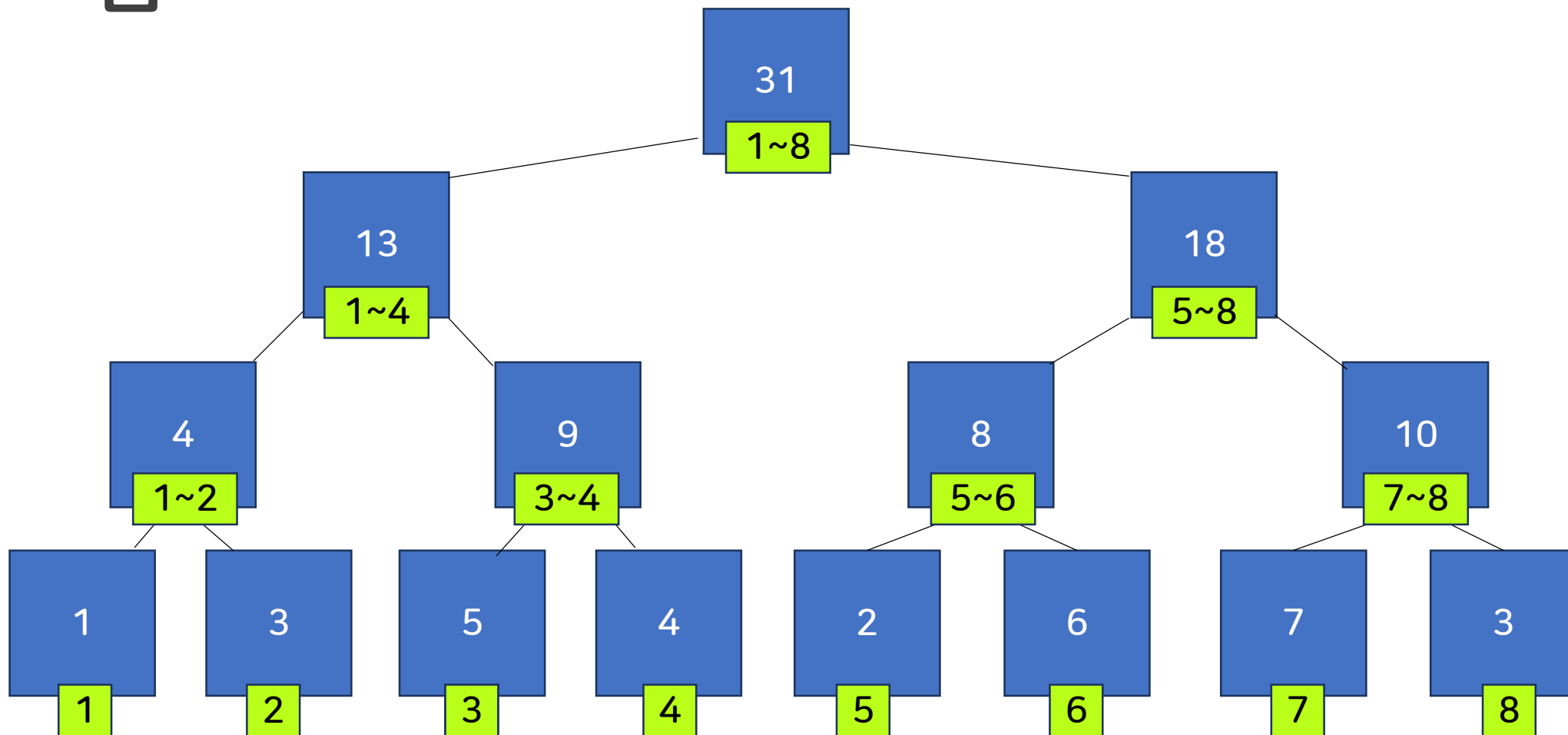
# 도입

2개씩 묶어서 더한다!



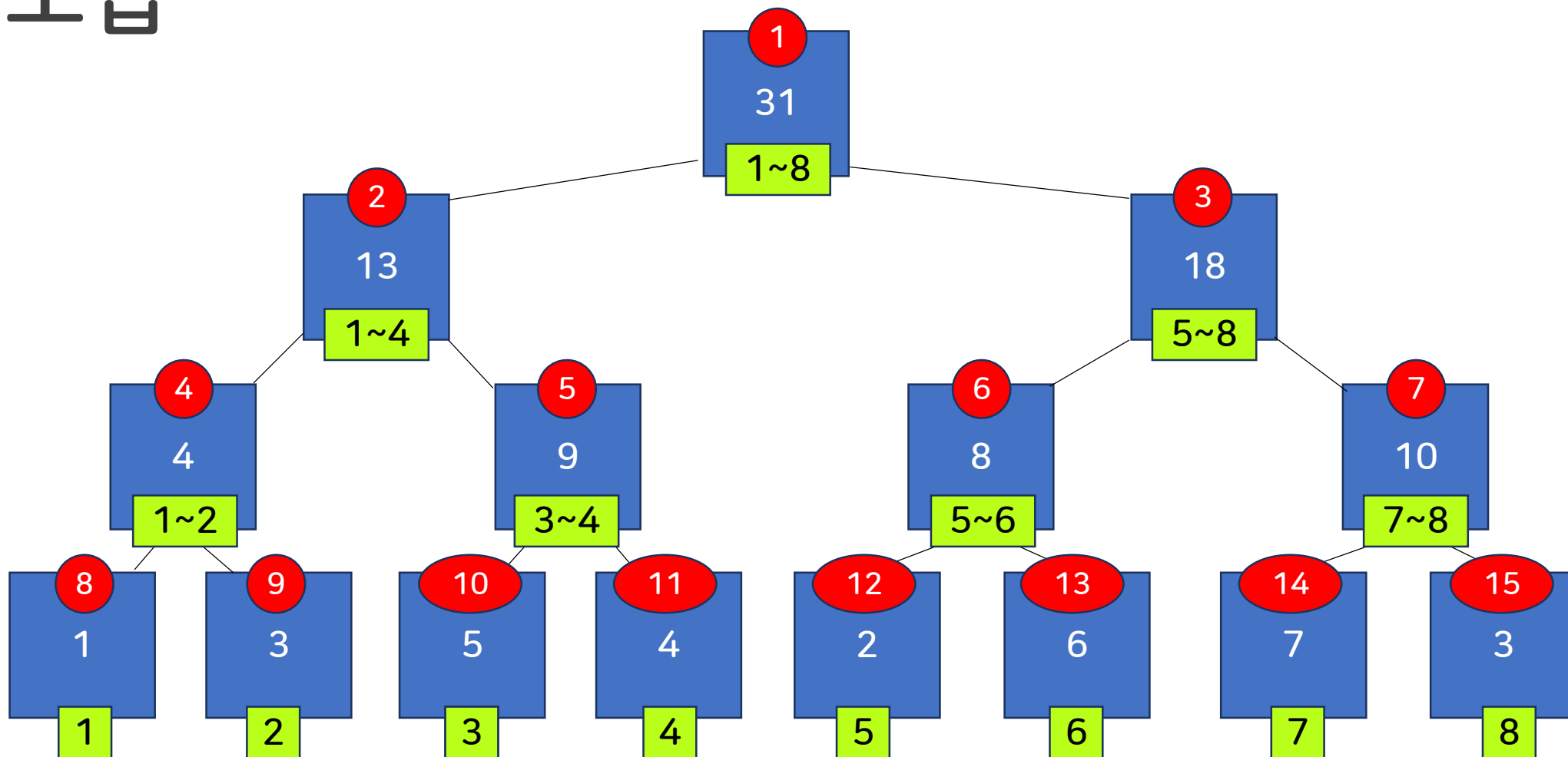
# 도입

다음과 같은 트리 모양이 나온다.



# 도입

빨간 원의 숫자는 segment tree 배열의 index를 뜻한다.





# 도입

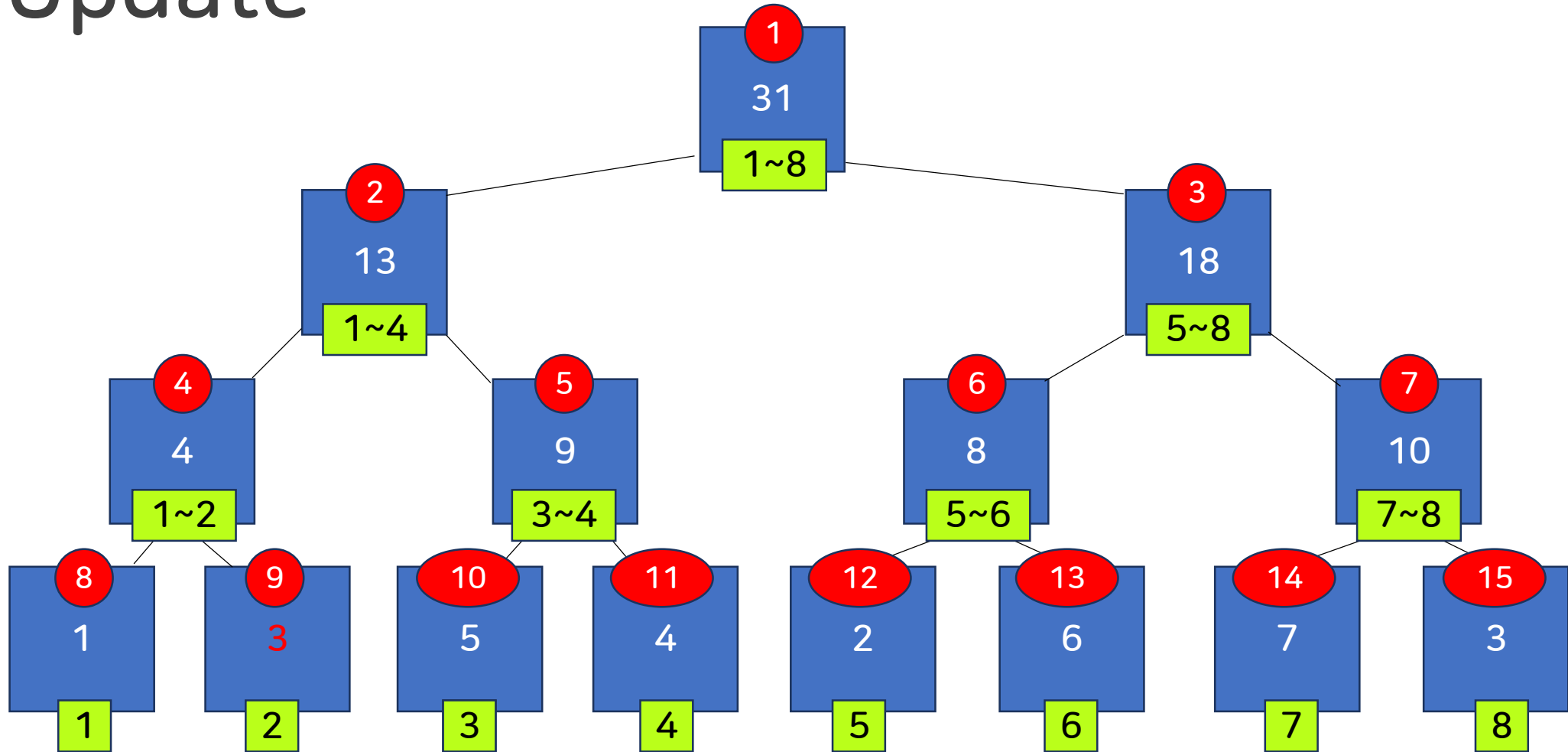
index	1	2	3	4	5	6	7
tree	31	13	18	4	9	8	10

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

이런 모양의 배열이 된다

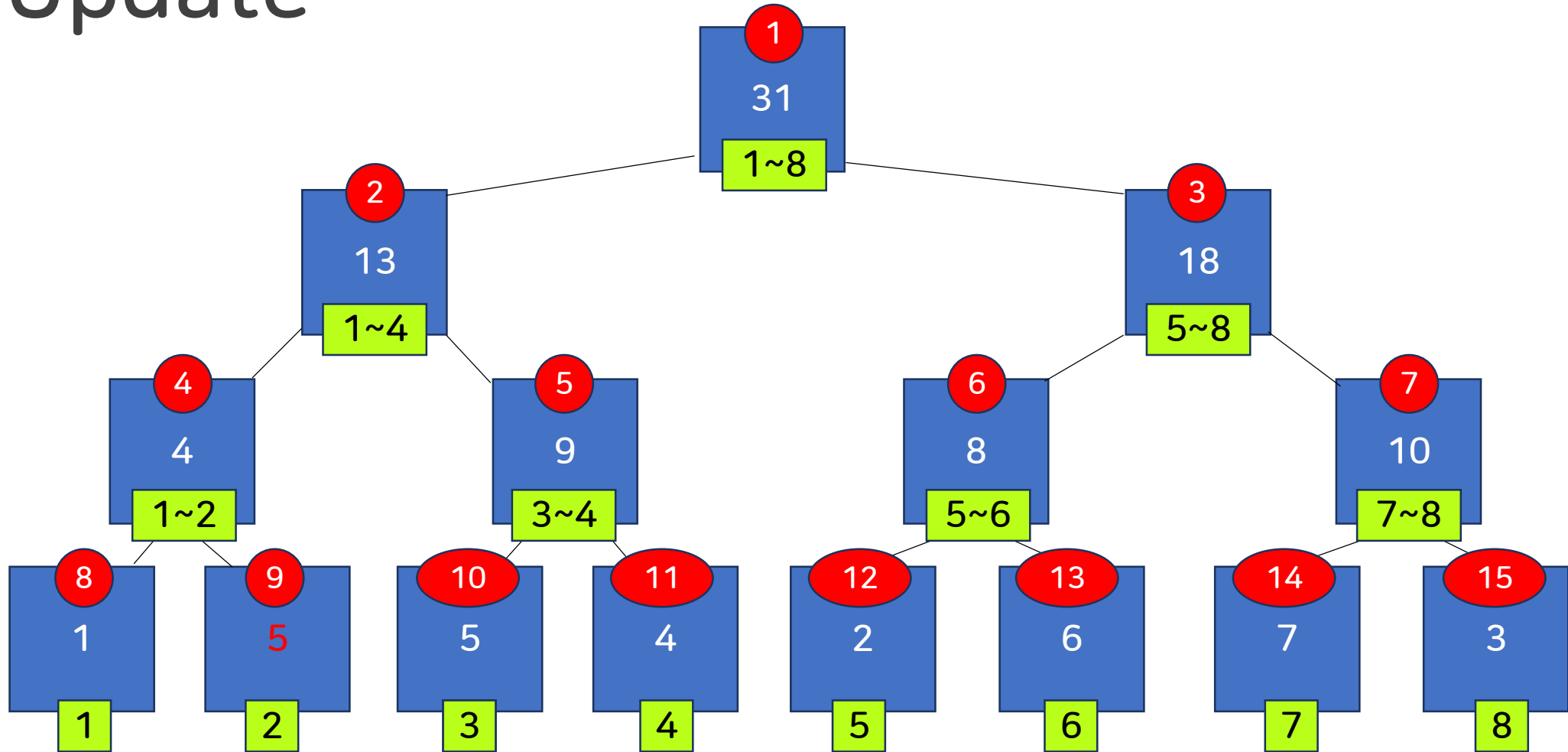
# Update

만약 2번 index에 저장된 값이 변한다면????



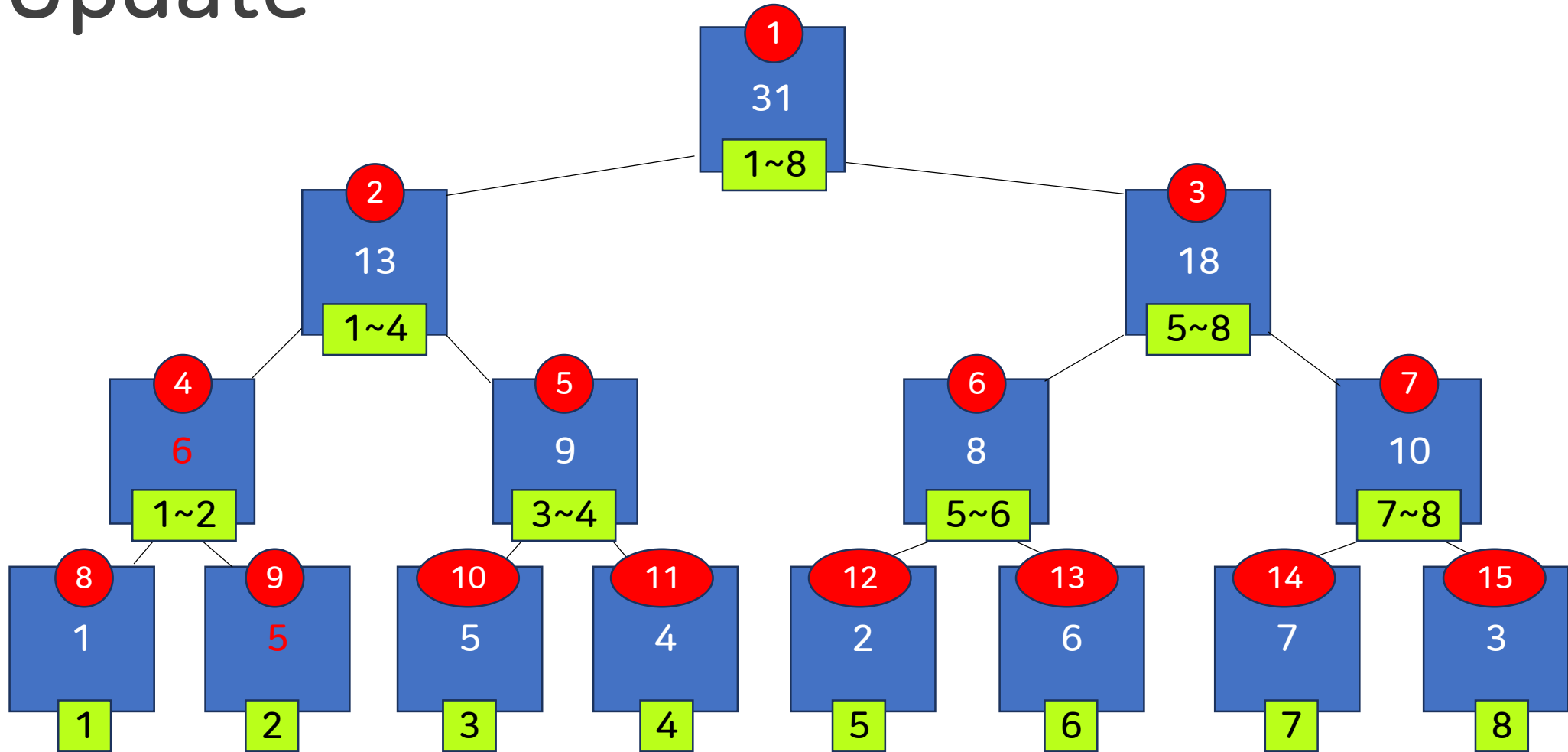
# Update

만약 2번 index에 저장된 값이 변한다면????



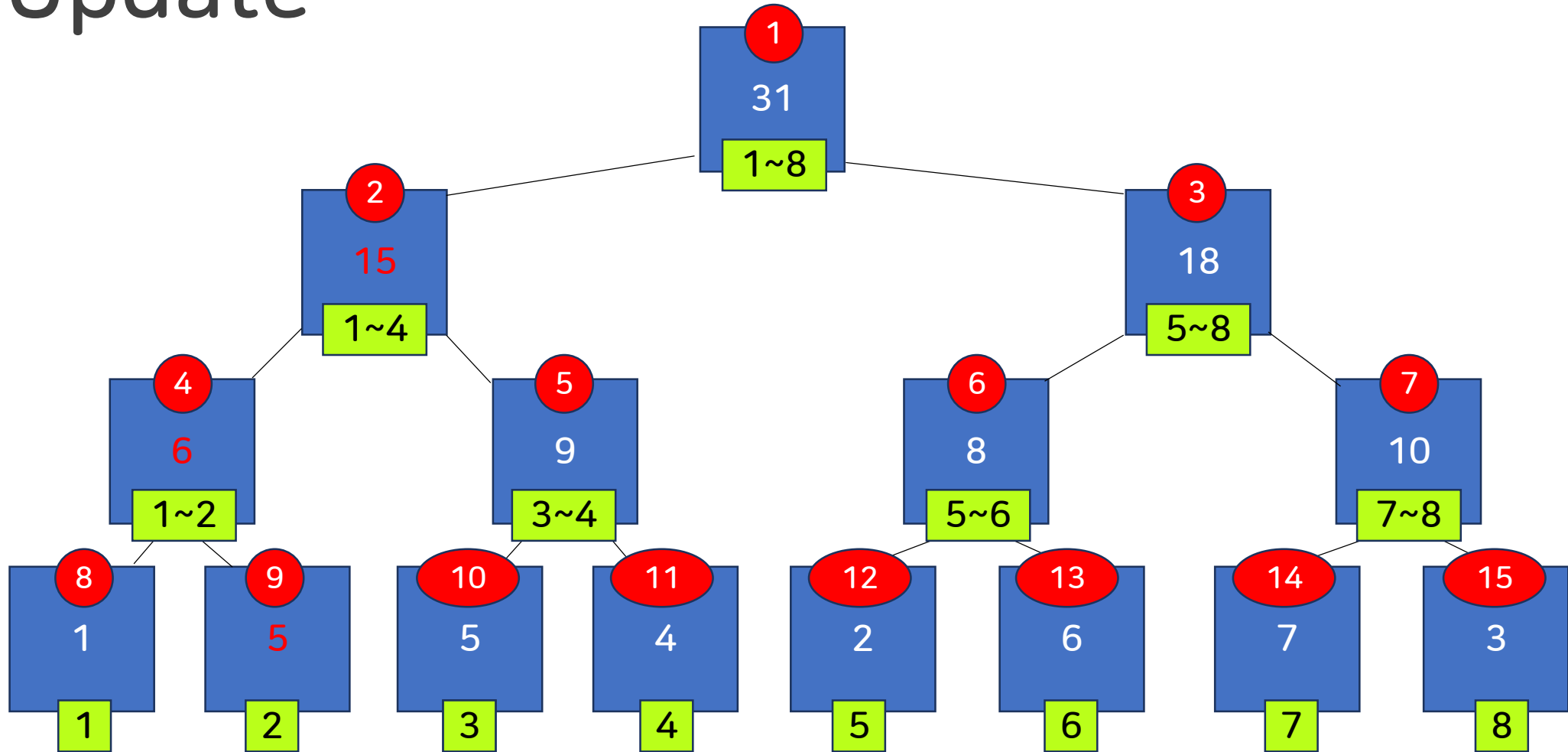
# Update

만약 2번 index에 저장된 값이 변한다면????



# Update

만약 2번 index에 저장된 값이 변한다면????

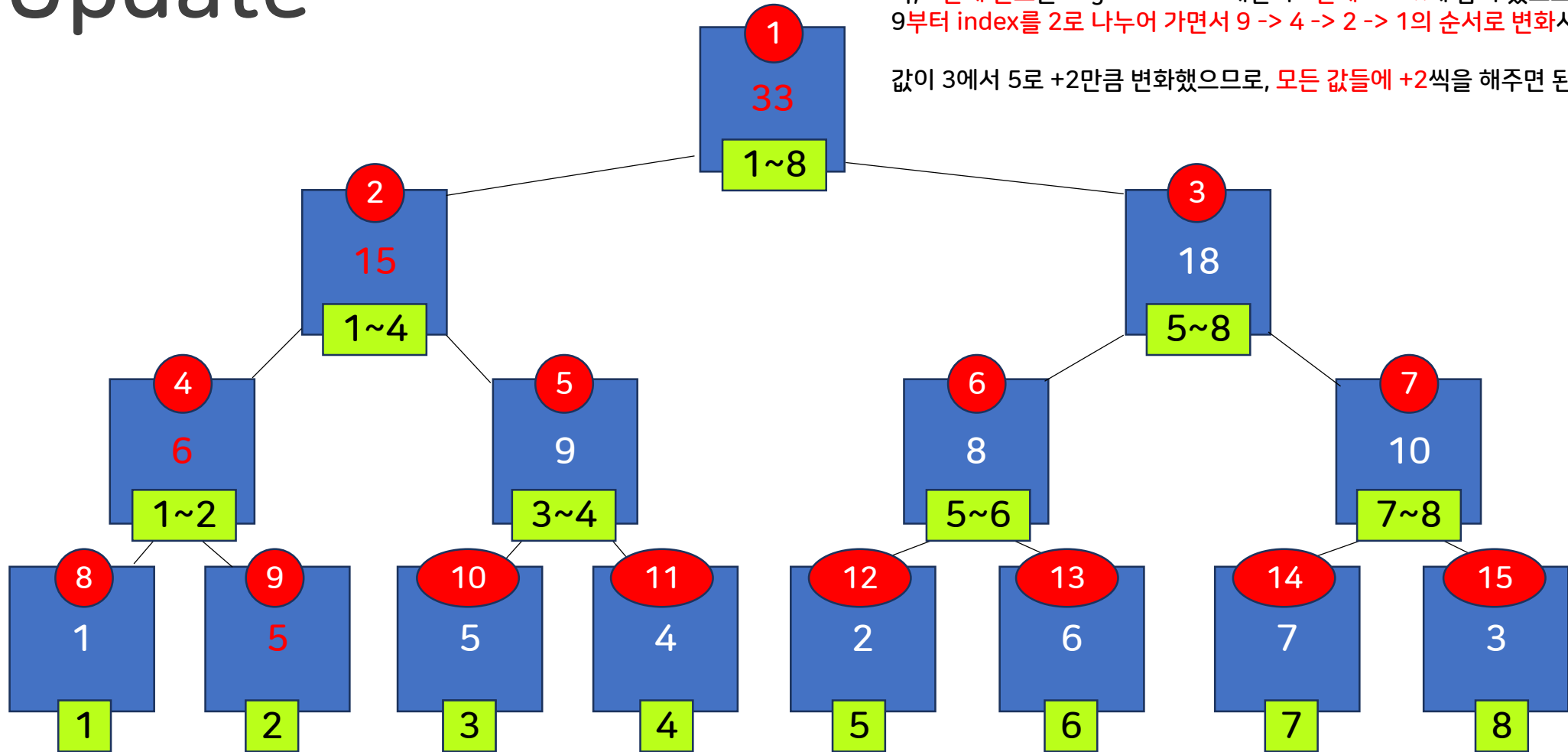


# Update

부모 node의 값들만 변화시키면 된다!

즉, 2번째 원소는 segment tree 배열의 9번째 index에 담겨 있으므로,  
9부터 index를 2로 나누어 가면서 9 -> 4 -> 2 -> 1의 순서로 변화시켜주면 된다.

값이 3에서 5로 +2만큼 변화했으므로, 모든 값들에 +2씩을 해주면 된다!



# 구현

```
class segtree {  
public:  
    vector<ll> tree;  
    int size;
```

**class**를 이용해서 구현해보자!

먼저 segment tree 배열인 **vector<long long> tree**를 선언해주고,  
배열의 크기인 **size**를 선언해준다.

# 구현

```
segtree(int n)
{
    for (size = 1; size < n; size *= 2);
    tree.resize(2 * size);
}
```

생성자를 만들어준다. 여기서 인자인  $n$ 은 배열의 크기이다.  
위의 예제에서는  $n$ 이 8이 된다.



# 구현

```
segtree(int n)
{
    for (size = 1; size < n; size *= 2);
    tree.resize(2 * size);
}
```

for문은 n 이상의 가장 작은 2의 배수를 구하는 과정이다.

예제에서 보았듯이 segment tree는 full binary tree이므로,

2의 배수 개 만큼의 초기 배열의 크기가 필요한데, 그것을 size에 저장해준다.

예를 들어, 만약 n이 6이었다면, size는 8이 될 것이고, 7번, 8번 index에 들어있는 값을 0으로 해주면 아무 문제가 없다.

# 구현

```
segtree(int n)
{
    for (size = 1; size < n; size *= 2);
    tree.resize(2 * size);
}
```

그 후 tree vector의 크기를 넉넉히  $2 * \text{size}$ 로 해주게 된다면 틀은 다 잡히게 된다.  
위의 예제에서는 size가 8이고, 16칸 짜리 segment tree 배열이 만들어지므로 충분하다.

왜  $2 * \text{size}$ 인지는 한번 생각해보자.

# 구현

```
void update(int pos, int x)
{
    //원래 index의 tree배열 상의 위치를 pos에 저장한다.
    int index = size + pos - 1;
    //원래 index의 값과 새로 업데이트할 값의 차이를 x에 저장한다.
    int u = x - tree[index];
    //pos의 위치에 있는 값을 포함하는 모든 구간합의 정보를 업데이트한다
    while (index)
    {
        tree[index] += u;
        index /= 2;
    }
}
```

update의 함수를 만들어 보았다.

위 예제에서 볼 때에는 원래는 3이 들어있던 2번째 index의 값을 5로 바꾸어 주는 것이었으므로, **pos는 2가 되고, x는 5가 된다.**

# 구현

```
void update(int pos, ll x)
{
    //원래 index의 tree배열 상의 위치를 pos에 저장한다.
    int index = size + pos - 1;
    //원래 index의 값과 새로 업데이트할 값의 차이를 x에 저장한다.
    int u = x - tree[index];
    //pos의 위치에 있는 값을 포함하는 모든 구간합의 정보를 업데이트한다
    while (index)
    {
        tree[index] += u;
        index /= 2;
    }
}
```

tree 배열 상에서의 위치를 index에 저장한다.

예제에서는 pos가 2였으므로,

$\text{index} = 8 + 2 - 1 = 9$ 가 되어 3이 담겨 있는 index를 저장하게 된다.

# 구현

```
void update(int pos, int x)
{
    //원래 index의 tree배열 상의 위치를 pos에 저장한다.
    int index = size + pos - 1;
    //원래 index의 값과 새로 업데이트할 값의 차이를 x에 저장한다.
    int u = x - tree[index];
    //pos의 위치에 있는 값을 포함하는 모든 구간합의 정보를 업데이트한다
    while (index)
    {
        tree[index] += u;
        index /= 2;
    }
}
```

u에 변화되는 값을 저장한다.

예제에서는 3에서 5로 값이 바뀌었으므로, +2만큼 값이 변하게 된다.

따라서  $u = x - \text{tree}[9] = 5 - 3 = 2$ 가 된다.

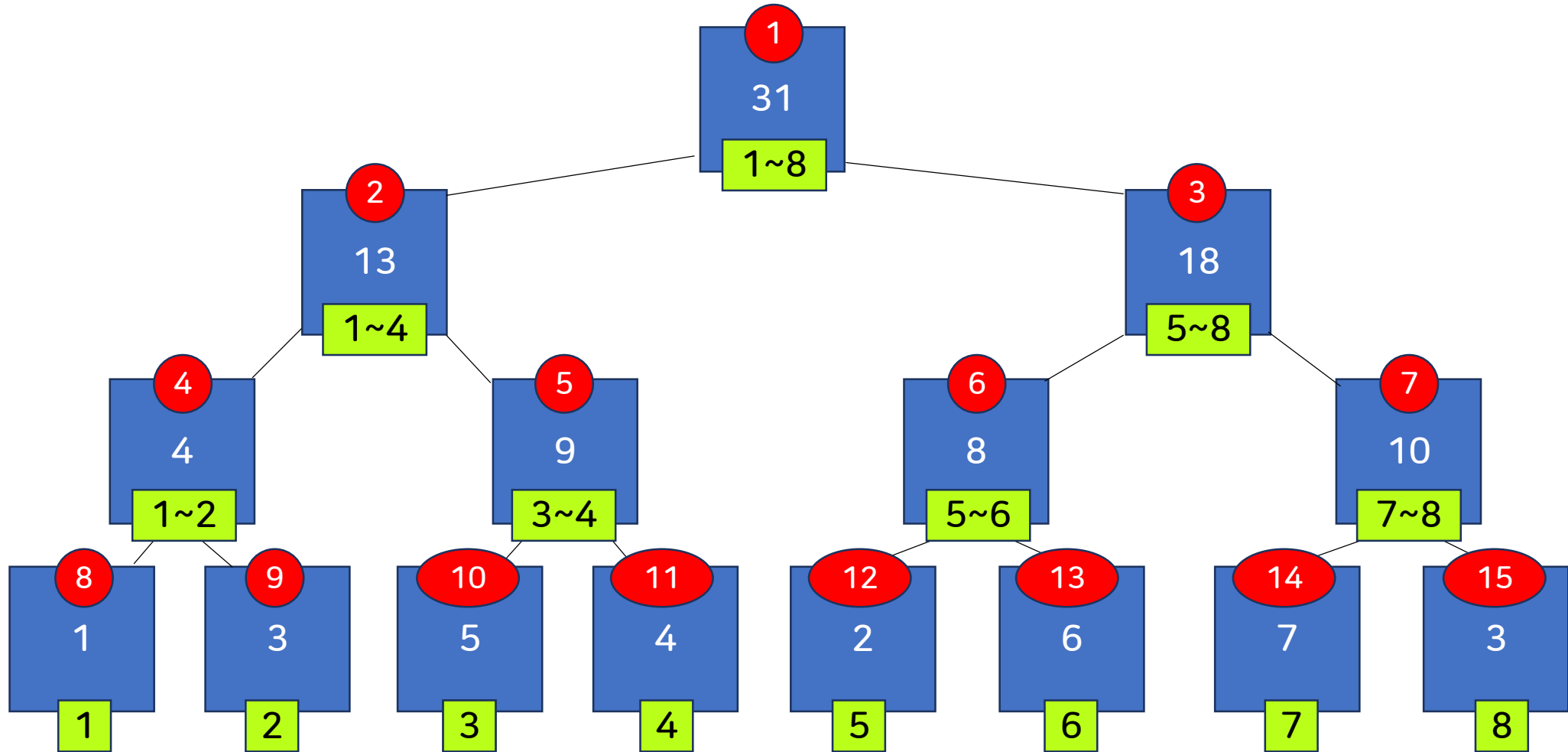
# 구현

```
void update(int pos, ll x)
{
    //원래 index의 tree배열 상의 위치를 pos에 저장한다.
    int index = size + pos - 1;
    //원래 index의 값과 새로 업데이트할 값의 차이를 x에 저장한다.
    int u = x - tree[index];
    //pos의 위치에 있는 값을 포함하는 모든 구간합의 정보를 업데이트한다
    while (index)
    {
        tree[index] += u;
        index /= 2;
    }
}
```

while문을 돌면서 2번째 index와  
그 구간합을 저장한 tree의 원소 값을 모두 update시켜준다.  
즉, 9 -> 4 -> 2 -> 1번째 값에 모두 2씩을 더해준다.

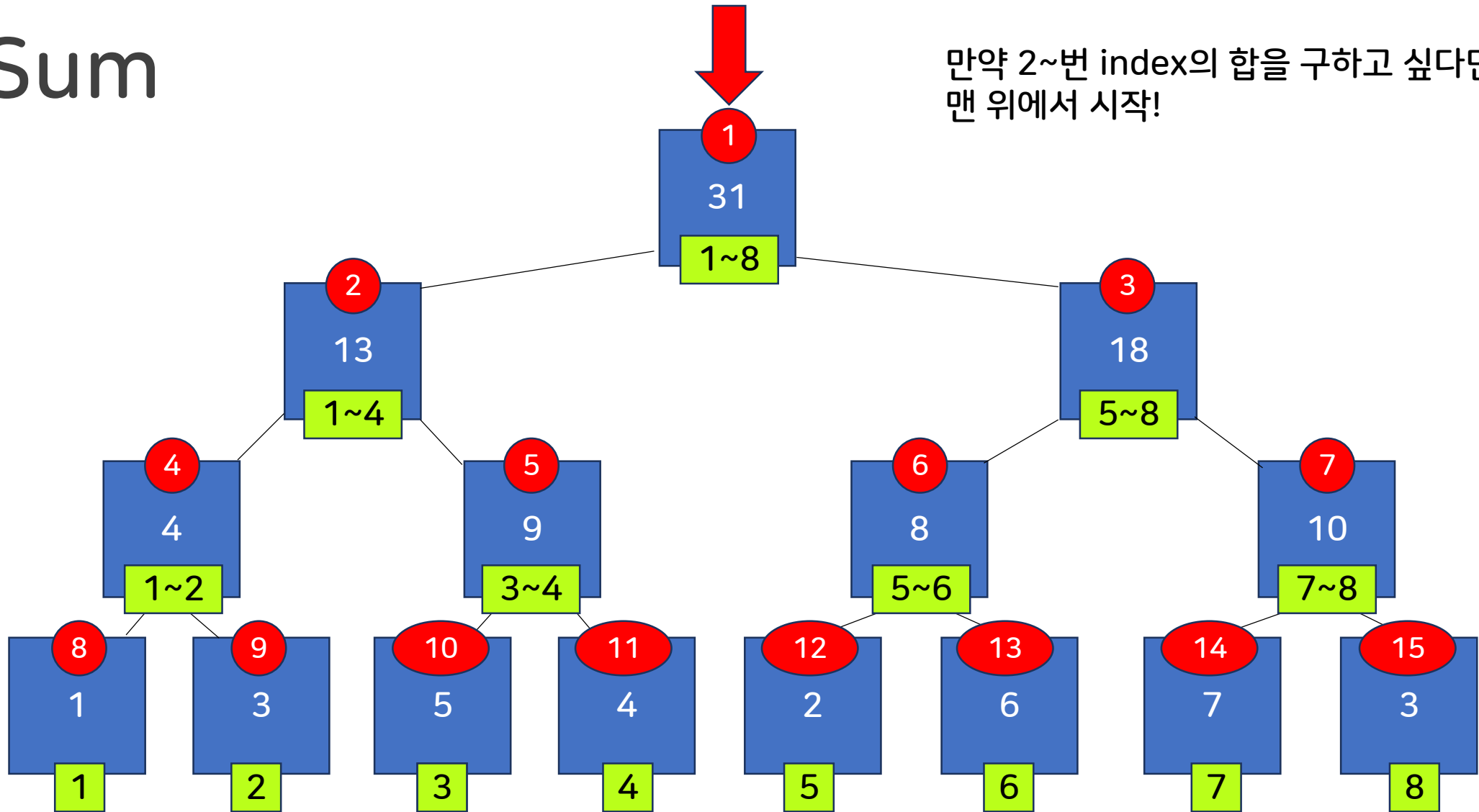
# Sum

만약 2~번 index의 합을 구하고 싶다면?



# Sum

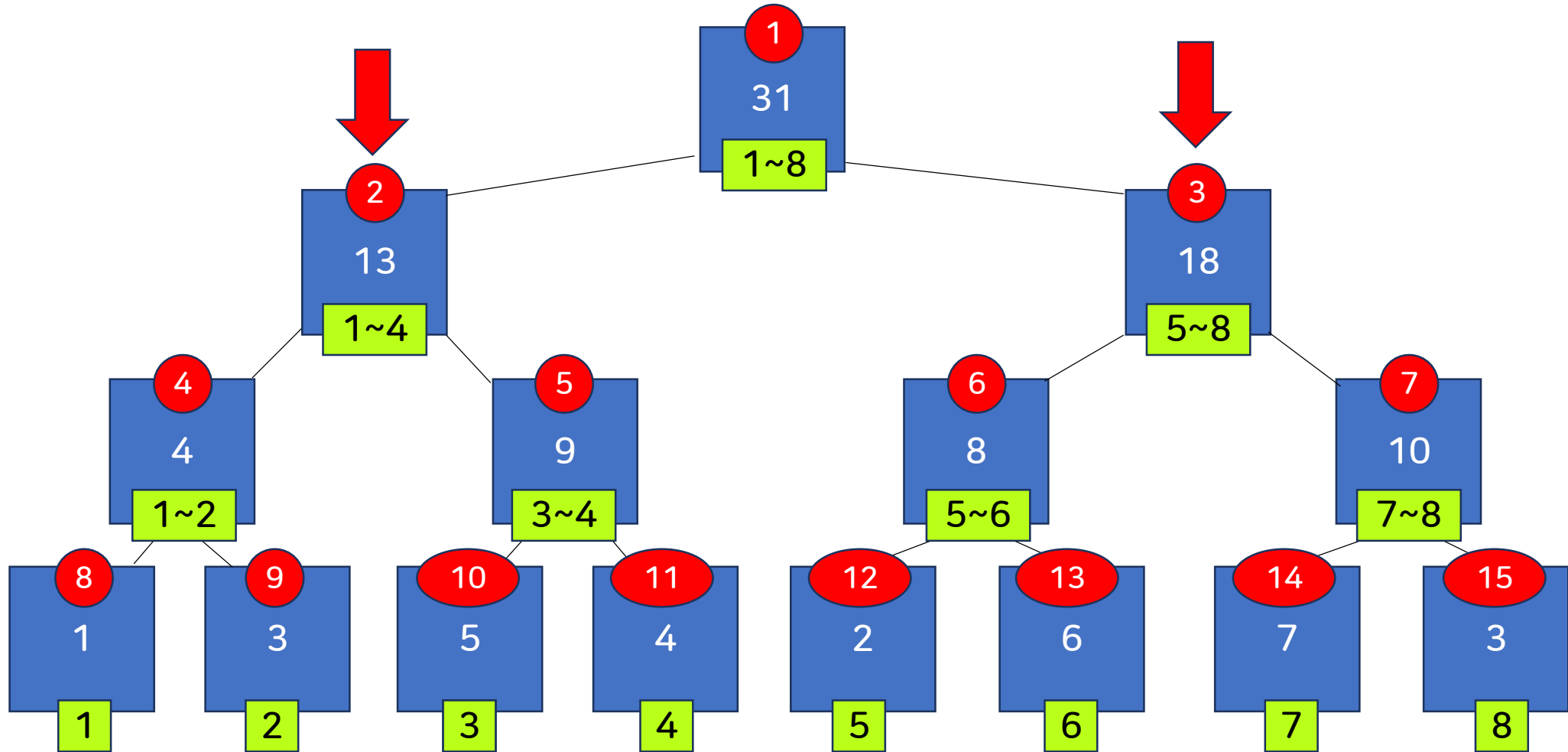
만약 2~번 index의 합을 구하고 싶다면?  
맨 위에서 시작!





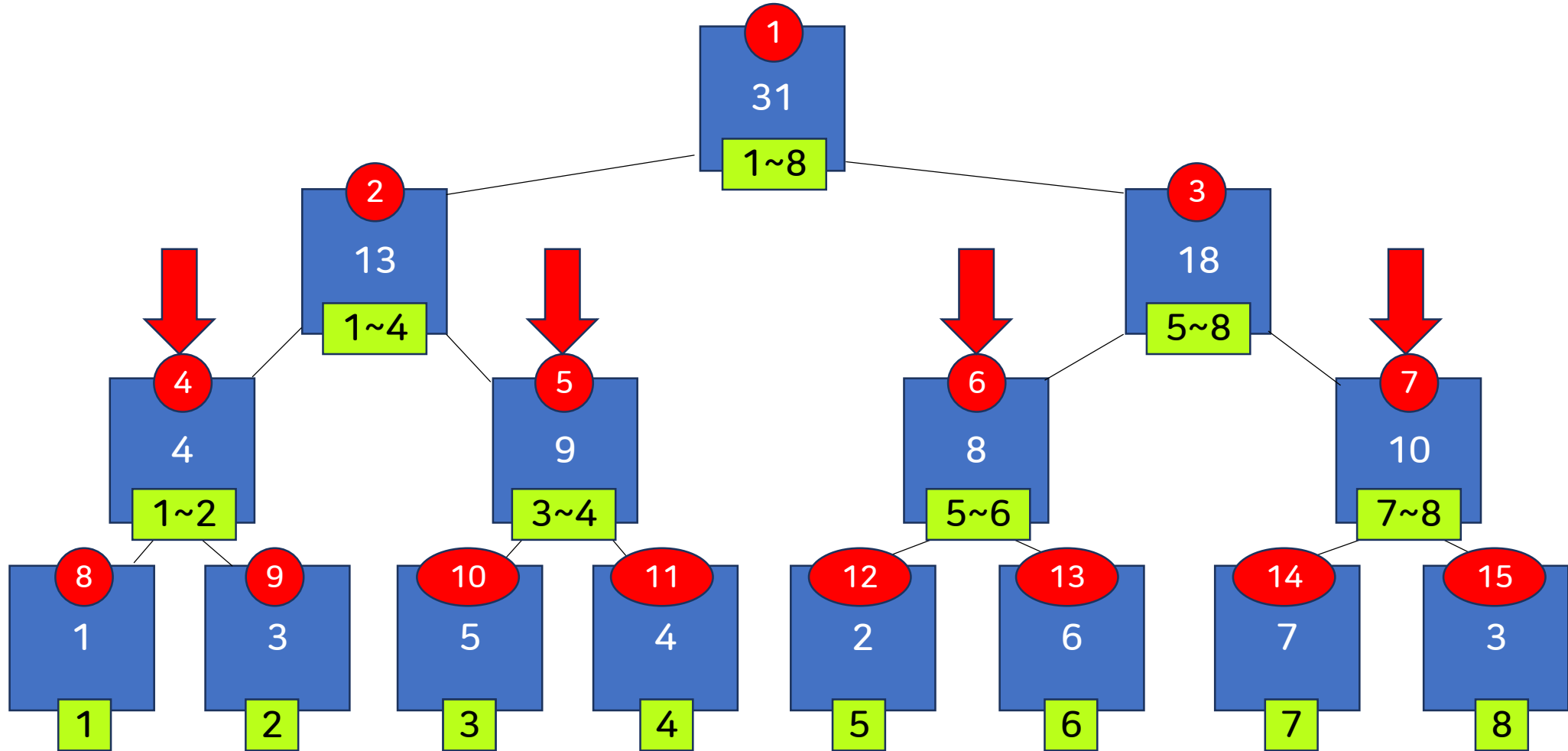
# Sum

만약 2~번 index의 합을 구하고 싶다면?  
1~8이 2~5에 들어가지 않으므로, 양 옆을 모두 탐색!



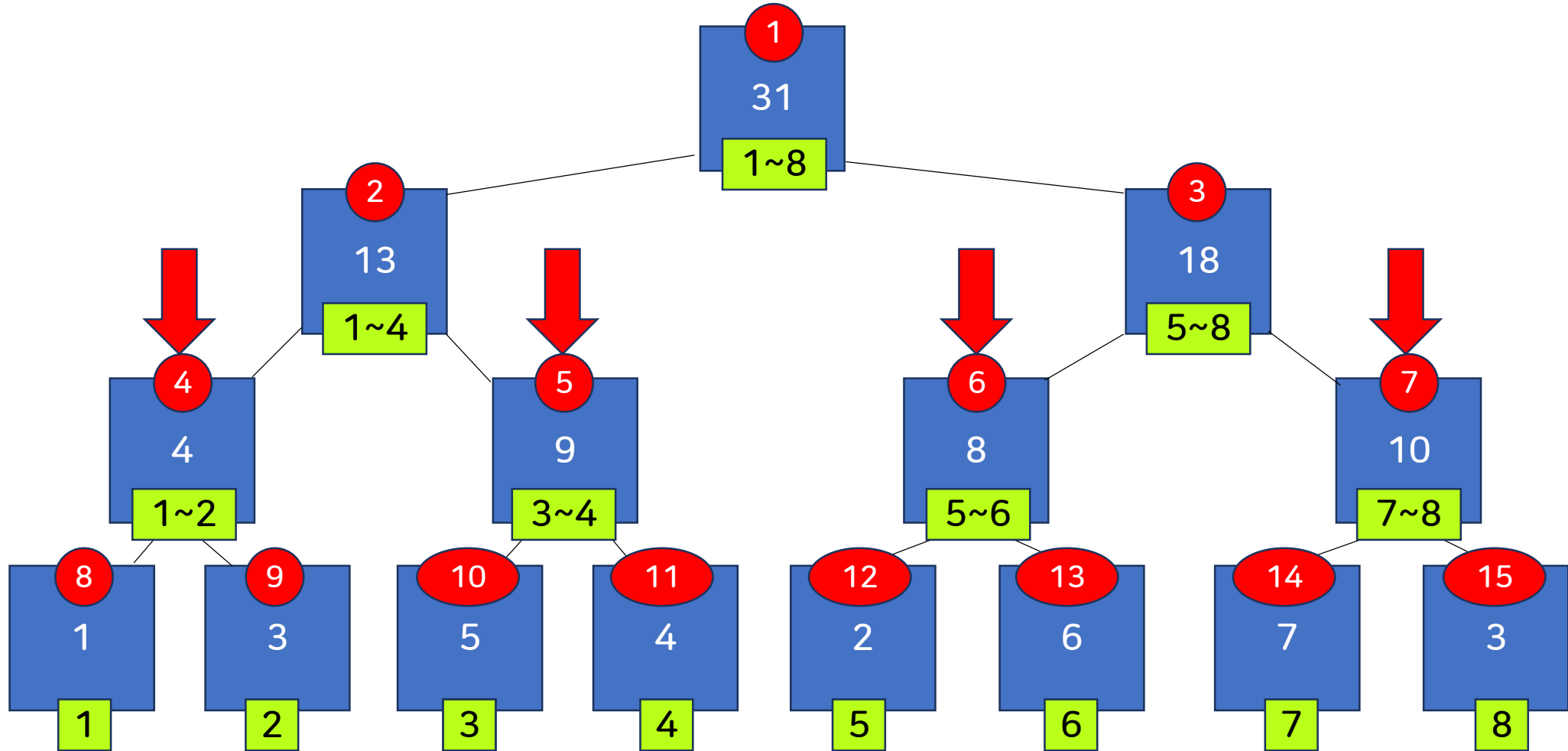
# Sum

만약 2~번 index의 합을 구하고 싶다면?  
1~4, 5~8 모두 2~5에 포함되지 않고, 겹치는 부분이 있으므로,  
둘 다 양 옆을 모두 탐색!



# Sum

만약 2~번 index의 합을 구하고 싶다면?  
1~4, 5~8 모두 2~5에 포함되지 않고, 겹치는 부분이 있으므로,  
둘 다 양 옆을 모두 탐색!



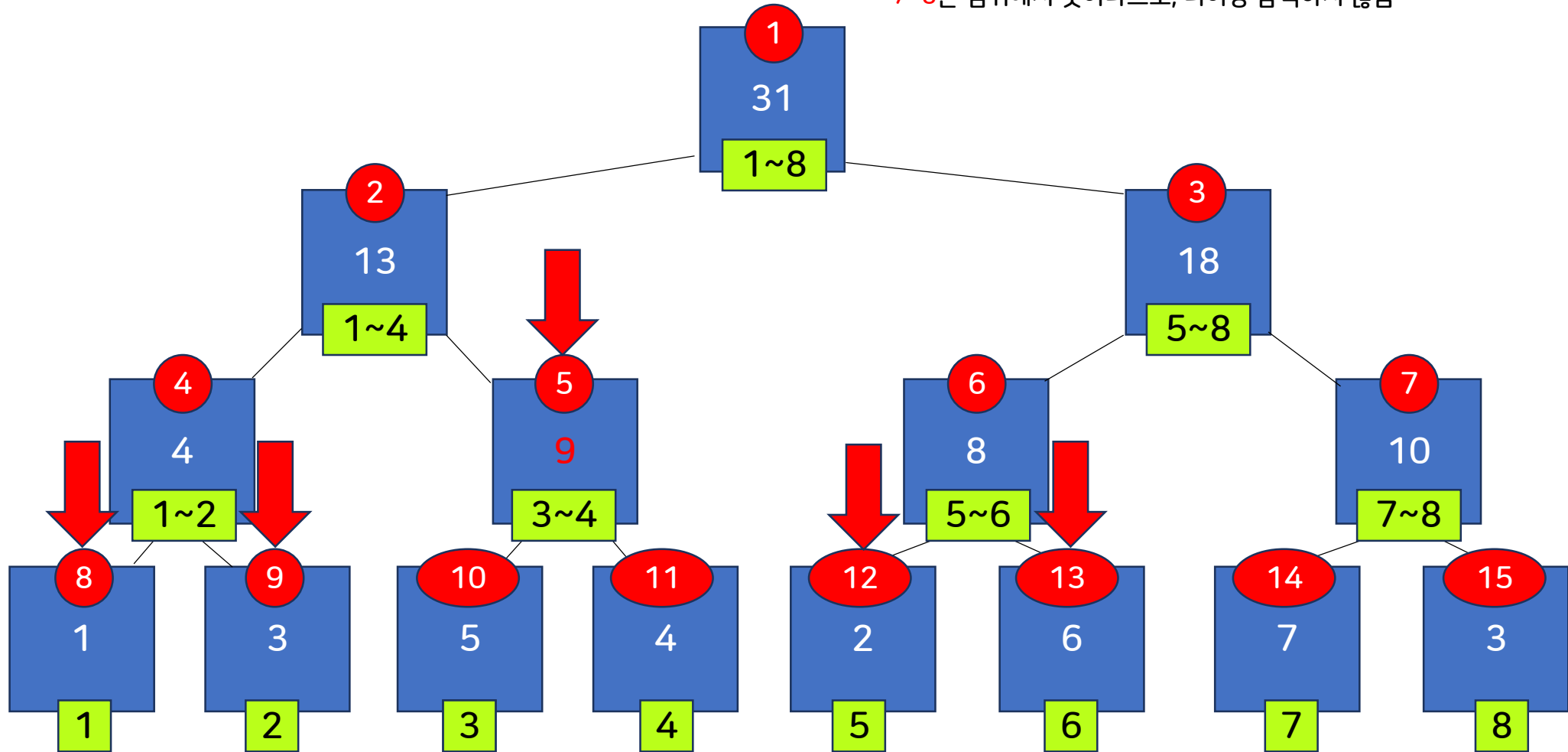
# Sum

만약 2~5 index의 합을 구하고 싶다면???

1~2, 5~6은 2~5에 포함되지 않고, 겹치는 부분이 있으므로, 둘 다 양 옆을 모두 탐색

3~4는 2~5에 포함되므로, 그 값을 return

7~8은 범위에서 벗어나므로, 더이상 탐색하지 않음

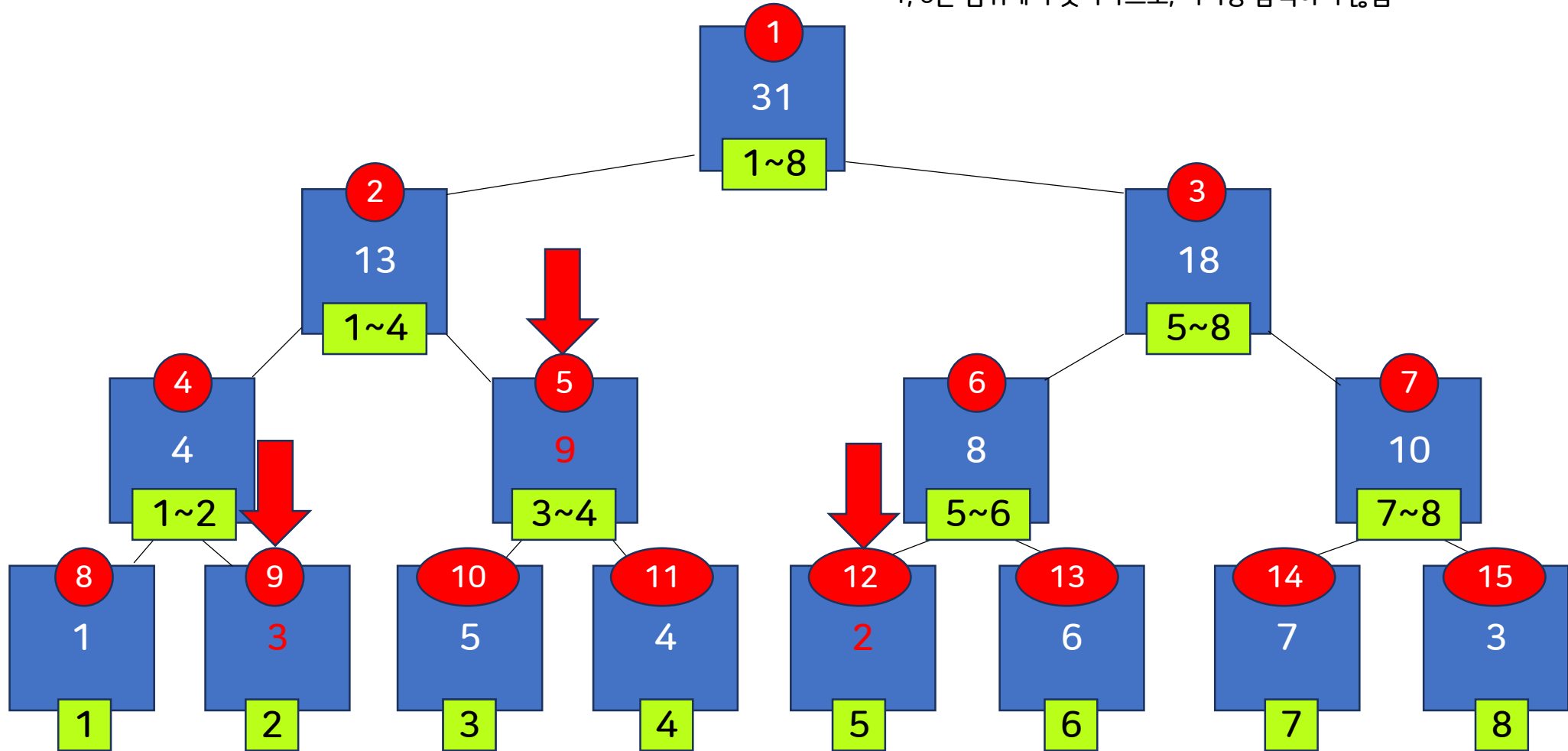


# Sum

만약 2~5 index의 합을 구하고 싶다면???

2, 5는 2~5에 포함되므로, 그 값을 return

1, 6은 범위에서 벗어나므로, 더이상 탐색하지 않음



# 구현

```
|| getsum(int pos, int left, int right, int start, int end)
{
    //left, right가 내가 찾고자 하는 범위, pos가 내가 탐색중인 tree의 index, start, end가 이 index에서의 범위
    //만약 지금 탐색중인 위치가 내가 탐색하고자하는 left~right 범위를 벗어난다면, 0을 return해준다.
    if (right < start || left > end)
        return 0;
    //만약 지금 탐색중인 위치가 내가 탐색하고자 하는 left~right 범위 내에 포함되어있으면, 이 값을 return해준다.
    if (start >= left && end <= right)
        return tree[pos];
    //만약 지금 탐색중인 위치가 내가 탐색하고자 하는 left~right 범위와 겹치지만 포함되지는 않으면,
    //자식 index를 둘 다 탐색한 후 return값을 더해준다
    int mid = (start + end) / 2;
    return getsum(pos * 2, left, right, start, mid) + getsum(pos * 2 + 1, left, right, mid + 1, end);
}
```

left, right는 내가 찾고자 하는 범위, pos는 내가 탐색중인 tree의 index,  
start, end는 이 index에서의 범위를 나타낸다.

기본적으로 top-down으로 탐색하므로, pos는 1 그리고 start와 end는 시작과 끝 index로 시작한다.  
즉, 예시로 본다면 pos는 1, start는 1, end는 8, left는 2, right는 5가 되게 된다.

# 구현

```
|| getsum(int pos, int left, int right, int start, int end)
{
    //left, right가 내가 찾고자 하는 범위, pos가 내가 탐색중인 tree의 index, start, end가 이 index에서의 범위
    //만약 지금 탐색중인 위치가 내가 탐색하고자하는 left~right 범위를 벗어난다면, 0을 return해준다.
    if (right < start || left > end)
        return 0;
    //만약 지금 탐색중인 위치가 내가 탐색하고자 하는 left~right 범위 내에 포함되어있으면, 이 값을 return해준다.
    if (start >= left && end <= right)
        return tree[pos];
    //만약 지금 탐색중인 위치가 내가 탐색하고자 하는 left~right 범위와 겹치지만 포함되지는 않으면,
    //자식 index를 둘 다 탐색한 후 return값을 더해준다
    int mid = (start + end) / 2;
    return getsum(pos * 2, left, right, start, mid) + getsum(pos * 2 + 1, left, right, mid + 1, end);
}
```

탐색중인 위치가 내가 탐색하고자 하는 **left~right 범위를 벗어난다면,**

0을 return하고 탐색을 종료한다.

예를 들면, 2~5 구간합을 구하는데, 지금 7~8을 탐색 중이라면, 0을 return하고 탐색을 종료하게 된다.

# 구현

```
int getsum(int pos, int left, int right, int start, int end)
{
    //left, right가 내가 찾고자 하는 범위, pos가 내가 탐색중인 tree의 index, start, end가 이 index에서의 범위
    //만약 지금 탐색중인 위치가 내가 탐색하고자하는 left~right 범위를 벗어난다면, 0을 return해준다.
    if (right < start || left > end)
        return 0;
    //만약 지금 탐색중인 위치가 내가 탐색하고자 하는 left~right 범위 내에 포함되어있으면, 이 값을 return해준다.
    if (start >= left && end <= right)
        return tree[pos];
    //만약 지금 탐색중인 위치가 내가 탐색하고자 하는 left~right 범위와 겹치지만 포함되지는 않으면,
    //자식 index를 둘 다 탐색한 후 return값을 더해준다
    int mid = (start + end) / 2;
    return getsum(pos * 2, left, right, start, mid) + getsum(pos * 2 + 1, left, right, mid + 1, end);
}
```

탐색중인 위치가 내가 탐색하고자 하는 **left~right 범위 내에 포함된다면,**

이 tree의 index에 저장된 값을 return해준다.

예를 들어 2~5의 구간합을 구할 때 3~4 구간을 탐색 중이라면, 이 값을 return해준 뒤 탐색을 종료한다.



# 구현

```
|| getsum(int pos, int left, int right, int start, int end)
{
    //left, right가 내가 찾고자 하는 범위, pos가 내가 탐색중인 tree의 index, start, end가 이 index에서의 범위
    //만약 지금 탐색중인 위치가 내가 탐색하고자하는 left~right 범위를 벗어난다면, 0을 return해준다.
    if (right < start || left > end)
        return 0;
    //만약 지금 탐색중인 위치가 내가 탐색하고자 하는 left~right 범위 내에 포함되어있으면, 이 값을 return해준다.
    if (start >= left && end <= right)
        return tree[pos];
    //만약 지금 탐색중인 위치가 내가 탐색하고자 하는 left~right 범위와 겹치지만 포함되지는 않으면,
    //자식 index를 둘 다 탐색한 후 return값을 더해준다
    int mid = (start + end) / 2;
    return getsum(pos * 2, left, right, start, mid) + getsum(pos * 2 + 1, left, right, mid + 1, end);
}
```

탐색중인 위치가 내가 구간합을 구하고자 하는 **left~right 범위와 겹치지만 포함되지는 않으면**,

자식 index를 둘 다 탐색한 후 더해준 값을 return해준다.

예를 들어, 2~5의 구간합을 구하는데 1~4의 구간을 탐색 중이라면 1~2, 3~4를 모두 탐색한 후 return값을 더해서 return해준다.

# 전체 코드

```
class segtree {
public:
    vector<ll> tree;
    int size;
    segtree(int n)
    {
        for (size = 1; size < n; size *= 2);
        tree.resize(2 * size);
    }
    void update(int pos, ll x)
    {
        //원래 index의 tree배열 상의 위치를 pos에 저장한다.
        int index = size + pos - 1;
        //원래 index의 값과 새로 업데이트할 값의 차이를 x에 저장한다.
        int u = x - tree[index];
        //pos의 위치에 있는 값을 포함하는 모든 구간합의 정보를 업데이트한다
        while (index)
        {
            tree[index] += u;
            index /= 2;
        }
    }
}
```

# 전체 코드

```
,
|| getsum(int pos, int left, int right, int start, int end)
{
    //left, right가 내가 찾고자 하는 범위, pos가 내가 탐색중인 tree의 index, start, end가 이 index에서의 범위
    //만약 지금 탐색중인 위치가 내가 탐색하고자하는 left~right 범위를 벗어난다면, 0을 return해준다.
    if (right < start || left > end)
        return 0;
    //만약 지금 탐색중인 위치가 내가 탐색하고자 하는 left~right 범위 내에 포함되어있으면, 이 값을 return해준다.
    if (start >= left && end <= right)
        return tree[pos];
    //만약 지금 탐색중인 위치가 내가 탐색하고자 하는 left~right 범위와 겹치지만 포함되지는 않으면,
    //자식 index를 둘 다 탐색한 후 return값을 더해준다
    int mid = (start + end) / 2;
    return getsum(pos * 2, left, right, start, mid) + getsum(pos * 2 + 1, left, right, mid + 1, end);
};
```

BOJ 2042 구간 합 구하기

# 구간 합 구하기

한번 스스로 구현해보세요. 뒷 장에 답이 있습니다.

# 구간 합 구하기

```
int main()
{
    int n, m, k;
    scanf("%d%d%d", &n, &m, &k);
    segtree tree(n);
    for (int i = 1; i <= n; i++)
    {
        int x;
        scanf("%d", &x);
        tree.update(i, x);
    }
    int t = m + k;
    while (t-->0)
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        if (a == 1)
            tree.update(b, c);
        else
            printf("%d\n", tree.getsum(1, b, c, 1, tree.size));
    }
}
```

앞에서 구현한 class를 그대로 사용하고,  
main문만 구현하면 됩니다!

BOJ 1275 커피숍2

# 커피숍2

구간 합 구하기와 비슷한 문제입니다



BOJ 2357 최솟값과 최대값

# 최솟값과 최댓값

구간합을 저장한 tree vector 대신에,  
최댓값과 최솟값을 저장하는 min\_tree vector와 max\_tree vector를 만든 뒤,  
getsum 대신에 getmax, getmin 함수를 비슷한 방법으로 구현해보세요!

BOJ 11505 구간 곱 구하기

# 구간 곱 구하기

더하기를 곱하기로만 바꾸면 되는데, update 함수를  
getsum 함수를 구현했던 것처럼 top-down 방식으로 구현해보세요!

# 챕터 2: Fenwick Tree

---

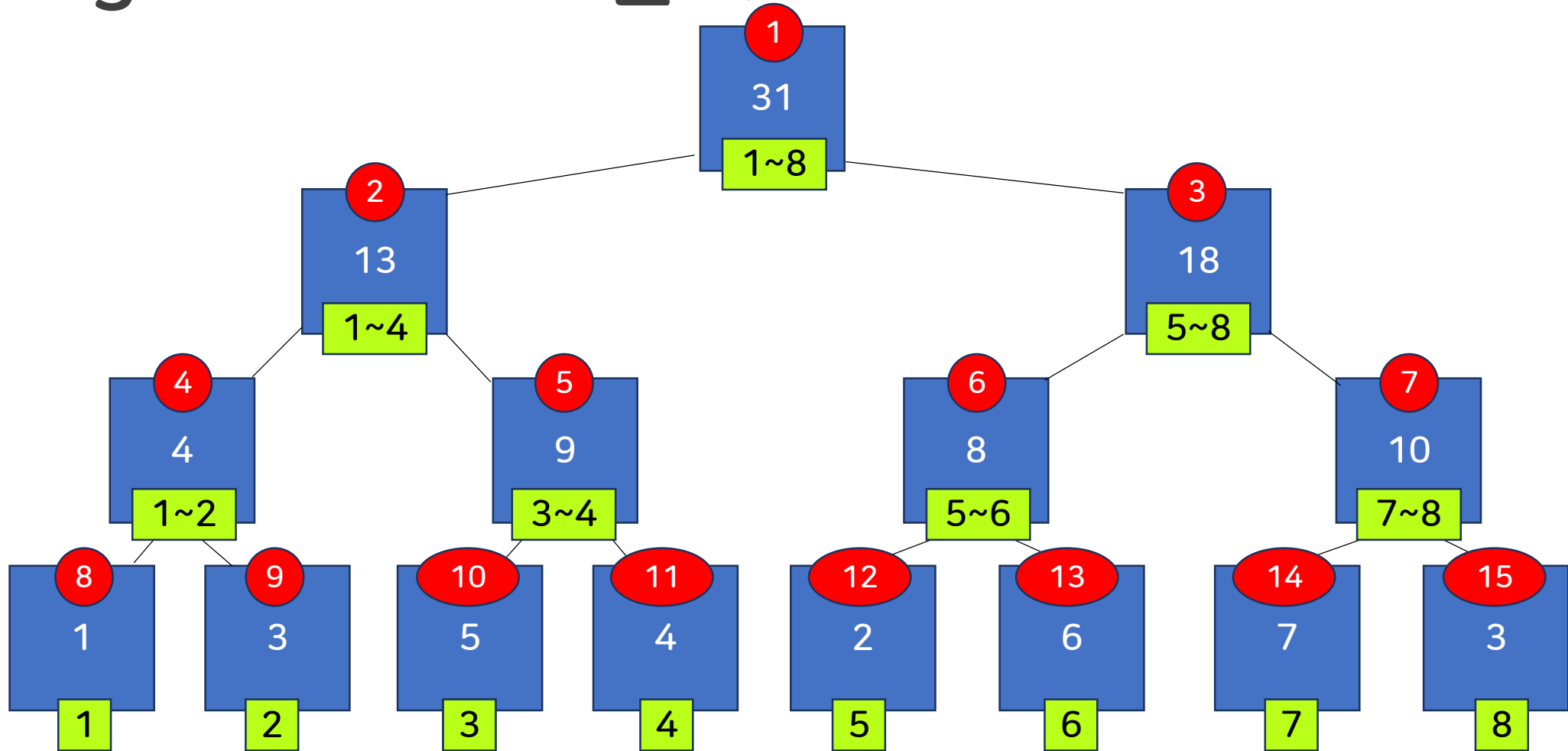
- 도입
- 구현

# Fenwick Tree

Segment Tree와 비슷한데, **배열의 크기가 Segment Tree의 절반**이라 메모리를 아낄 수 있다!

# Segment Tree 일때

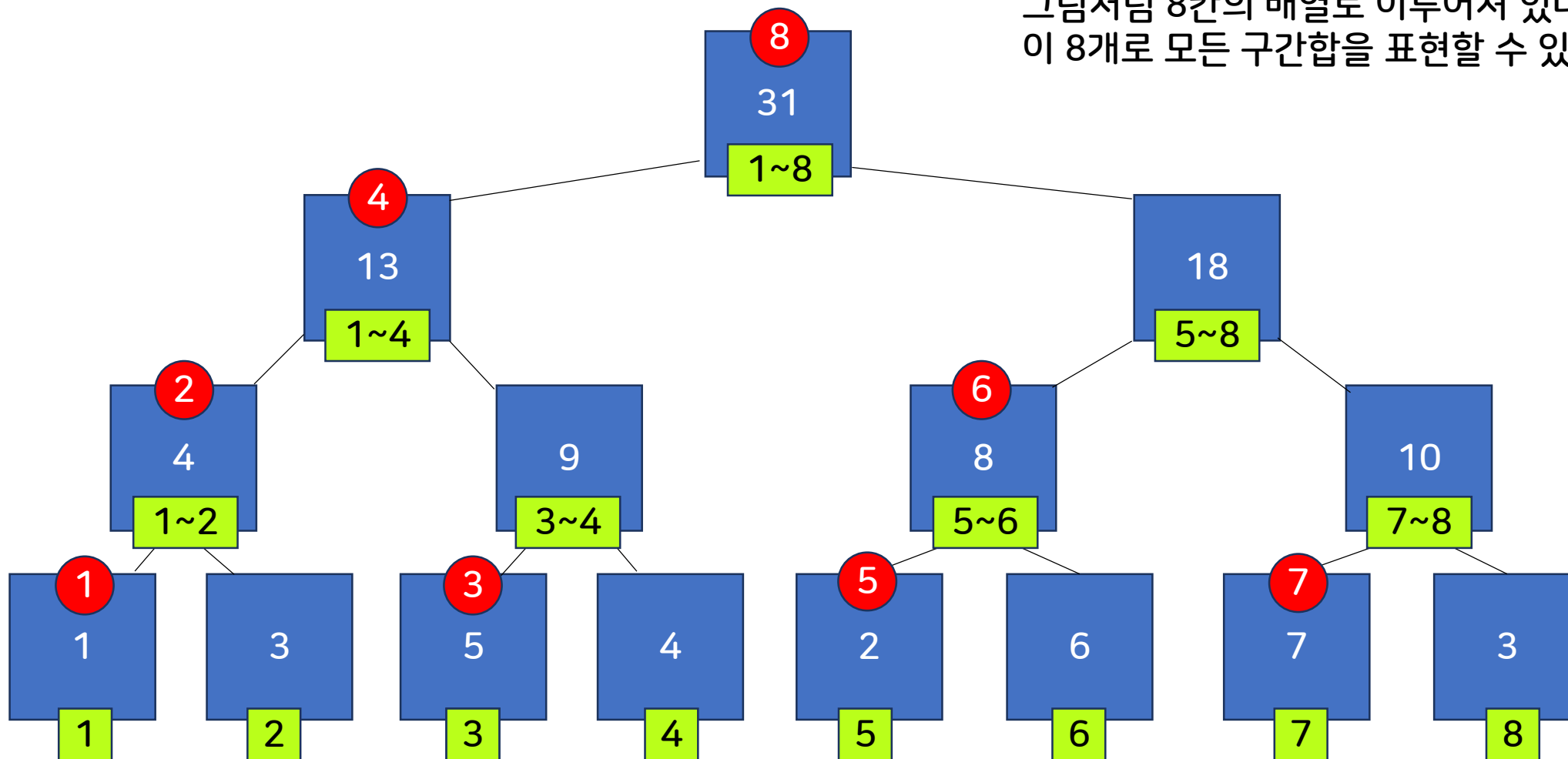
Fenwick Tree는?



# Fenwick Tree

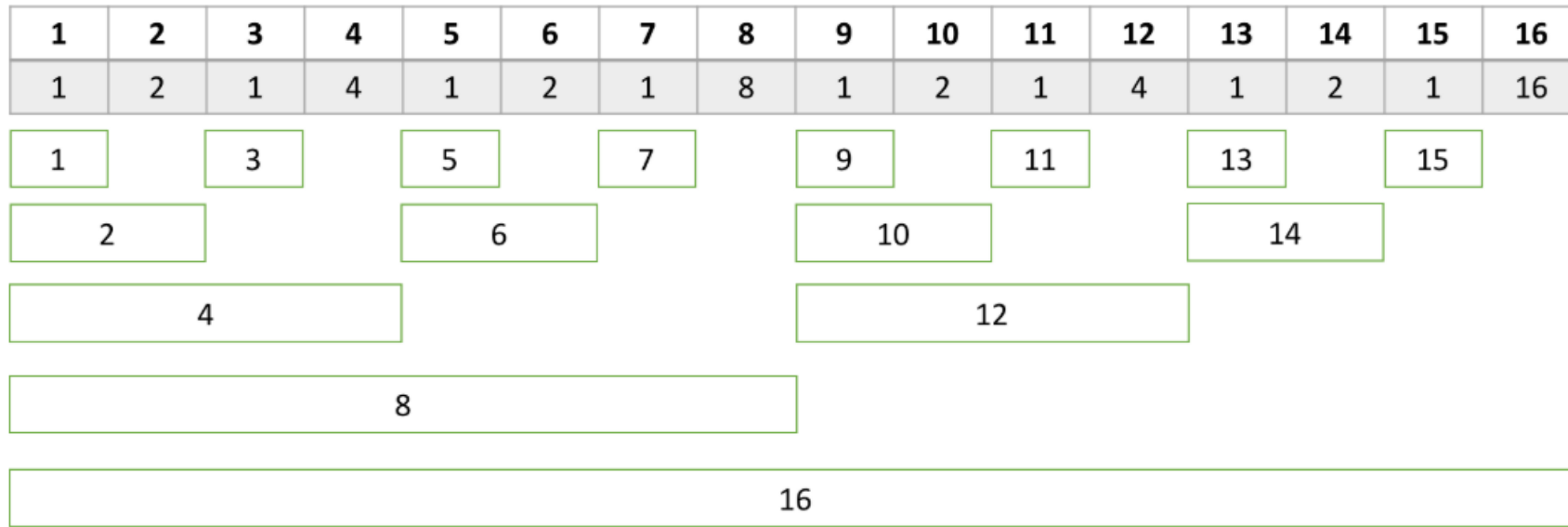
Fenwick Tree는???

그림처럼 8칸의 배열로 이루어져 있다!  
이 8개로 모든 구간합을 표현할 수 있다!





# Fenwick Tree



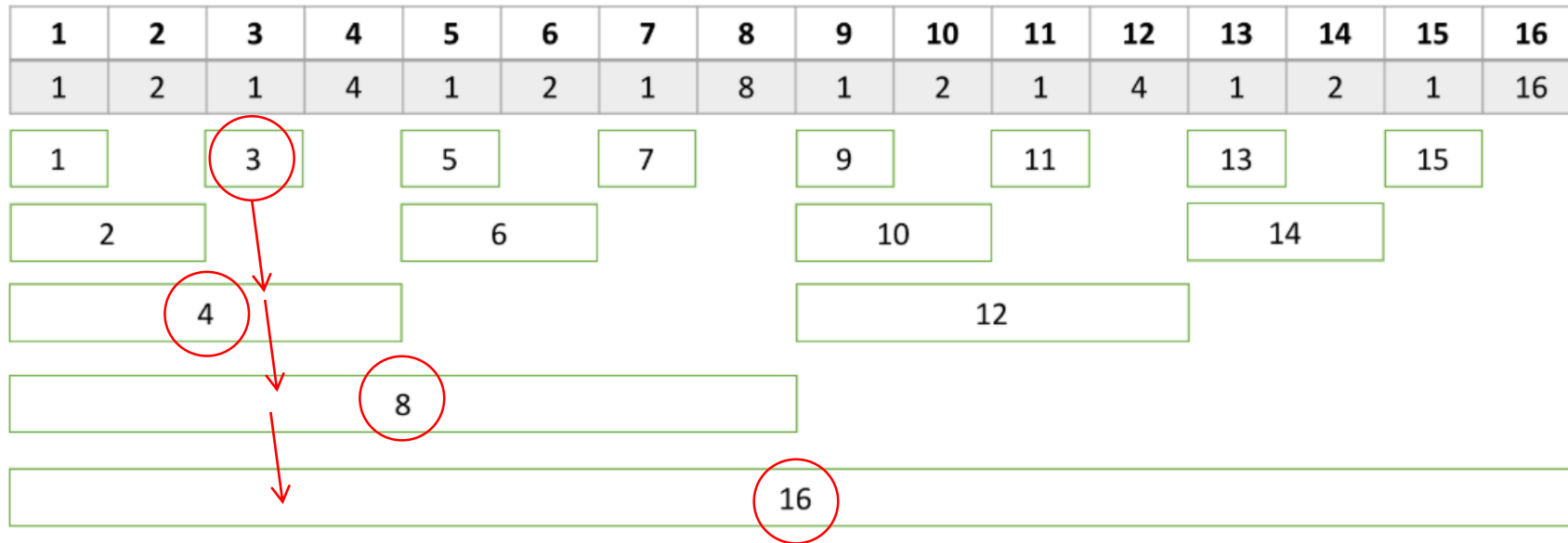
Fenwick Tree는 이와 같은 형태를 이루고 있다!

# Update

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16
1		3		5		7		9		11		13		15	
2				6				10				14			
4								12							
				8											
															16

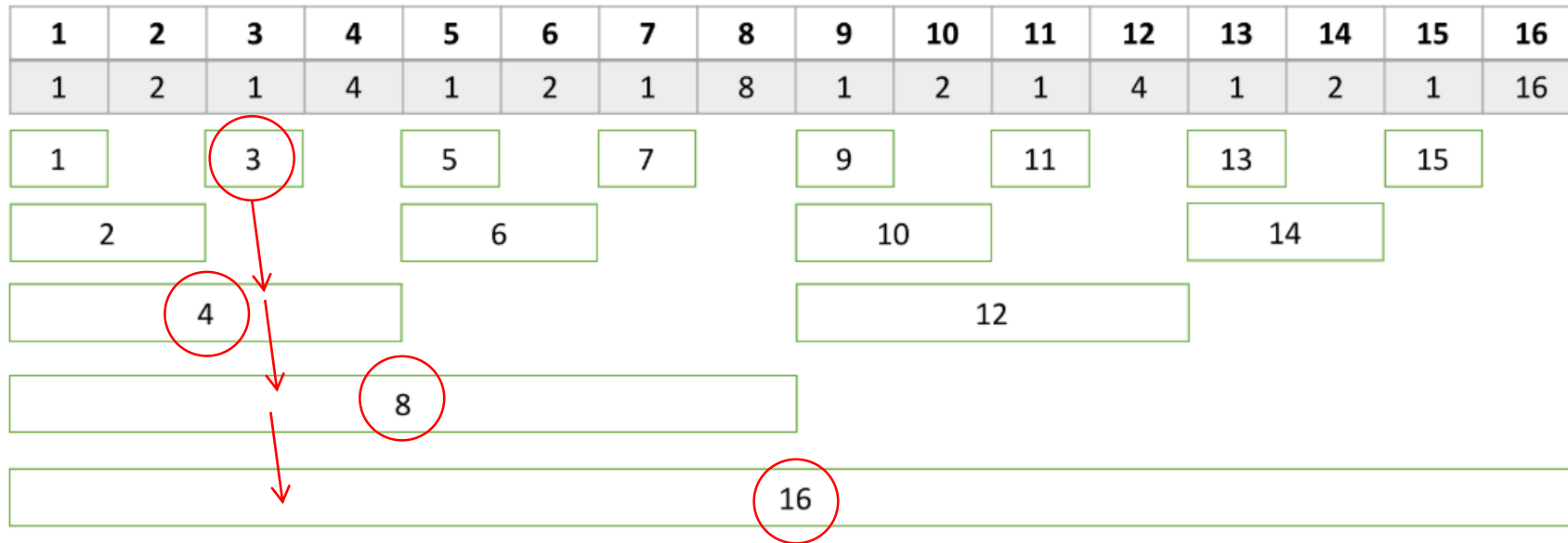
만약 3번 index를 update한다면, 몇 번 index들을 update해 주어야 할까?

# Update



3, 4, 8, 16번 index를 update시켜주어야 한다.

# Update



3, 4, 8, 16이 어떻게 나올까?

# Update

2진수로 표현해보자!

3 : 00011

+ 00001

4 : 00100

4 : 00100

+ 00100

8 : 01000

8 : 01000

+ 01000

16 : 10000

공통점은????

# Update

2진수로 표현해보자!

3 : 00011

+ 00001

4 : 00100

4 : 00100

+ 00100

8 : 01000

8 : 01000

+ 01000

16 : 10000

최하위 비트를 계속 더해준다!

# 최하위 비트 구하는 방법

```
int f(int x)
{
    ...
    return (x & -x);
}
```

3을 예로 들어보자.  $x = 0011$ ,  $-x = 1101$ (2의 보수)

$x \& -x$  를 하면,

0011

$\&1101 = 0001$  최하위 비트가 나온다!

# 구현

```
class fwtree {  
public:  
    vector<ll> tree;  
    vector<ll> numbers;  
    int size;
```

segment tree처럼 class를 만들자.

vector<long long> tree, numbers와 int size를 변수로 선언해준다.

tree는 구간합들을 저장하고 있는 Fenwick tree 배열이고,  
numbers는 원래의 값들을 저장하고 있는 배열이다.



# 구현

```
fwtree(int n)
{
    for (size = 1; size < n; size *= 2);
    tree.resize(size + 1);
    numbers.resize(size + 1);
}
```

생성자를 만들어준다. segment tree와 같은 방법을 사용하지만,  
tree와 numbers vector의 크기를  $2 * \text{size}$ 가 아닌 **size + 1로 해줘도 충분**하다.

# 구현

```
void update(int pos, ll x)
{
    //pos가 update하고자 하는 index, x가 바꾸고자 하는 값을 나타낸다
    //원래 저장되어 있는 값과 바꾸고자하는 값과의 차이를 u라는 변수에 담는다.
    int u = x - numbers[pos];
    //numbers 배열에서 pos index 위치의 값을 x로 바꾸어준다.
    numbers[pos] = x;
    //pos가 size 이하일 동안 최하위비트를 더해주면서 update해준다
    while (pos <= size)
    {
        tree[pos] += u;
        pos += (pos & (-pos));
    }
}
```

pos는 update하고자 하는 index, x는 바꾸고자 하는 값을 나타낸다.

# 구현

```
void update(int pos, int x)
{
    //pos가 update하고자 하는 index, x가 바꾸고자 하는 값을 나타낸다
    //원래 저장되어 있는 값과 바꾸고자하는 값과의 차이를 u라는 변수에 담는다.
    int u = x - numbers[pos];
    //numbers 배열에서 pos index 위치의 값을 x로 바꾸어준다.
    numbers[pos] = x;
    //pos가 size 이하일 동안 최하위비트를 더해주면서 update해준다
    while (pos <= size)
    {
        tree[pos] += u;
        pos += (pos & (-pos));
    }
}
```

3번 index에 원래 있던 값이 3이고, 바꾸고자 하는 값이 5라면,  
pos는 3, x는 5, u는  $5 - 3 = 2$ 가 된다.  
그리고 **numbers[pos]** 값을 바꾸고자 하는 값인 **x**로 바꾸어준다.

# 구현

```
void update(int pos, ll x)
{
    //pos가 update하고자 하는 index, x가 바꾸고자 하는 값을 나타낸다
    //원래 저장되어 있는 값과 바꾸고자하는 값과의 차이를 u라는 변수에 담는다.
    int u = x - numbers[pos];
    //numbers 배열에서 pos index 위치의 값을 x로 바꾸어준다.
    numbers[pos] = x;
    //pos가 size 이하일 동안 최하위비트를 더해주면서 update해준다
    while (pos <= size)
    {
        tree[pos] += u;
        pos += (pos & (-pos));
    }
}
```

pos가 size 이하인 동안에는 pos에 최하위비트를 더해주면서 해당 tree의 값을 update해준다.  
예를 들어 size가 16이고, pos가 3이라면 3, 4, 8, 16번째 값을 바꾸게 된다.

# Sum

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16
1		3		5		7		9		11		13		15	
2				6				10				14			
4								12							
8															
16															

그렇다면 구간합은 어떤 식으로 구할까?

# Sum

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16
1		3		5		7		9		11		13		15	
2				6				10				14			
4								12							
8															
16															

Fenwick Tree는 segment tree처럼

a ~ b까지의 구간 합을 구하는 함수를 구현하는 것이 아니라

처음부터 b까지, 즉 **1~b까지의 구간 합을 구하는 함수를 구현**한다.

# Sum

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16
1		3		5		7		9		11		13		15	
2				6				10				14			
4								12							
8															
16															

그렇다면  $a \sim b$ 까지의 구간합을 구하려면

$1 \sim b$ 까지의 구간합에서  $1 \sim (a-1)$ 까지의 구간합을 빼주면 된다.

즉,  $\text{Sum}(b) - \text{Sum}(a-1)$ 을 해주면 된다.

# Sum

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16
1		3		5		7		9		11		13		15	
2				6				10				14			
4								12							
8															
16															

만약 13까지의 구간합을 구하려면 어느 index를 더해야 하는가?



# Sum

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16
1		3		5		7		9		11		13		15	
2				6				10				14			
4								12							
			8												
															16

8, 12, 13번 index의 값들을 더해주면 된다!  
여기에는 어떤 규칙이 있을까?

# Sum

2진수로 표현해보자!

13 : 11001

- 00001

12 : 11000

12 : 11000

- 01000

8 : 10000

8 : 10000

- 10000

0 : 00000

공통점은?

# Sum

2진수로 표현해보자!

13 : 11001

- 00001

12 : 11000

12 : 11000

- 01000

8 : 10000

8 : 10000

- 10000

0 : 00000

0이 될 때까지 최하위 비트를 계속 빼준다!

# 구현

```
// sum(int pos)
{
    //구간합을 저장할 변수 ret를 0으로 초기화시켜준다.
    int ret = 0;
    //pos가 즉, index가 0이 되기 전까지 tree에 저장된 구간합의 값을 ret에 계속 더해준다
    while (pos > 0)
    {
        ret += tree[pos];
        pos -= (pos & (-pos));
    }
    //ret 값을 return해준다
    return ret;
}
```

pos를 인자로 받고, 1~pos까지의 구간합을 return해주는 함수이다.  
먼저 return해줄 구간합을 담을 변수 **ret**를 **0**으로 초기화해준다

# 구현

```
// sum(int pos)
{
    //구간합을 저장할 변수 ret를 0으로 초기화시켜준다.
    int ret = 0;
    //pos가 즉, index가 0이 되기 전까지 tree에 저장된 구간합의 값을 ret에 계속 더해준다
    while (pos > 0)
    {
        ret += tree[pos];
        pos -= (pos & (-pos));
    }
    //ret 값을 return해준다
    return ret;
}
```

그 후 pos가 0이 되기 전까지 최하위비트를 계속 빼 주면서 구간합을 ret에 계속 더해준다.

예를 들어 pos가 13이라면 13, 12, 8에 담겨 있는 값들을 ret에 더해준다.

그 후 ret의 값을 return해준다.

# 전체 코드

```
class fwtree {
public:
    vector<ll> tree;
    vector<ll> numbers;
    int size;
    fwtree(int n)
    {
        for (size = 1; size < n; size *= 2);
        tree.resize(size + 1);
        numbers.resize(size + 1);
    }
};
```

# 전체 코드

```
void update(int pos, int x)
{
    //pos가 update하고자 하는 index, x가 바꾸고자 하는 값을 나타낸다
    //원래 저장되어 있는 값과 바꾸고자하는 값과의 차이를 u라는 변수에 담는다.
    int u = x - numbers[pos];
    //numbers 배열에서 pos index 위치의 값을 x로 바꾸어준다.
    numbers[pos] = x;
    //pos가 size 이하일 동안 최하위비트를 더해주면서 update해준다
    while (pos <= size)
    {
        tree[pos] += u;
        pos += (pos & (-pos));
    }
}
```

# 전체 코드

```

}
|| sum(int pos)
{
    //구간합을 저장할 변수 ret를 0으로 초기화시켜준다.
    || ret = 0;
    //pos가 즉, index가 0이 되기 전까지 tree에 저장된 구간합의 값을 ret에 계속 더해준다
    while (pos > 0)
    {
        ret += tree[pos];
        pos -= (pos & (-pos));
    }
    //ret 값을 return해준다
    return ret;
}
};
```



BOJ 2042 구간 곱 구하기

# 구간 곱 구하기

아까 풀었던 문제지만 Fenwick Tree로 다시 풀어보자!

BOJ 12837 가계부(Hard)

# 가계부(Hard)

아까 풀었던 구간 합 구하기랑 비슷하다!

BOJ 3653 영화수집

# 영화 수집

배열의 위치에 DVD가 있으면 1 없으면 0을 저장하고

Fenwick Tree를 이용하여 구간 합을 구한다.

Fenwick Tree의 size를  $N + M$ 으로 설정하고 DVD를 옮겨 주면 된다.

# BOJ 3006 터보소트

# 터보소트

배열의 위치에 숫자가 있으면 1 없으면 0을 저장하고  
Fenwick Tree를 이용하여 구간 합을 구한다.

Index연산을 여러 번 하기 때문에 머리가 아프다!!



감사합니다!