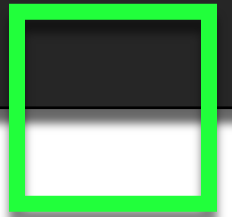
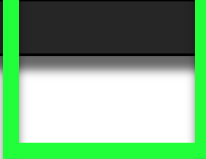


#고급반  
#1주차

# Segment & Fenwick Tree

T. 정태현





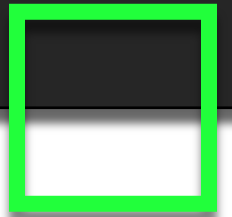
# Segment Tree & Fenwick Tree

구간에 대한 쿼리를 빠른 시간 안에 효율적으로 처리할 수 있는 자료구조

1. 배열의 특정 구간의 합 또는 최솟값, 최댓값 등
2. 배열의 값을 변경

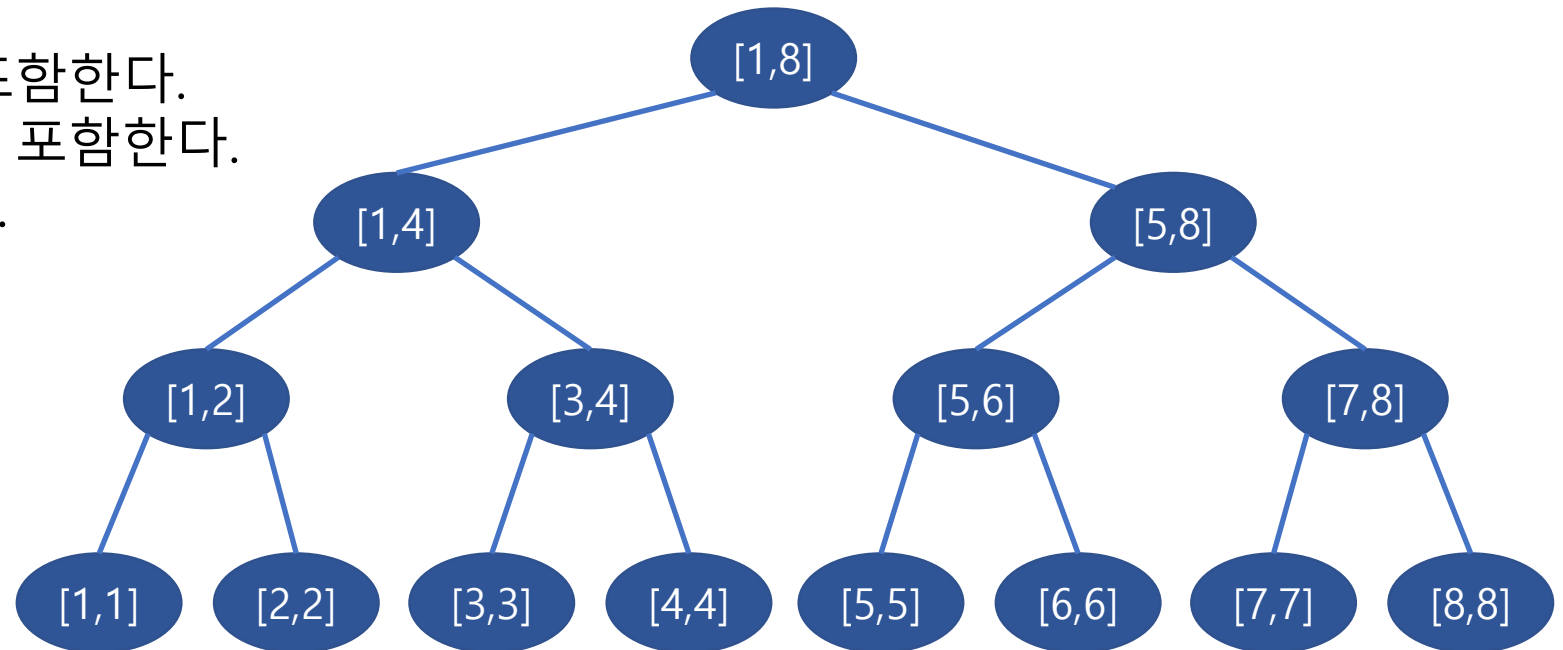
#1

# Segment Tree



# Segment Tree

1. 리프 노드는 자기 자신만을 포함한다.
2. 부모 노드는 자식들의 구간을 포함한다.
3. 루트는 전체 구간을 포함한다.
4. 모든 구간은 연속적이다.



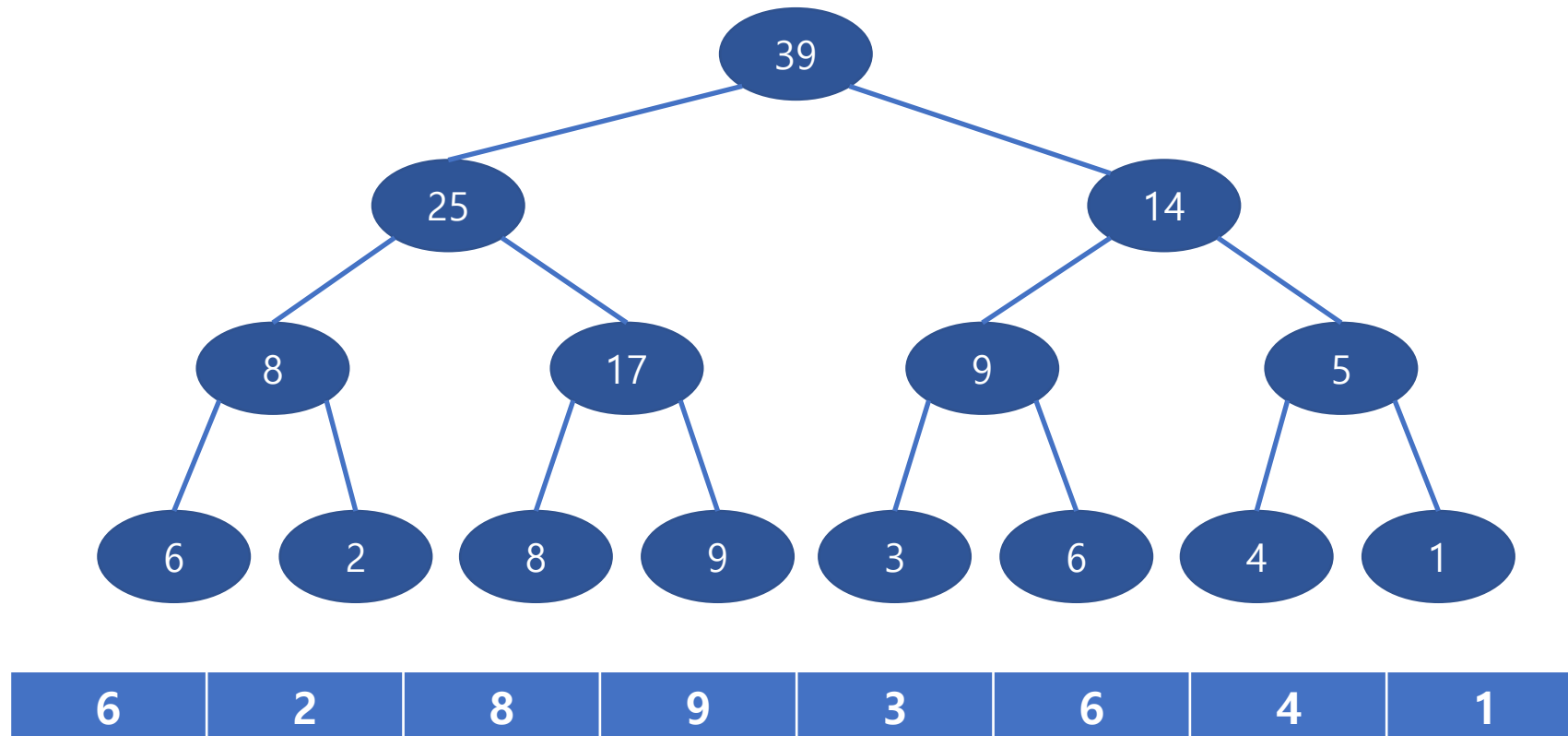
# Segment Tree

6	2	8	9	3	6	4	1
---	---	---	---	---	---	---	---

위와 같은 배열에서 구간합을 저장하는 Segment Tree를 만들어보자.

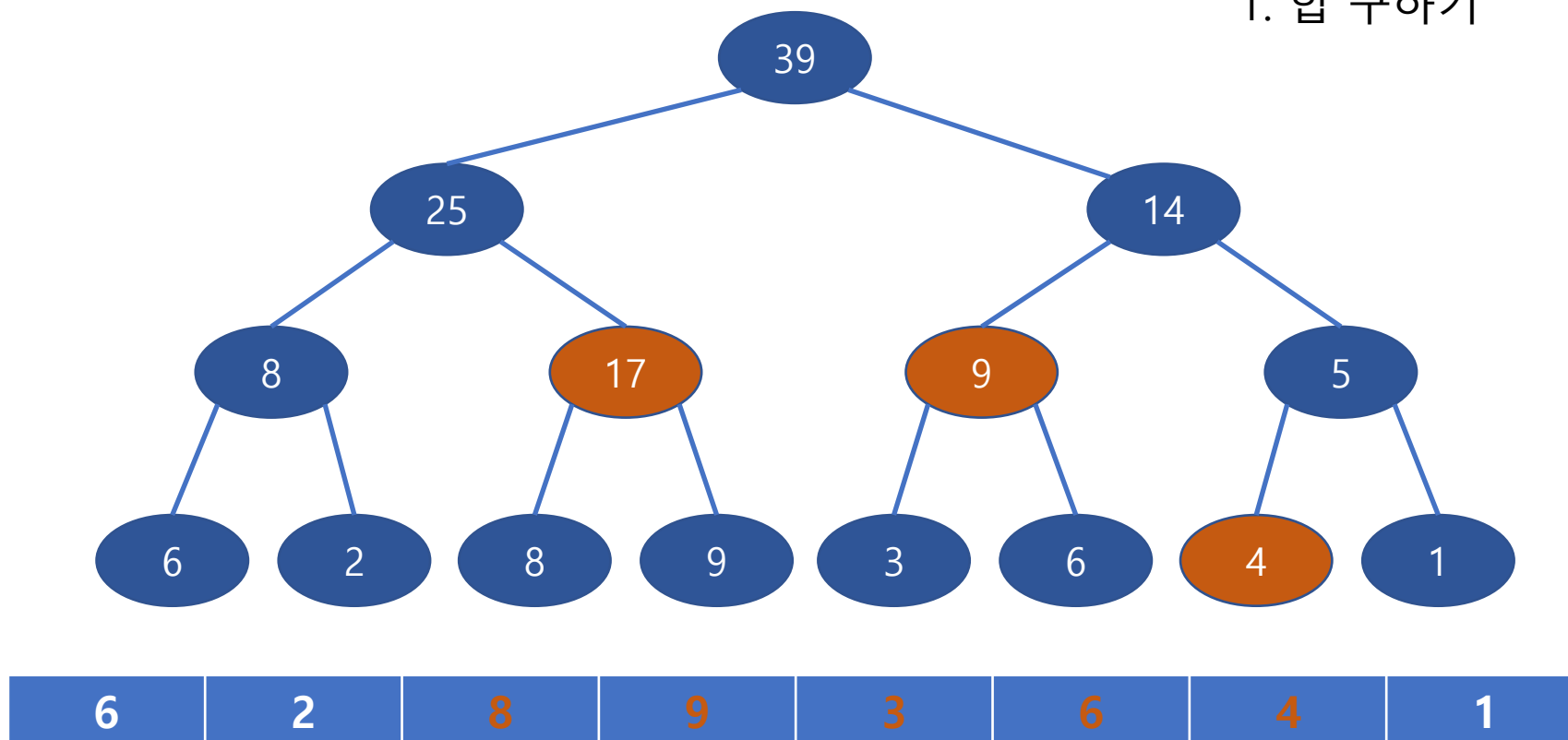


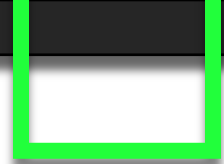
# Segment Tree



# Segment Tree

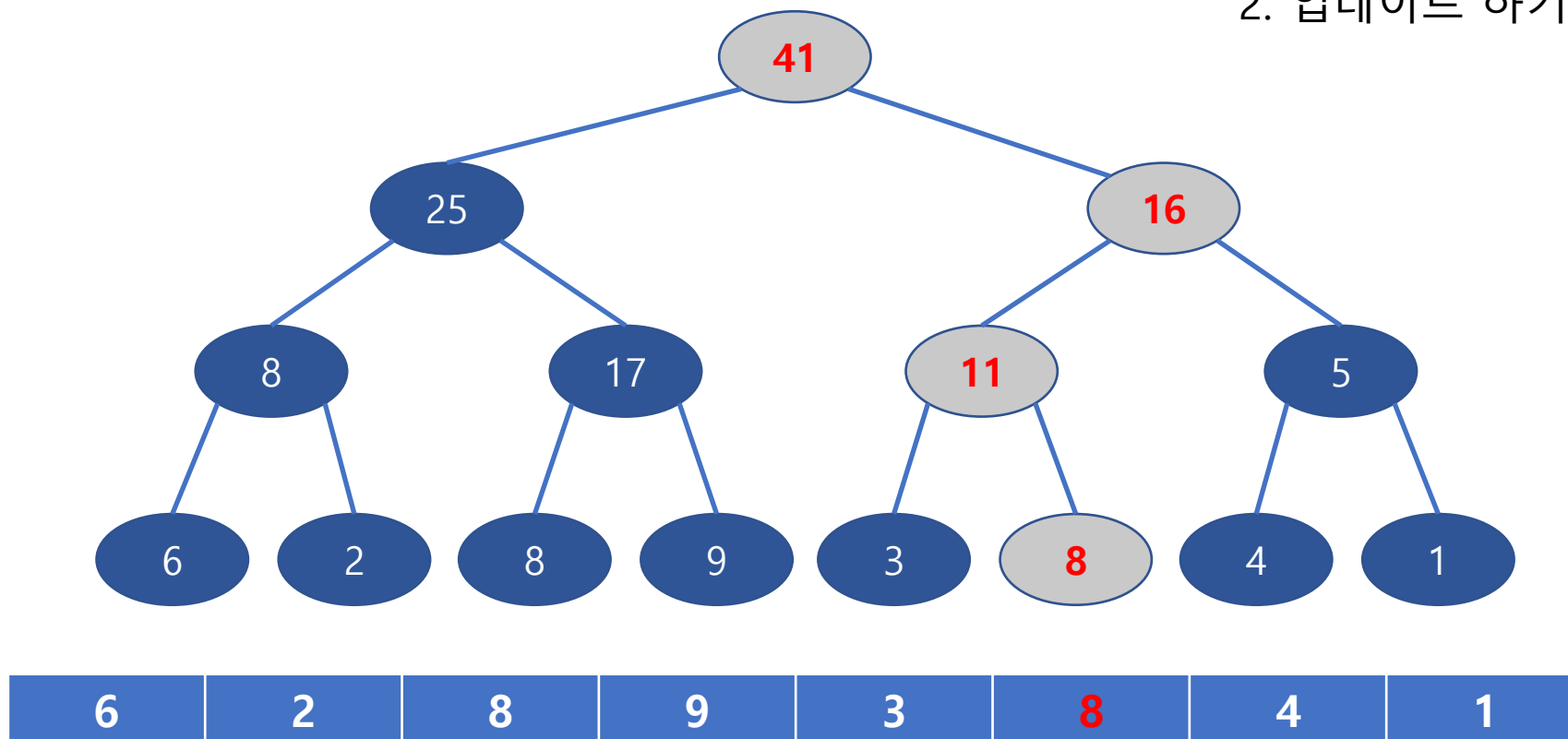
1. 합 구하기





# Segment Tree

2. 업데이트 하기







# Segment Tree

설마 트리를 직접 구현 해야하나...?



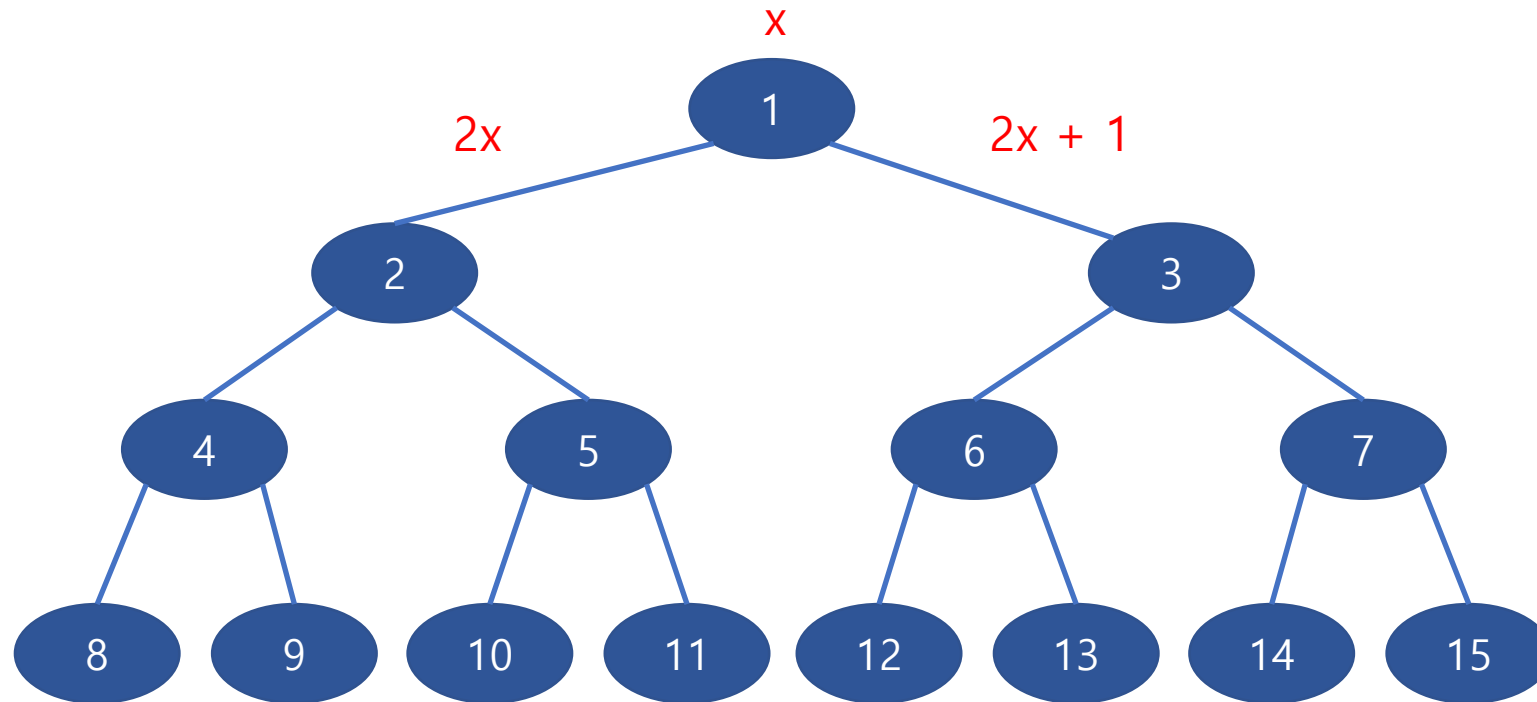
# Segment Tree

설마 트리를 직접 구현 해야하나...?

**Segment Tree를 완전 이진 트리라고 가정하면 쉽게 구현 가능**

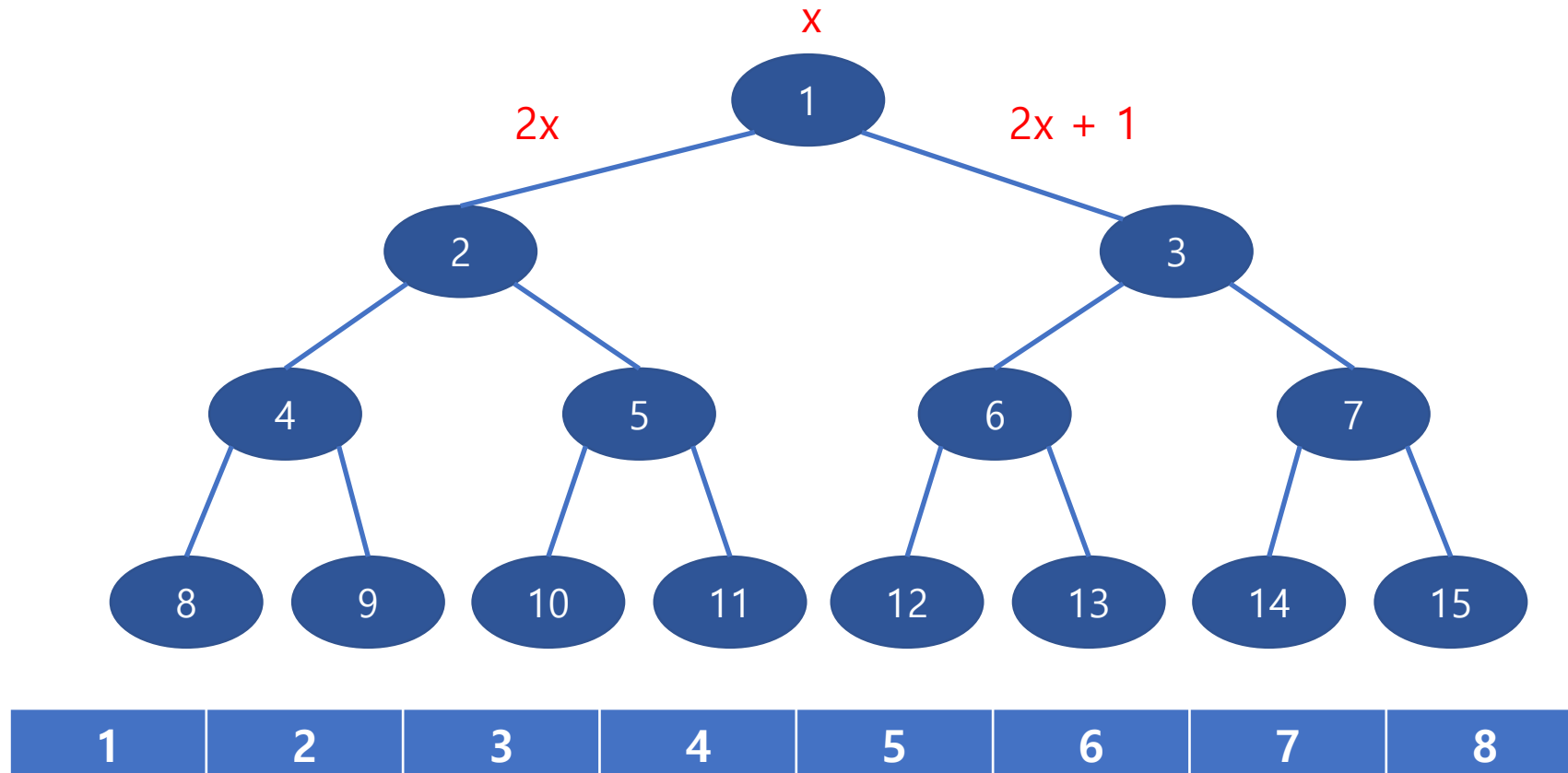


# Segment Tree



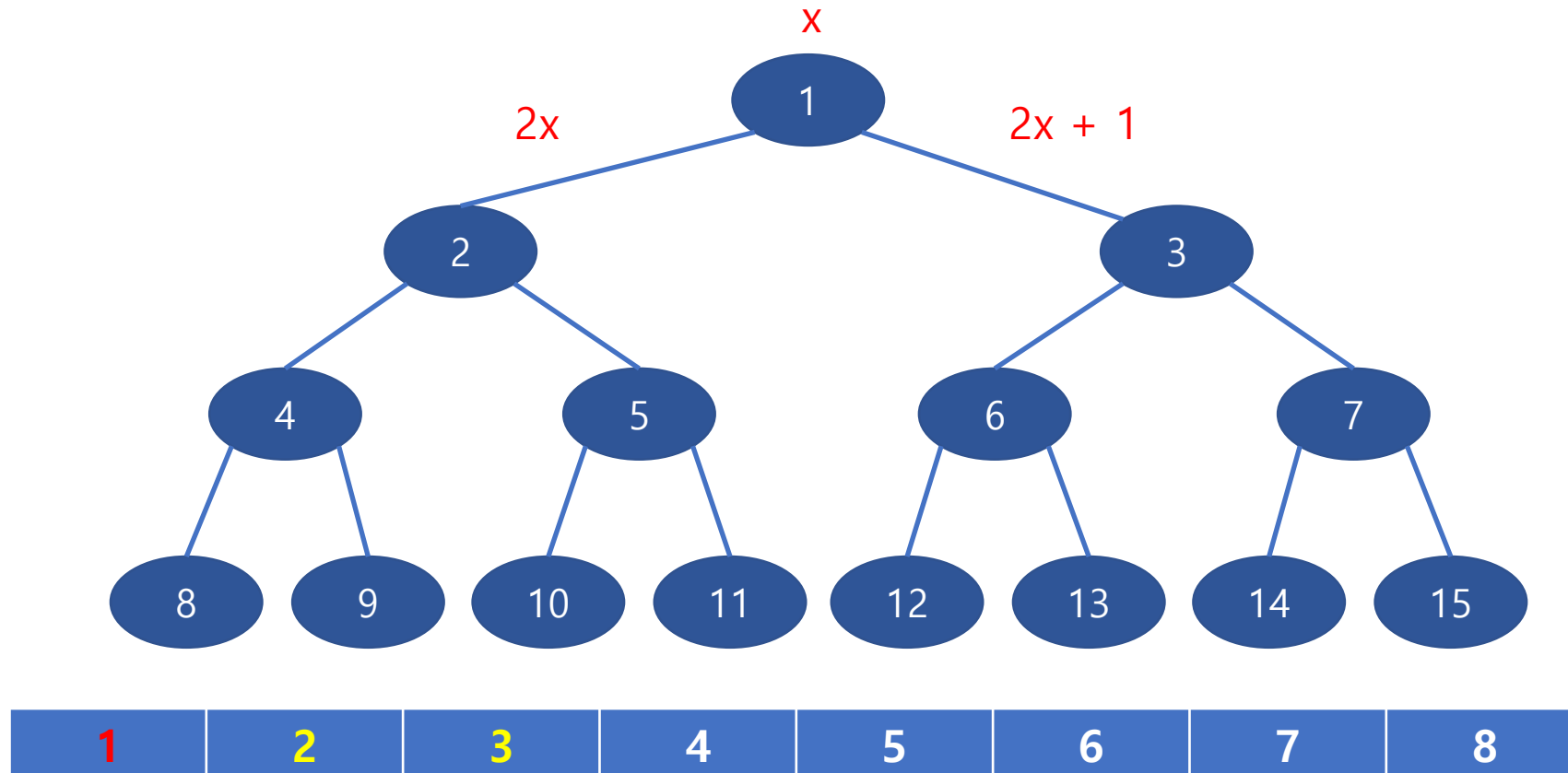


# Segment Tree



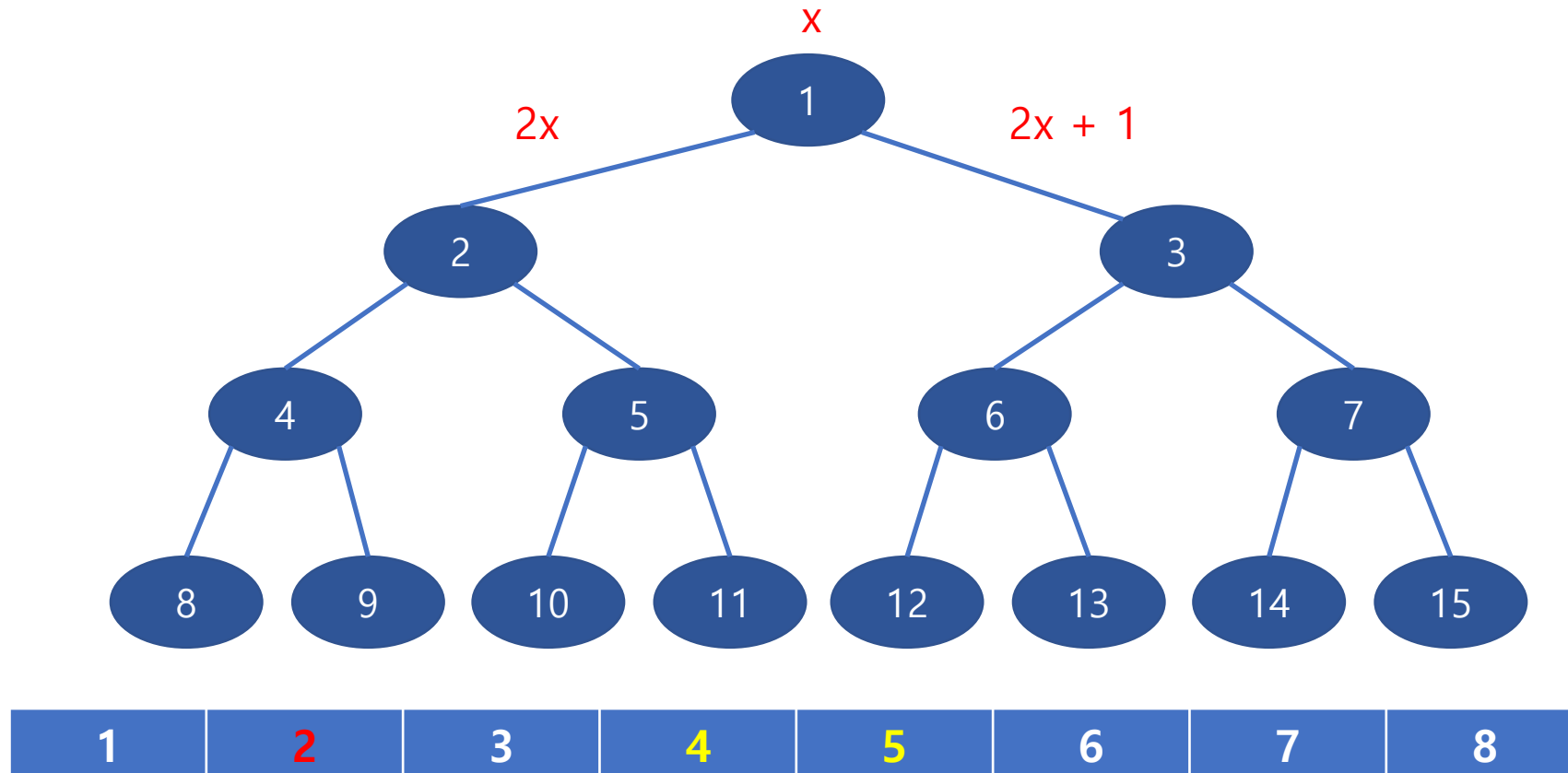


# Segment Tree



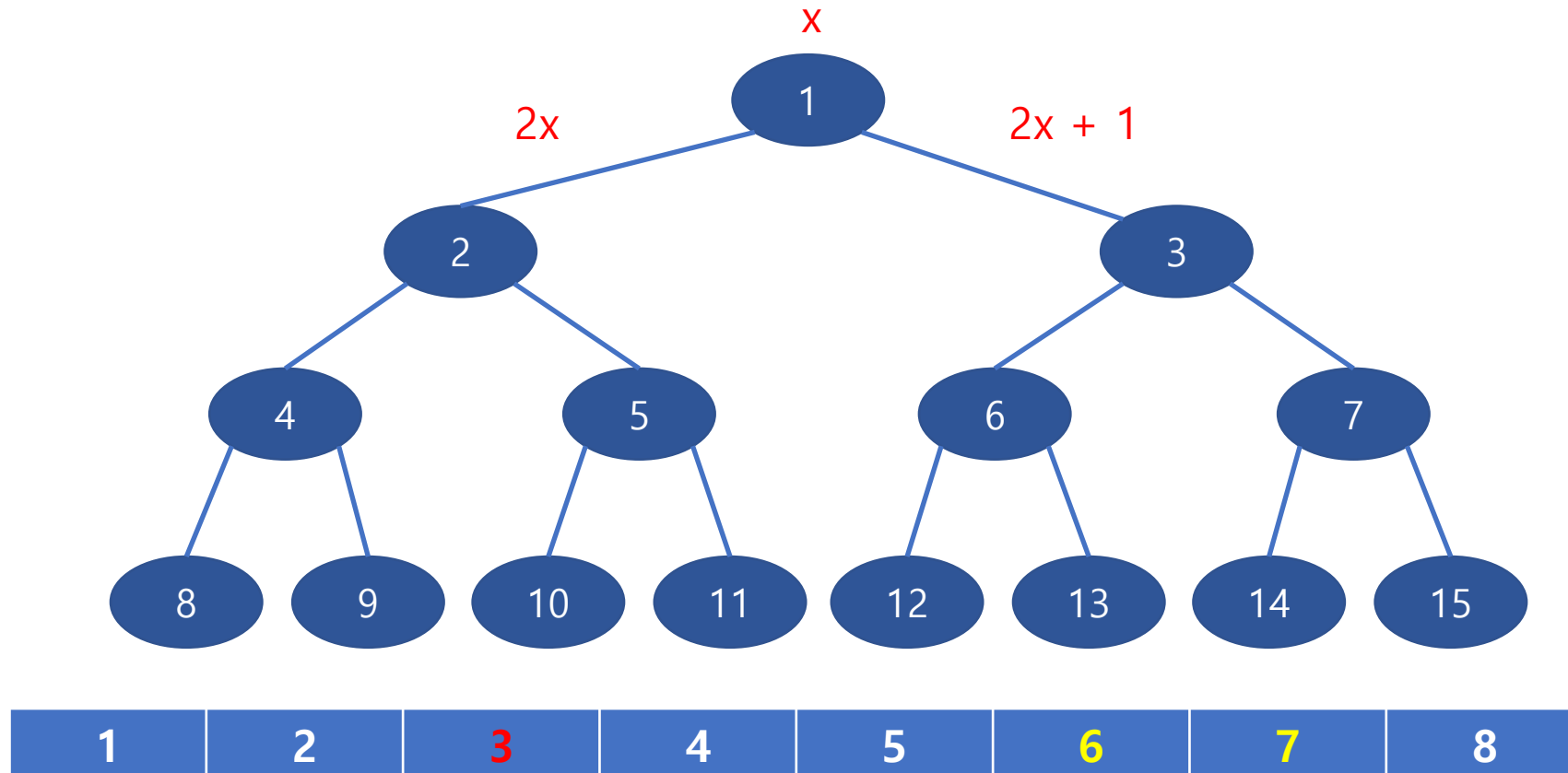


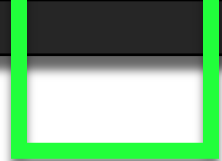
# Segment Tree





# Segment Tree

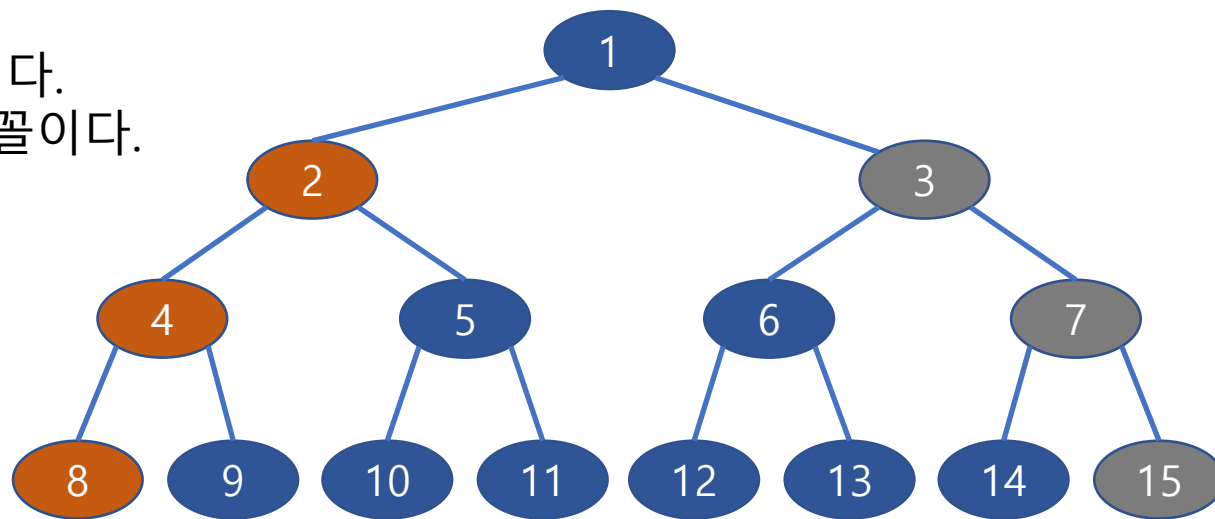




# Segment Tree

## 유용한 성질

1. 특정 Level의 가장 왼쪽 노드 index는  $2^n$  꼴이다.
2. 특정 Level의 가장 오른쪽 노드 index는  $2^{n+1}-1$  꼴이다.



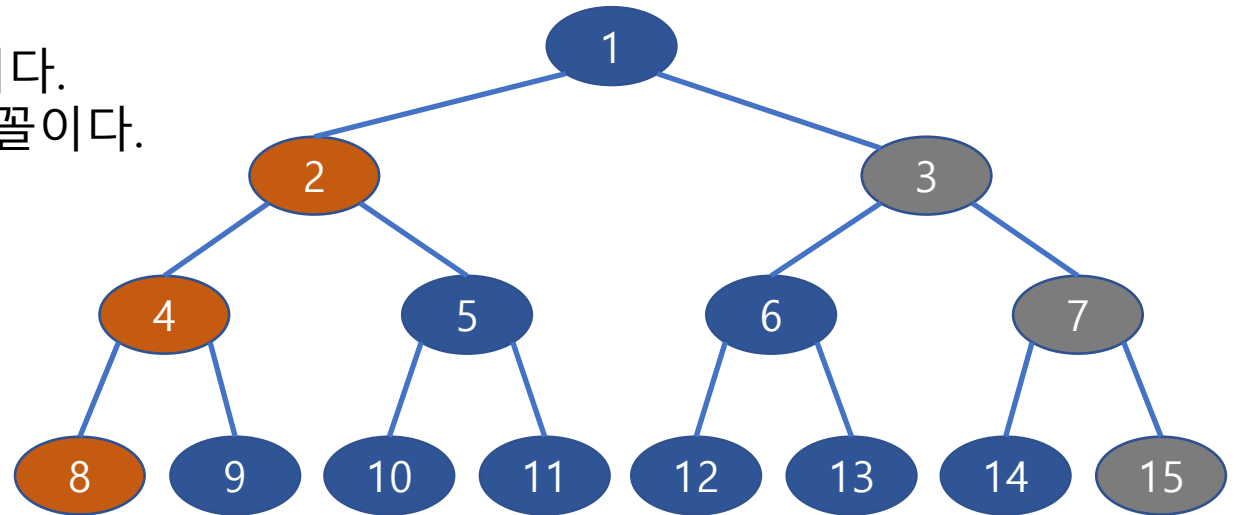


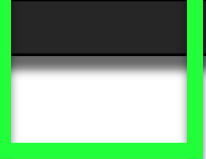
# Segment Tree

## 유용한 성질

1. 특정 Level의 가장 왼쪽 노드 index는  $2^n$  꼴이다.
2. 특정 Level의 가장 오른쪽 노드 index는  $2^{n+1}-1$  꼴이다.

위의 두가지 성질을 이용하여  
간단하게 구현이 가능하다!!





# Segment Tree 구현

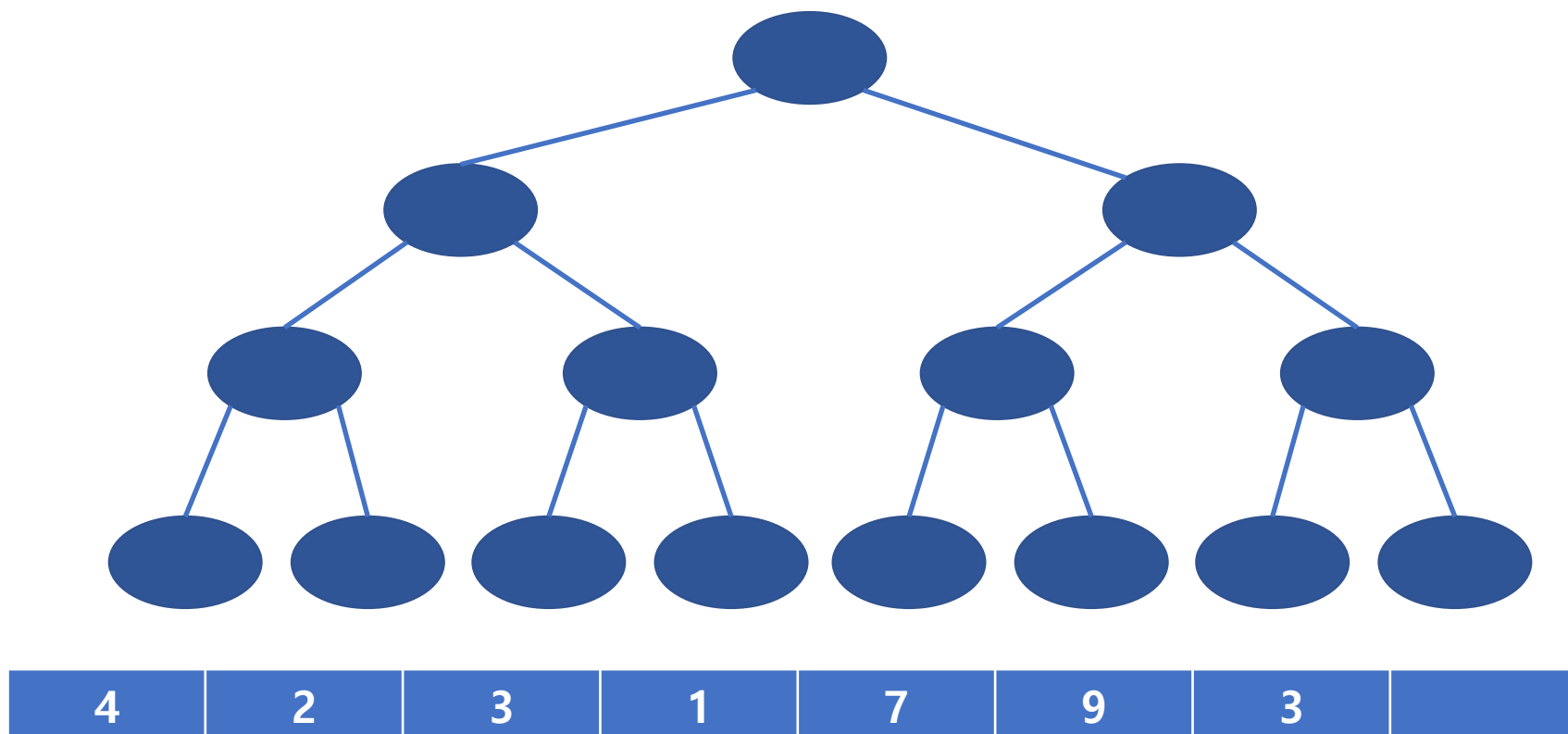
Segment Tree의 Size = 표현할 배열의 크기  $N$  이상의 2의 거듭제곱 수 중 최소값의 두배

# Segment Tree 구현

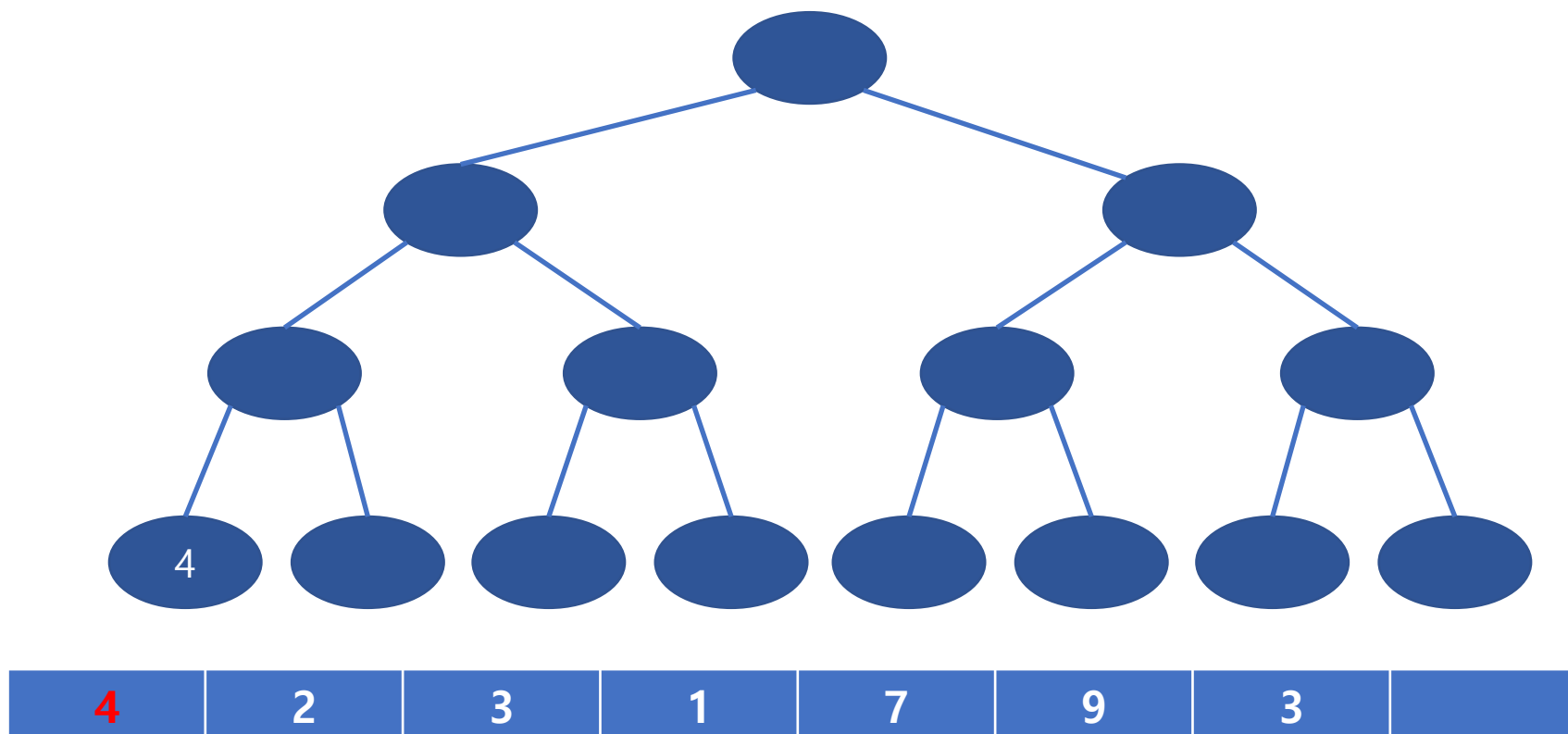
Segment Tree의 Size = 표현할 배열의 크기  $N$  이상의 2의 거듭제곱 수 중 최소값의 두배

Ex)  $\{n = 6, \text{Size} = 16\}$ ,  $\{n = 14, \text{Size} = 32\}$ ,  $\{n = 16, \text{Size} = 32\}$

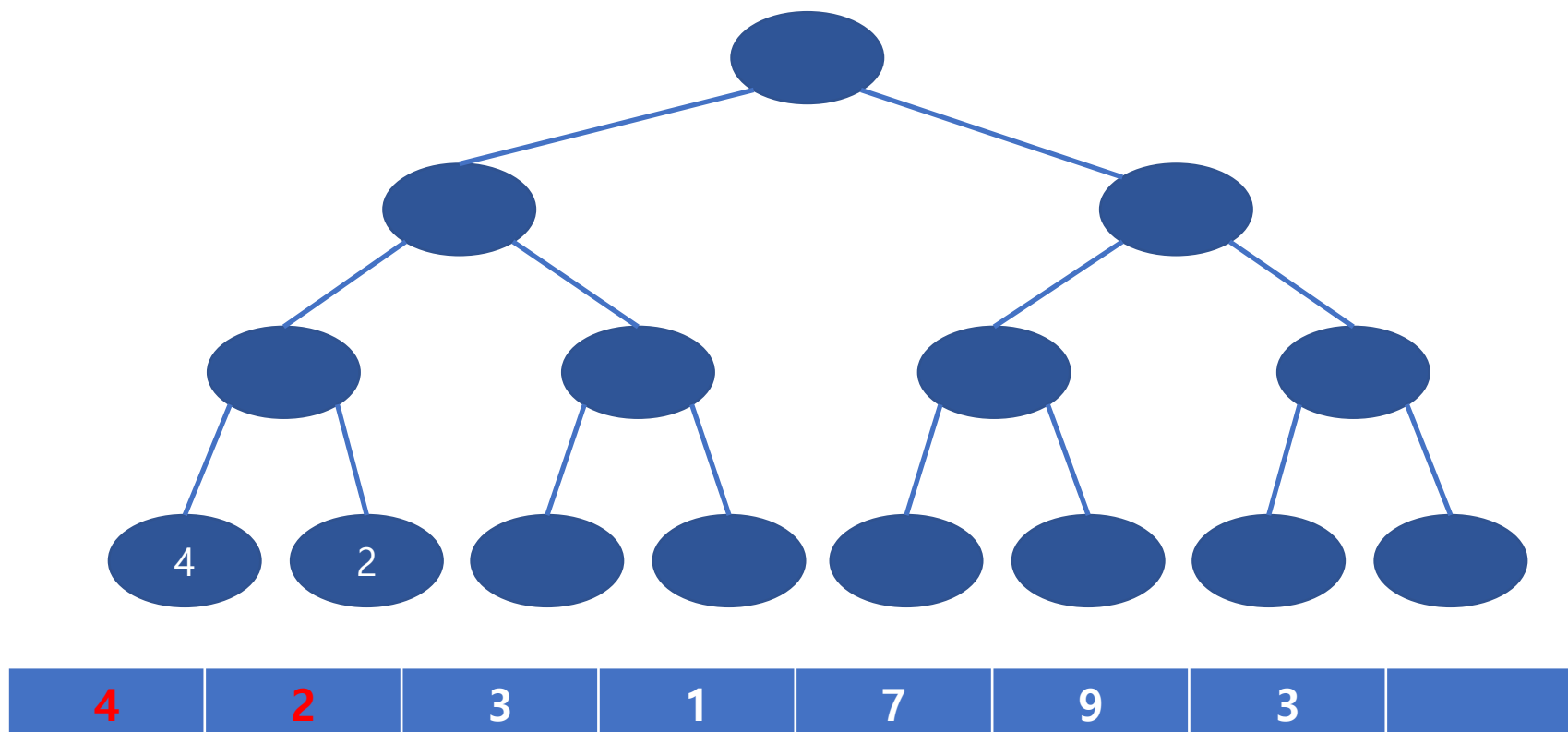
# Segment Tree 구현



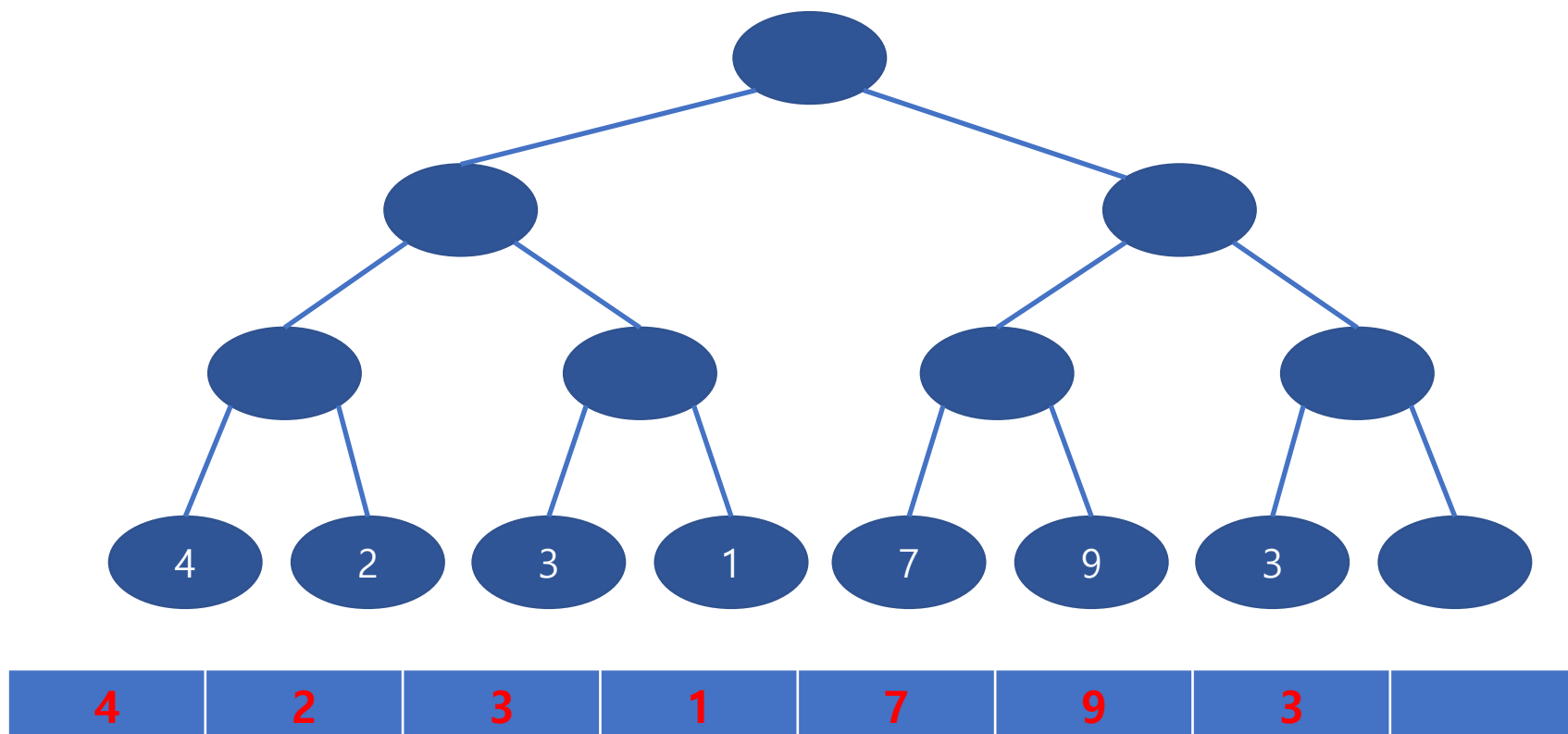
# Segment Tree 구현



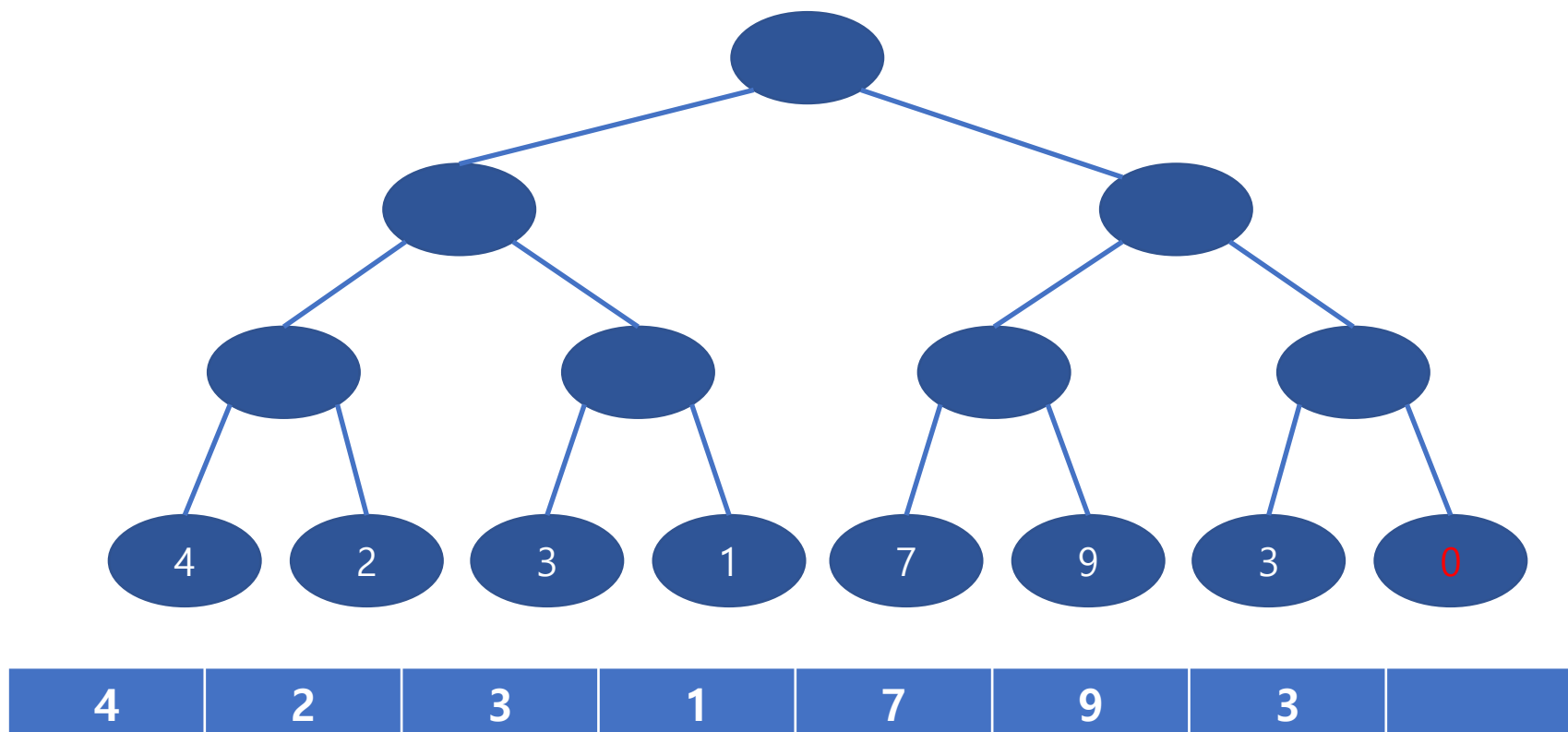
# Segment Tree 구현



# Segment Tree 구현



# Segment Tree 구현





# Segment Tree 구현

```
1 int main()
2 {
3     cin >> n;
4     for(int i = 1; i <= n; i++)
5         cin >> d[i];
6     Size = (1 << ((int)ceil(log2(n)) + 1));
7     SegTree.resize(Size);
8
9     for(int i = 1; i <= n; i++)
10         SegTree[i + Size / 2 - 1] = d[i];
11 }
12
13
```

# Segment Tree 구현

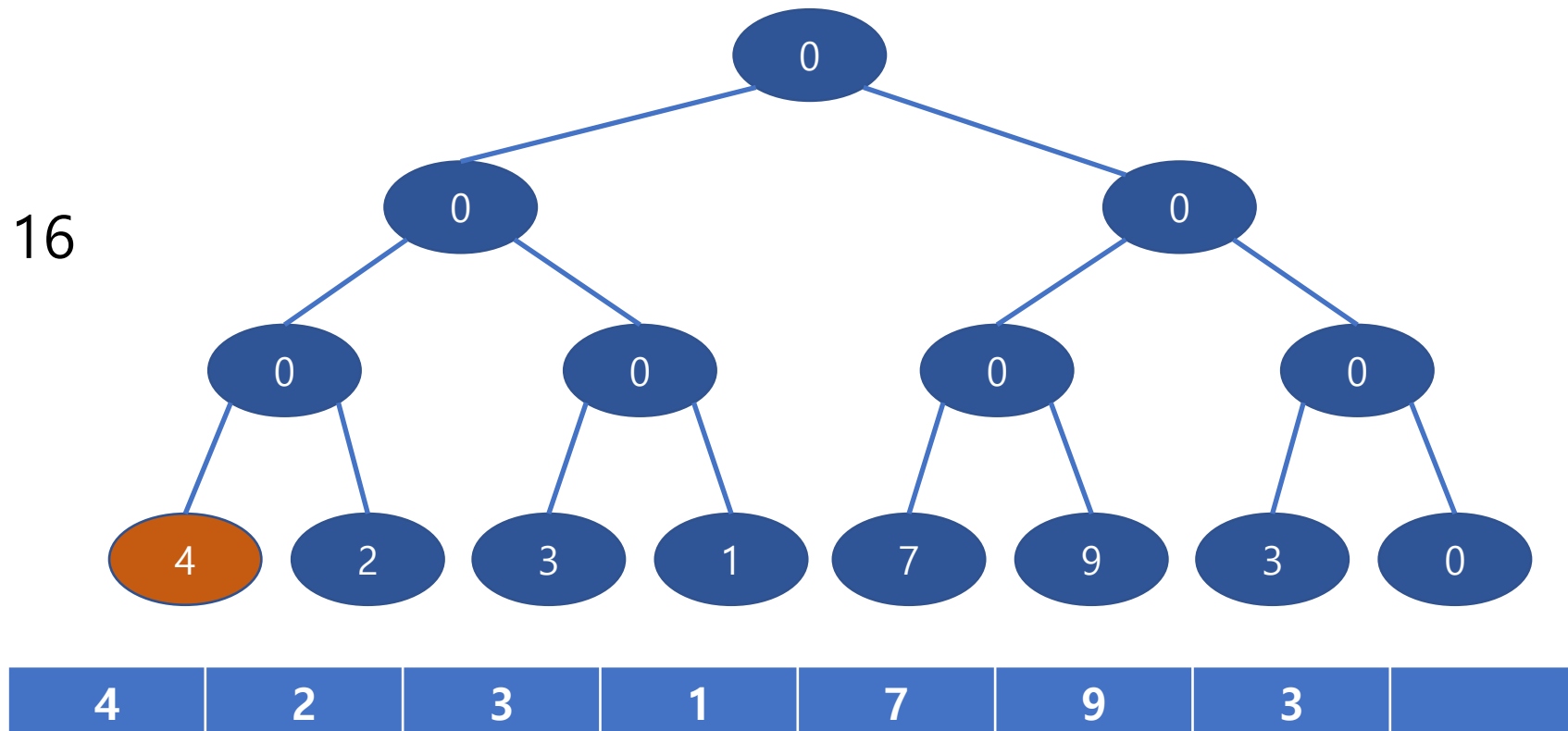
## 유용한 성질

1. 특정 Level의 가장 왼쪽 노드 index는  $2^n$  꼴이다.
2. 특정 Level의 가장 오른쪽 노드 index는  $2^{n+1}-1$  꼴이다.

```
1 int main()
2 {
3     cin >> n;
4     for(int i = 1; i <= n; i++)
5         cin >> d[i];
6     Size = (1 << ((int)ceil(log2(n)) + 1));
7     SegTree.resize(Size);
8
9     for(int i = 1; i <= n; i++)
10         SegTree[i + Size / 2 - 1] = d[i];
11 }
12
13
```

# Segment Tree 구현

$n = 7$   
Size = 16



# Segment Tree 구현

```
1
2   for(int i = 1; i <= n; i++)
3       SegTree[i + Size / 2 - 1] = d[i];
4
5   for(int i = Size / 2 - 1; i >= 1; i--)
6       SegTree[i] = SegTree[i*2] + SegTree[i*2 + 1];
7
8   }
```

# Segment Tree 구현

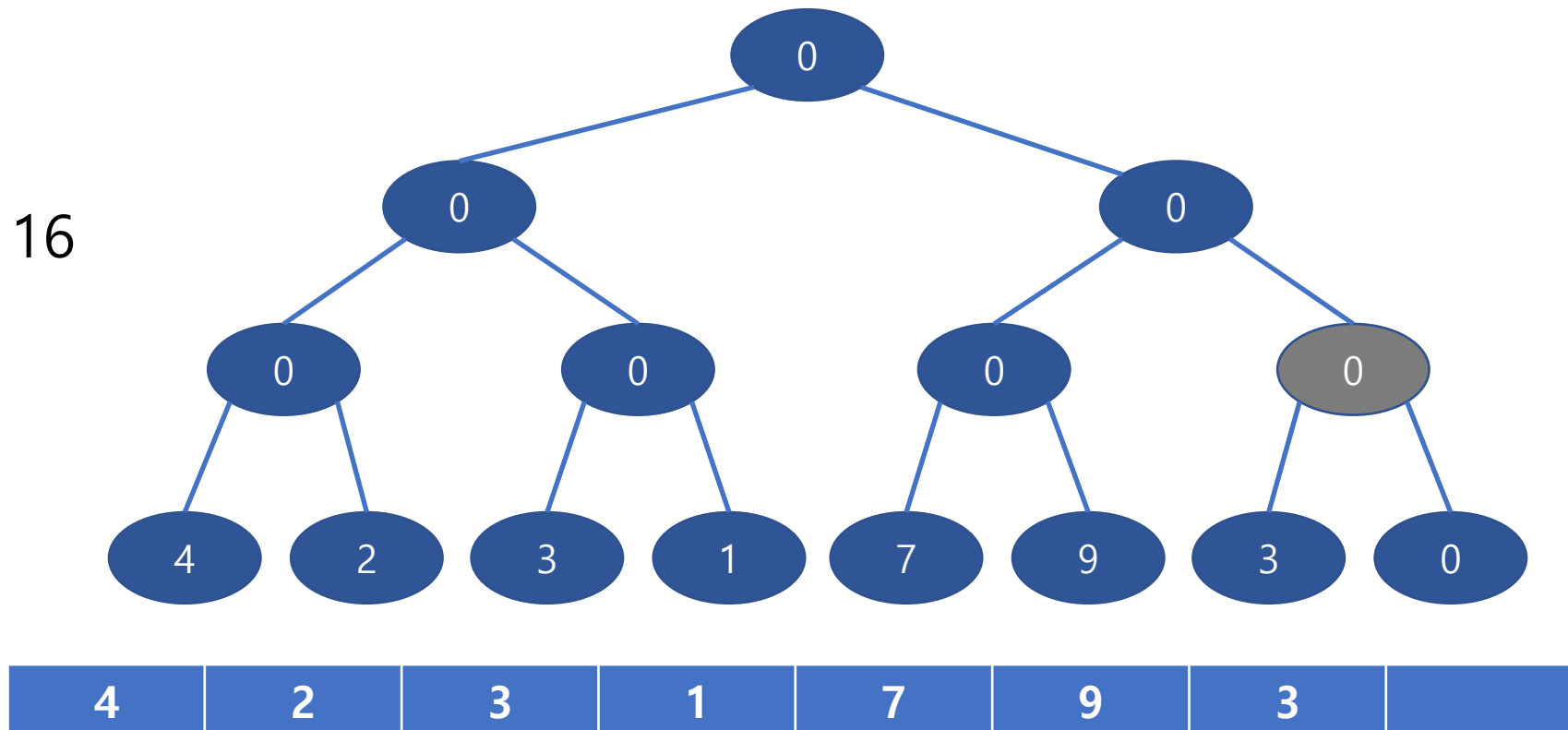
## 유용한 성질

1. 특정 Level의 가장 왼쪽 노드 index는  $2^n$  꼴이다.
2. 특정 Level의 가장 오른쪽 노드 index는  $2^n - 1$  꼴이다.

```
1
2   for(int i = 1; i <= n; i++)
3       SegTree[i + Size / 2 - 1] = d[i];
4
5   for(int i = Size / 2 - 1; i >= 1; i--)
6       SegTree[i] = SegTree[i*2] + SegTree[i*2 + 1];
7
8   }
```

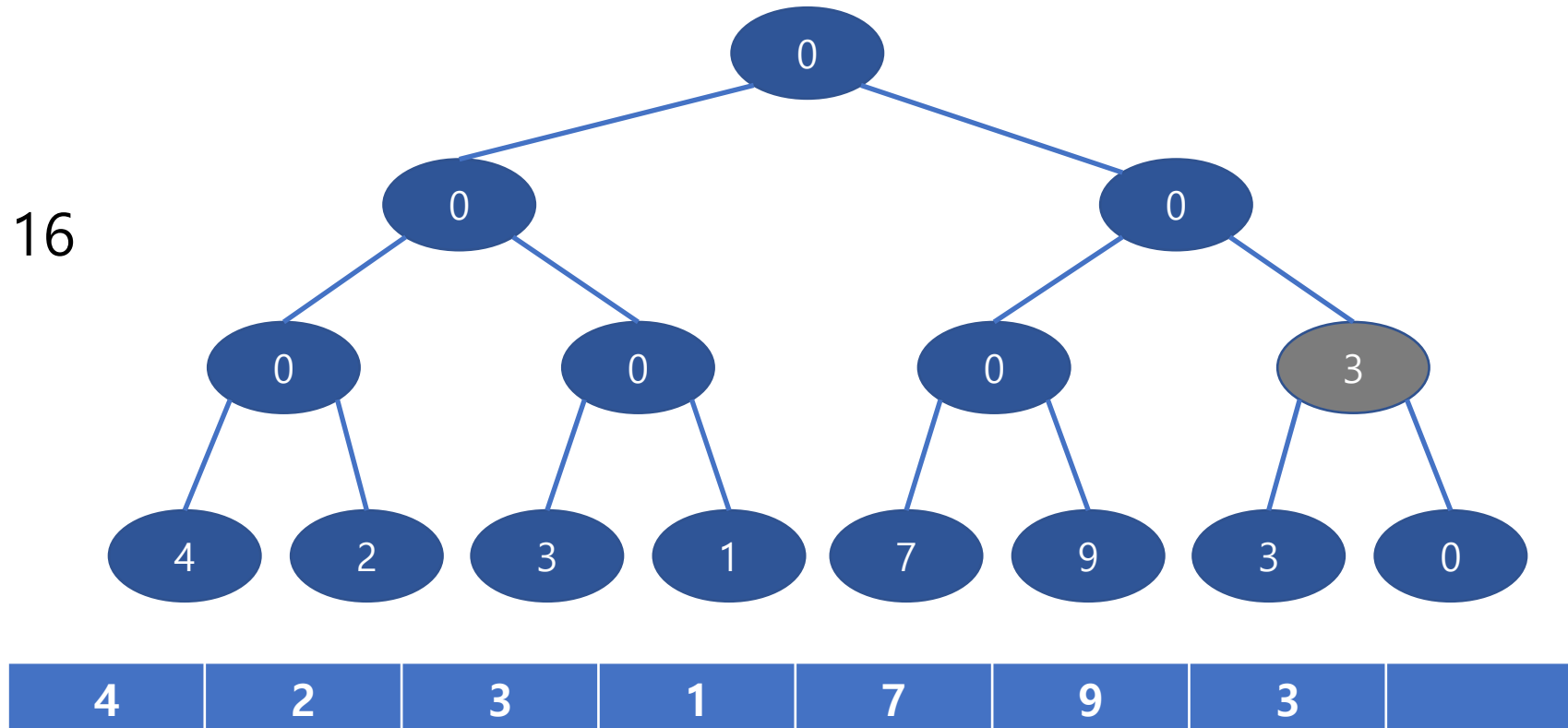
# Segment Tree 구현

$n = 7$   
Size = 16



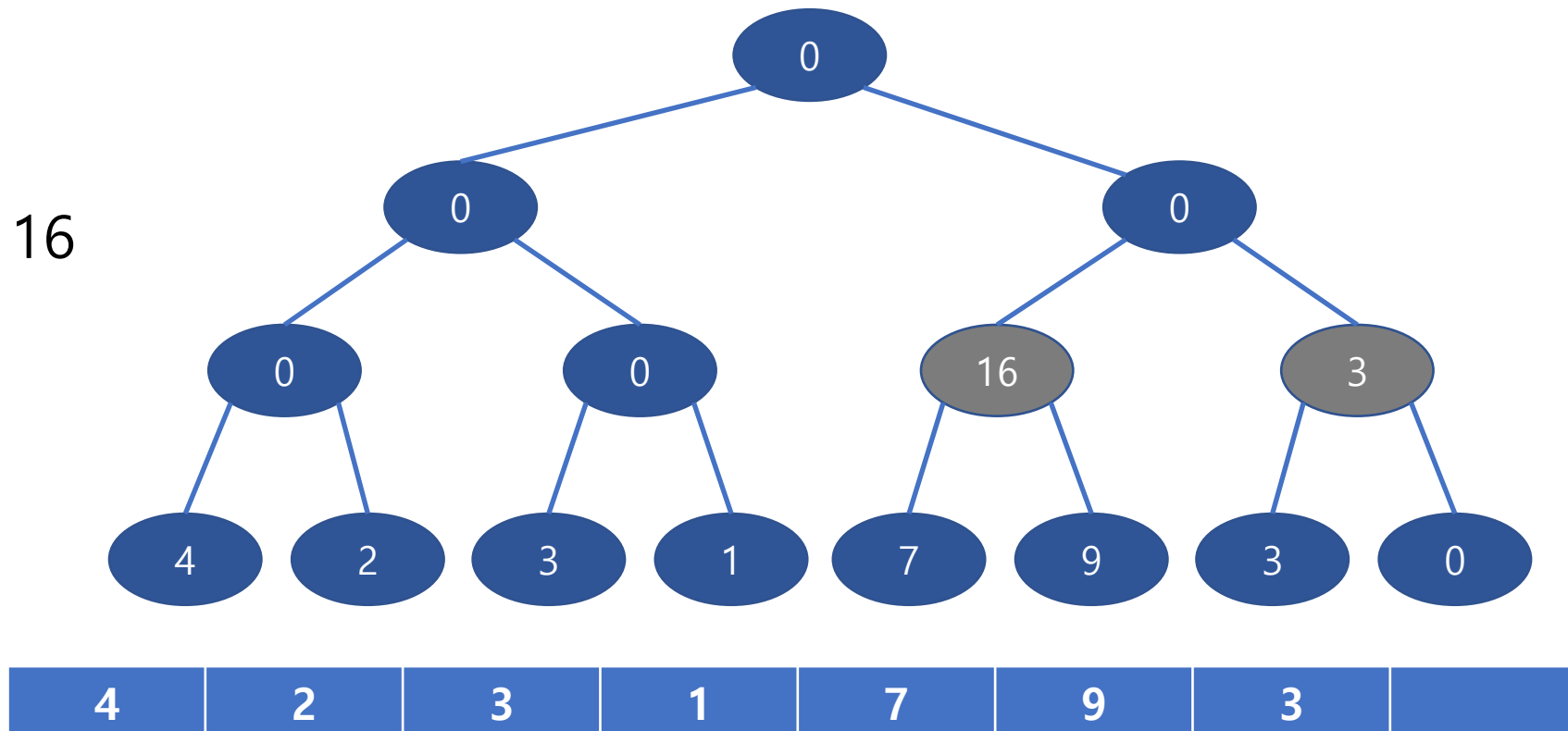
# Segment Tree 구현

$n = 7$   
Size = 16



# Segment Tree 구현

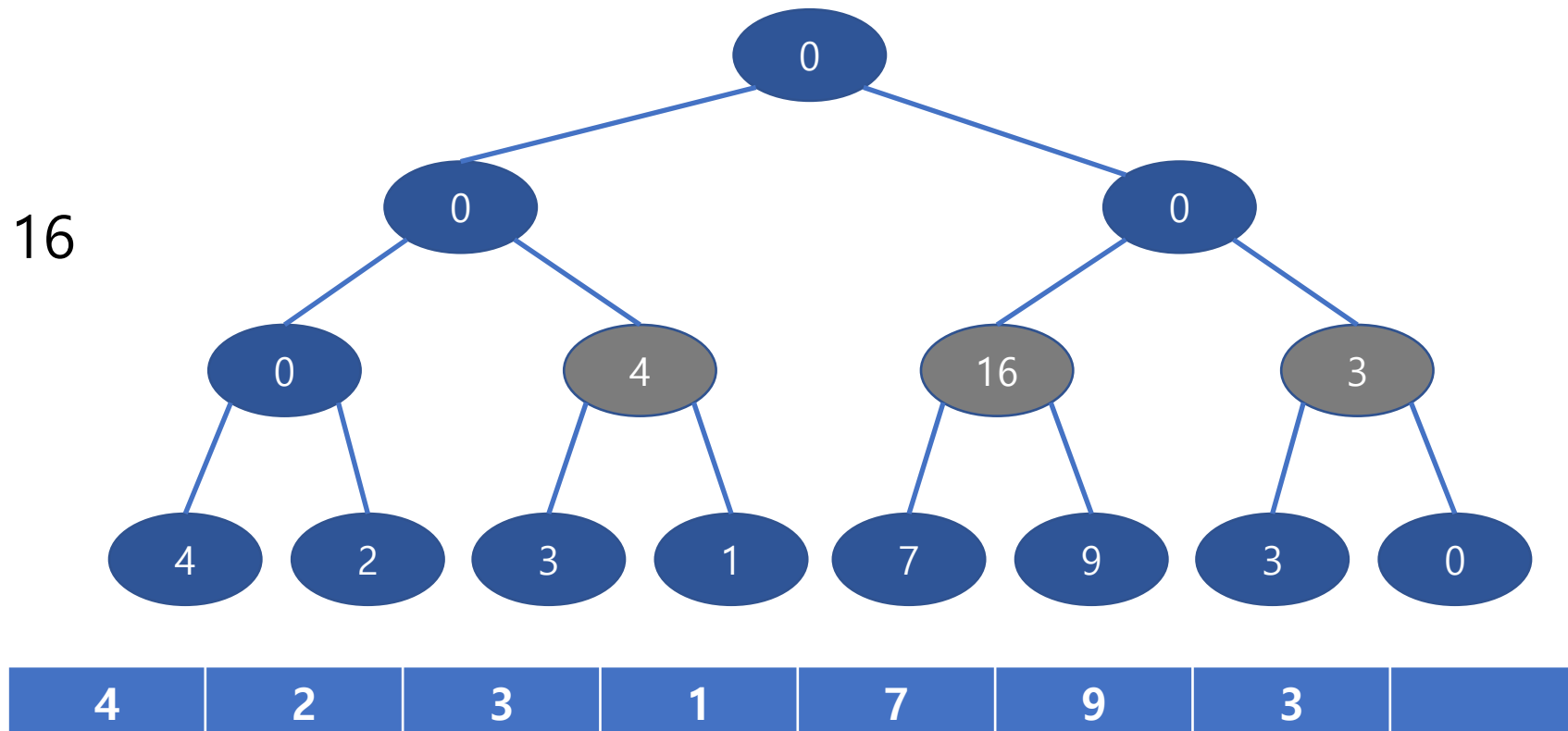
$n = 7$   
Size = 16





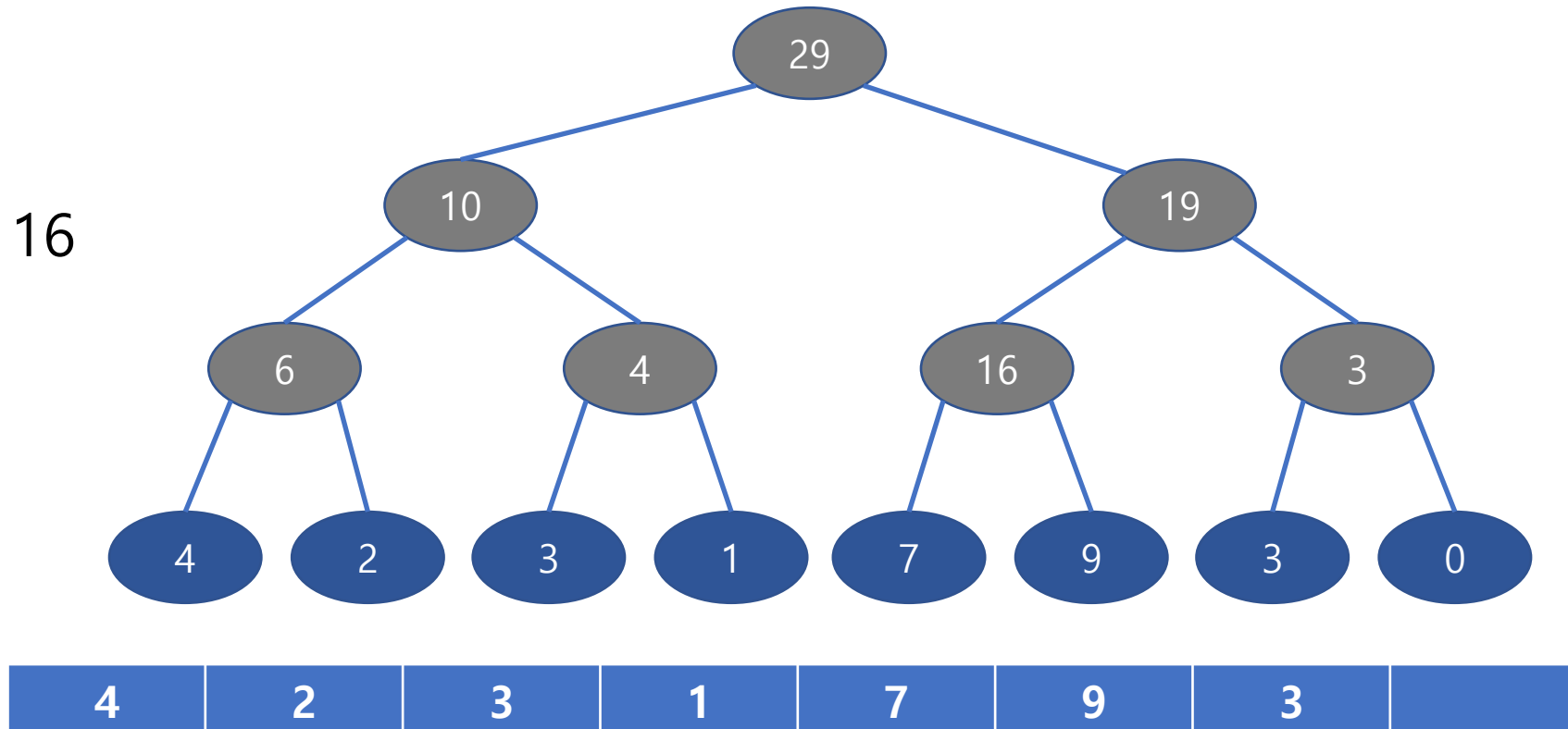
# Segment Tree 구현

$n = 7$   
Size = 16



# Segment Tree 구현

$n = 7$   
Size = 16



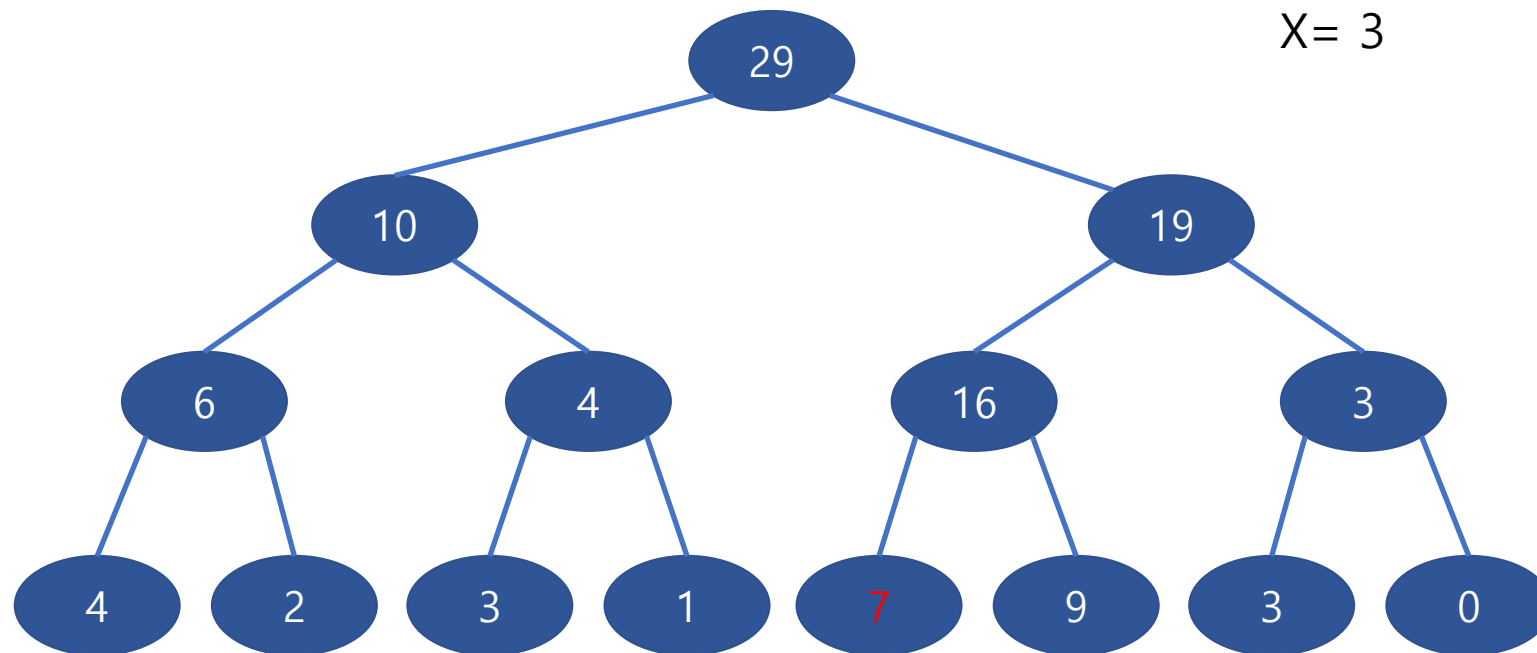
# Segment Tree 구현

배열의 index번 수를 x로 변경

```
1 void Update(int index, int x)
2 {
3     index += Size / 2 - 1;
4     SegTree[index] = x;
5     while(index > 1)
6     {
7         index /= 2;
8         SegTree[index] = SegTree[index*2] + SegTree[index*2 + 1];
9     }
10 }
11
12
13
```

# Segment Tree 구현

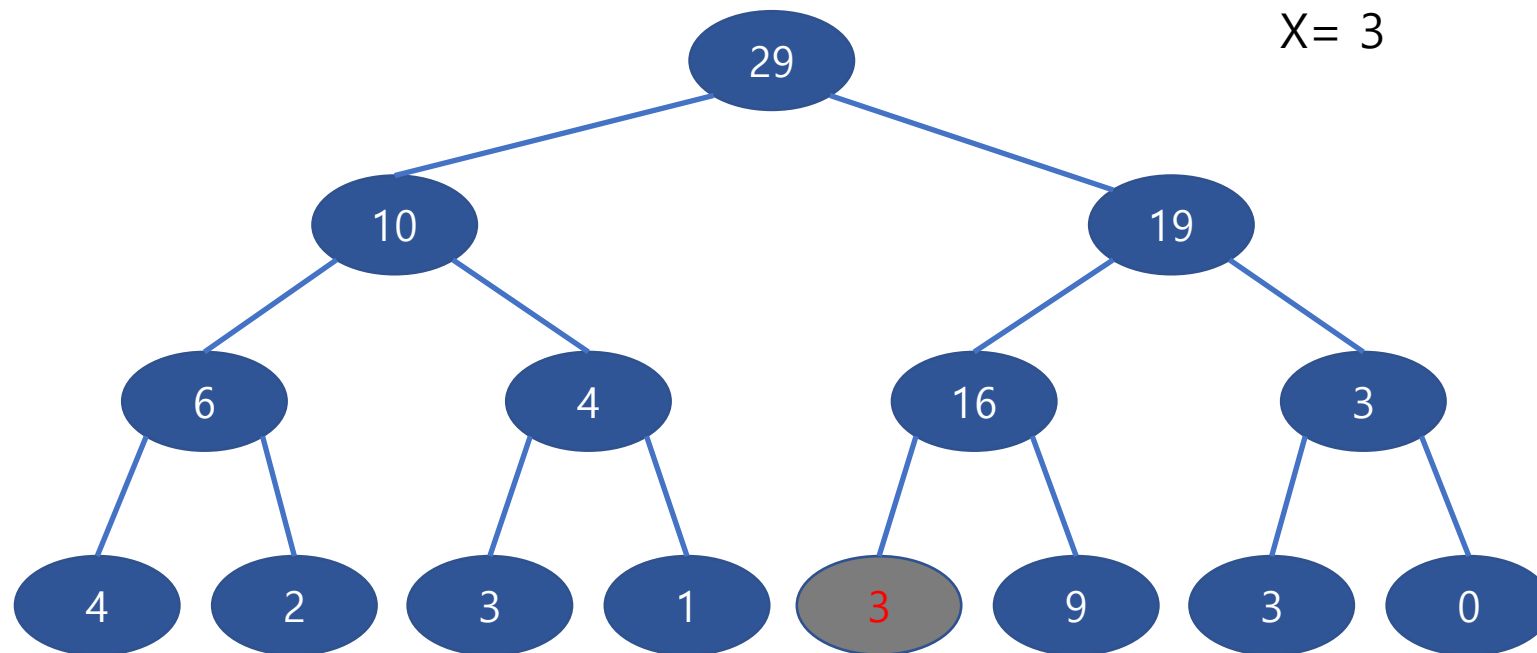
Index = 5  
X = 3



# Segment Tree 구현

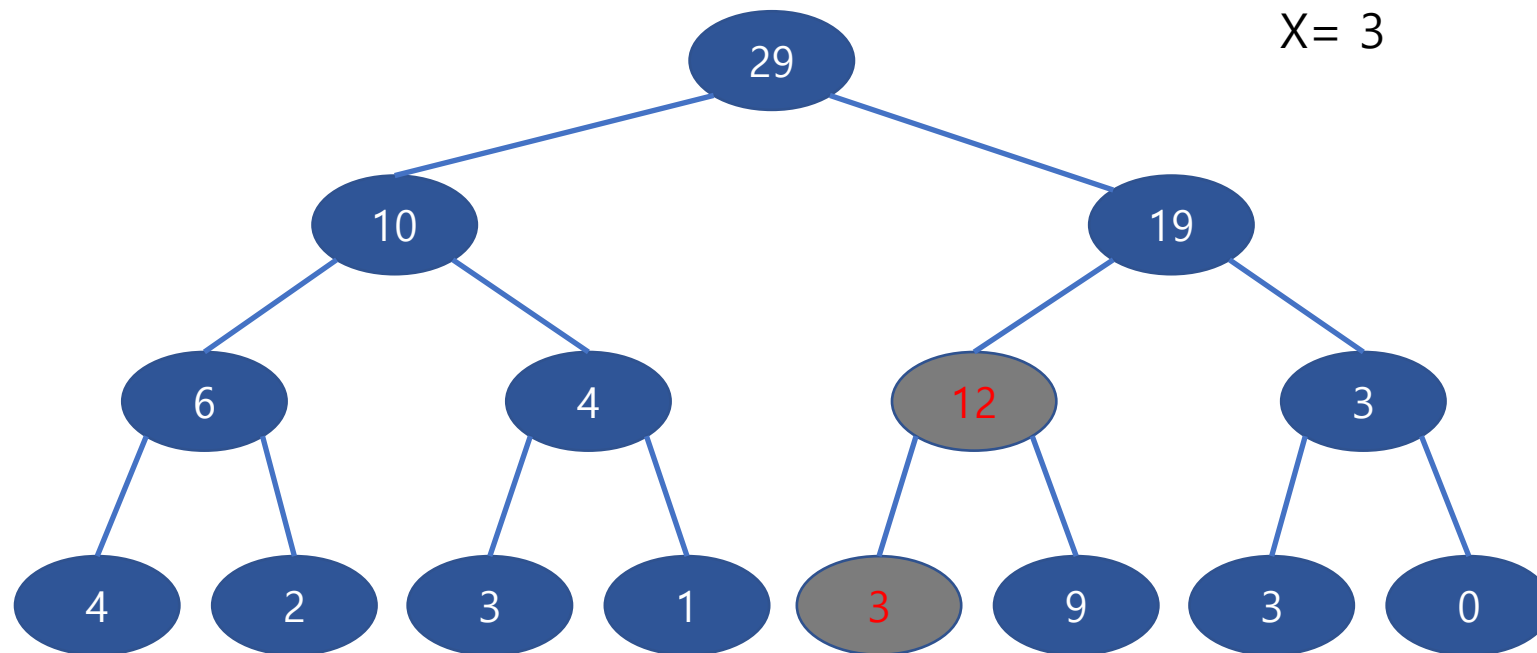
$$\text{Index} = 5 + 8 - 1 = 12$$

X = 3



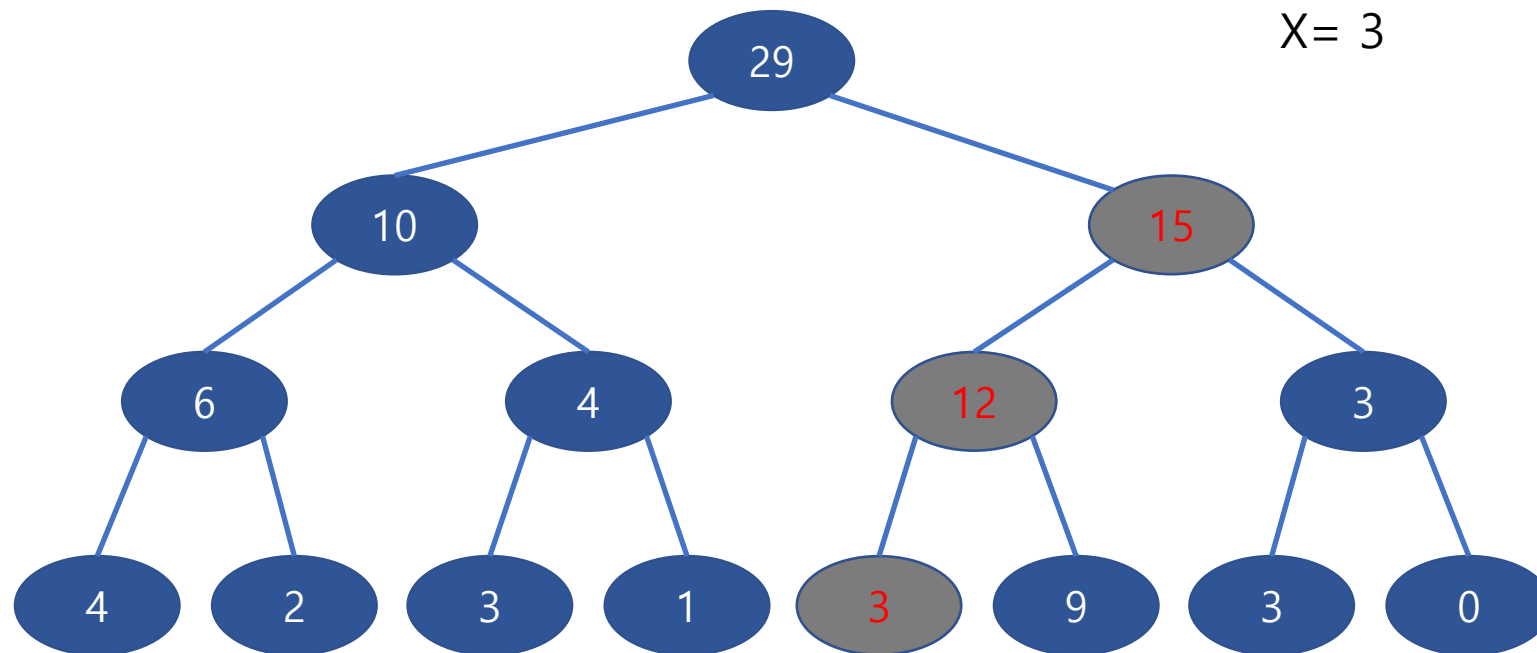
# Segment Tree 구현

$$\text{Index} = 12 / 2 = 6$$
$$X = 3$$



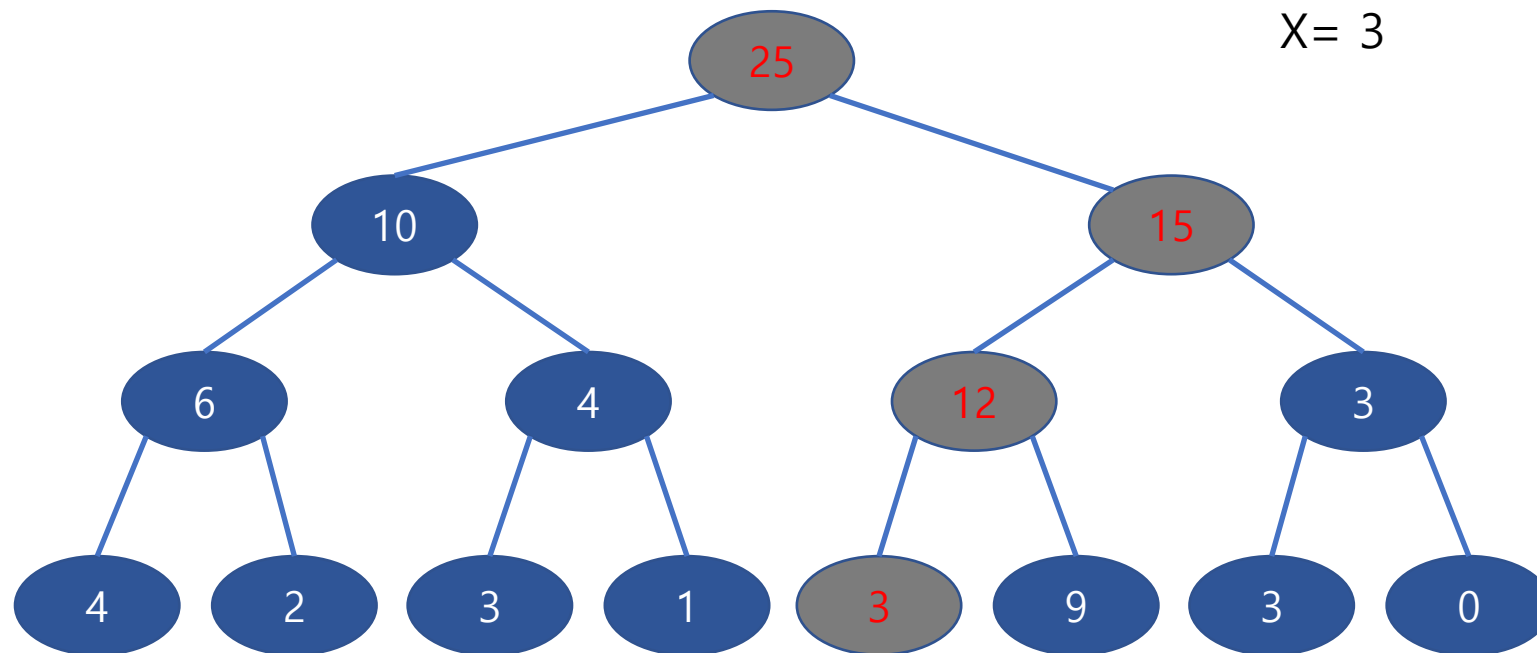
# Segment Tree 구현

$$\text{Index} = 6 / 2 = 3$$
$$X = 3$$



# Segment Tree 구현

$$\text{Index} = 3 / 2 = 1$$
$$X = 3$$





# Segment Tree 구현

배열의 [L, R]의 합을 리턴

now\_L, now\_R = SegTree[index]가  
표현하는 구간을 나타냄

```
1 int Sum(int L, int R, int now_L, int now_R, int index)
2 {
3     if (L > now_R || R < now_L) return 0;
4     if (L <= now_L && now_R <= R) return SegTree[index];
5
6     return Sum(L, R, now_L, (now_L + now_R) / 2, index*2)
7         + Sum(L, R, (now_L + now_R) / 2 + 1, now_R, index*2 + 1);
8 }
9
10
11
12
13
```

# Segment Tree 구현

배열의 [L, R]의 합을 리턴

now\_L, now\_R = SegTree[index]가  
표현하는 구간을 나타냄

```
1 int Sum(int L, int R, int now_L, int now_R, int index)
2 {
3     if (L > now_R || R < now_L) return 0;
4     if (L <= now_L && now_R <= R) return SegTree[index];
5     return Sum(L, R, (now_L + now_R) / 2, (now_L + now_R) / 2 + 1, index*2)
6         + Sum(L, R, (now_L + now_R) / 2 + 1, now_R, index*2 + 1);
7 }
8
9
10
11
12
13
```

현재 SegTree[index]가 표현하는 구간이  
내가 찾으려는 구간을 포함하지 않으면 0 리턴

# Segment Tree 구현

배열의 [L, R]의 합을 리턴

now\_L, now\_R = SegTree[index]가  
표현하는 구간을 나타냄

```
1 int Sum(int L, int R, int now_L, int now_R, int index)
2 {
3     if (L > now_R || R < now_L) return 0;
4     if (L <= now_L && now_R <= R) return SegTree[index];
5
6     return S      현재 SegTree[index]가 표현하는 구간이
7                     내가 찾으려는 구간에 완전히 포함되면 SegTree[index] 리턴 );
8 }
9
10
11
12
13
```

# Segment Tree 구현

배열의 [L, R]의 합을 리턴

now\_L, now\_R = SegTree[index]가  
표현하는 구간을 나타냄

```
1 int Sum(int L, int R, int now_L, int now_R, int index)
2 {
3     if (L > now_R || R < now_L) return 0;
4     if (L <= now_L && now_R <= R) return SegTree[index];
5
6     return Sum(L, R, now_L, (now_L + now_R) / 2, index*2)
7         + Sum(L, R, (now_L + now_R) / 2 + 1, now_R, index*2 + 1);
8 }
9
10
11
12
13
```

어느 경우에도 속하지 않으면 내 자식 노드에서 탐색  
(구간을 둘로 쪼갬)

# Segment Tree 구현

배열의 [L, R]의 합을 리턴

now\_L, now\_R = SegTree[index]가  
표현하는 구간을 나타냄

```
1
2     cin >> L >> R;
3     cout << Sum(L, R, 1, Size / 2, 1);
4
5     return 0;
6 }
7
8
9
10
11
12
13
```

# Segment Tree 구현

배열의 [L, R]의 합을 리턴

now\_L, now\_R = SegTree[index]가  
표현하는 구간을 나타냄

```
1  
2     cin >> L >> R;  
3     cout << Sum(L, R, 1, Size / 2, 1);  
4  
5     return 0;  
6 }  
7  
8  
9  
10  
11  
12  
13
```

Size / 2 는 리프 노드의 개수를 나타냄

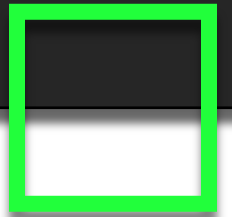


# 연습 문제

2042 : 구간 합 구하기  
2357 : 최소값과 최대값

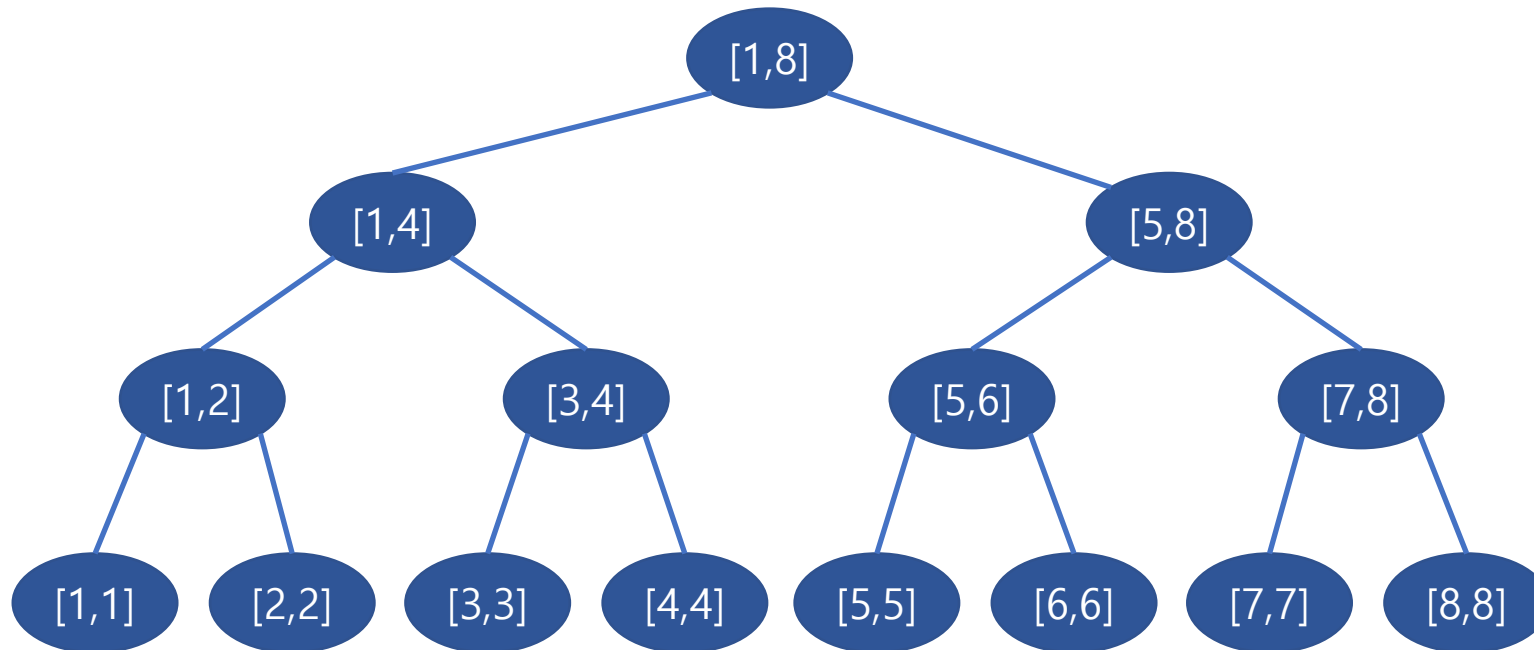
#2

# Fenwick Tree

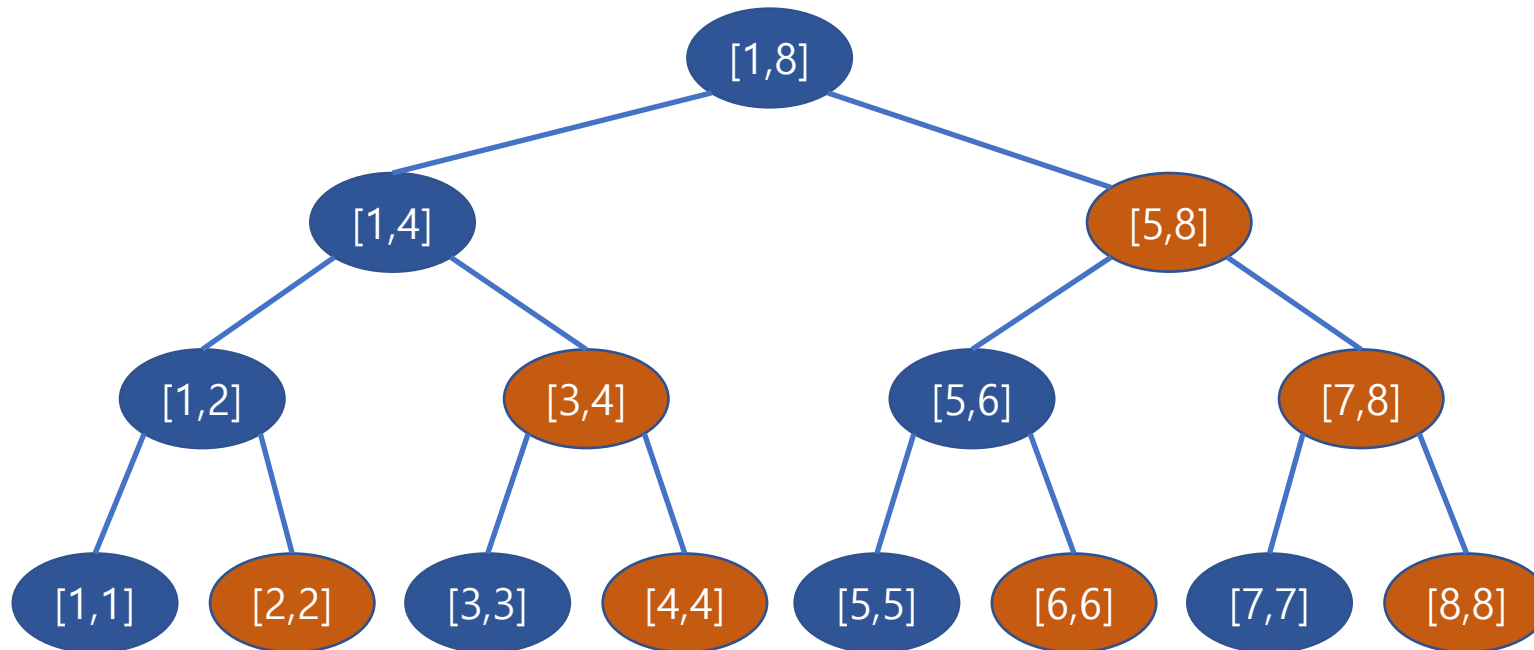




# Fenwick Tree(Binary Indexed Tree)



# Fenwick Tree(Binary Indexed Tree)

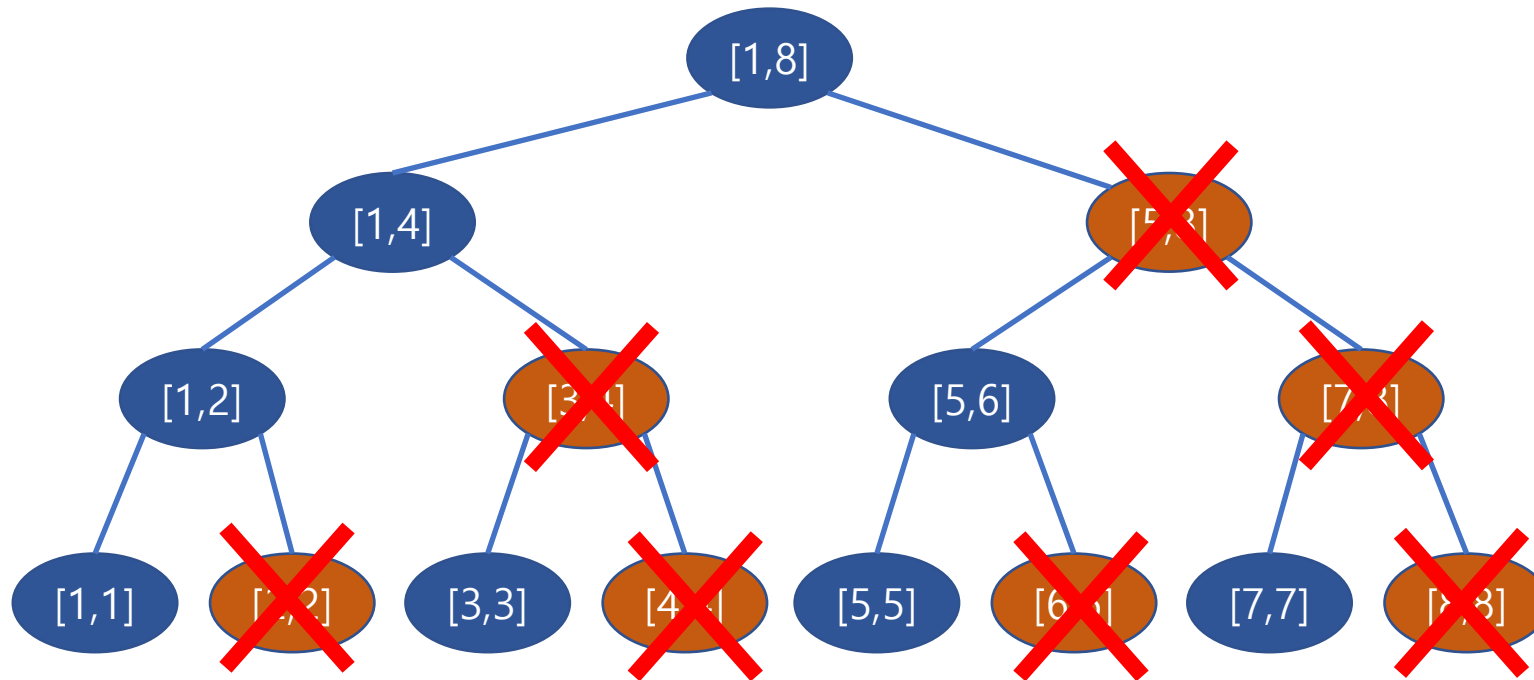




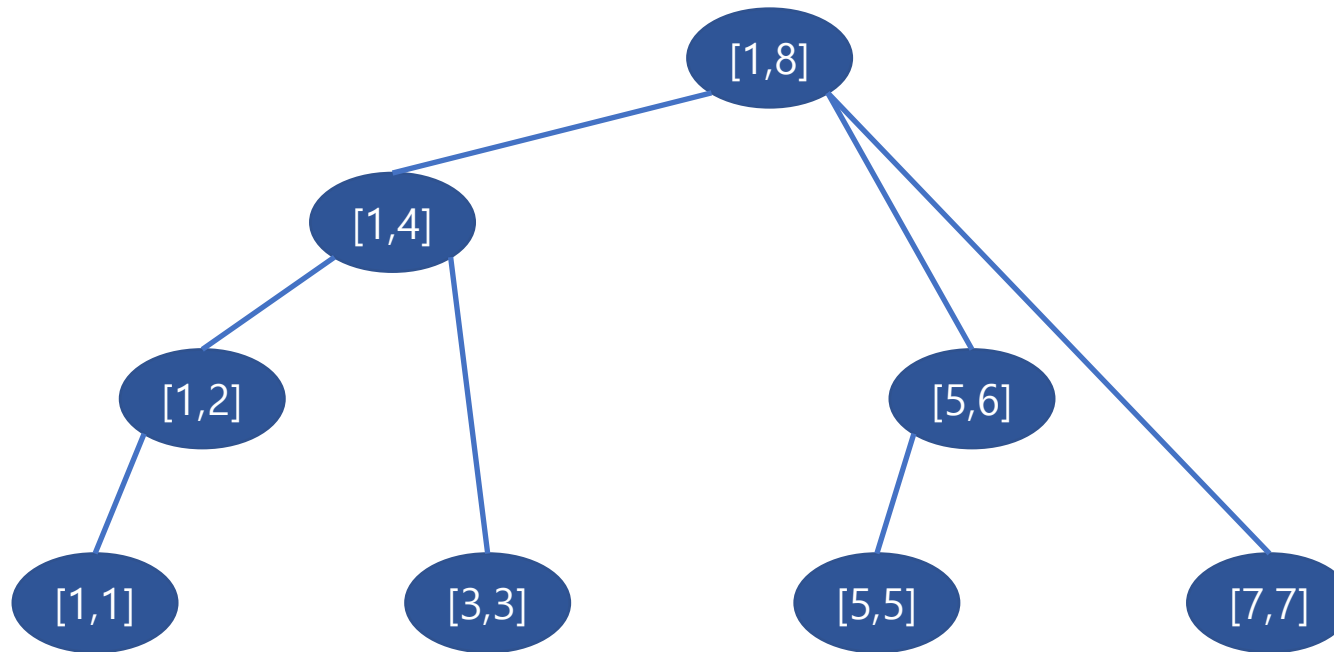
# Fenwick Tree(Binary Indexed Tree)

같은 구간을 표현하는 노드들이 있는데,  
이것들을 없애면 어떨까?

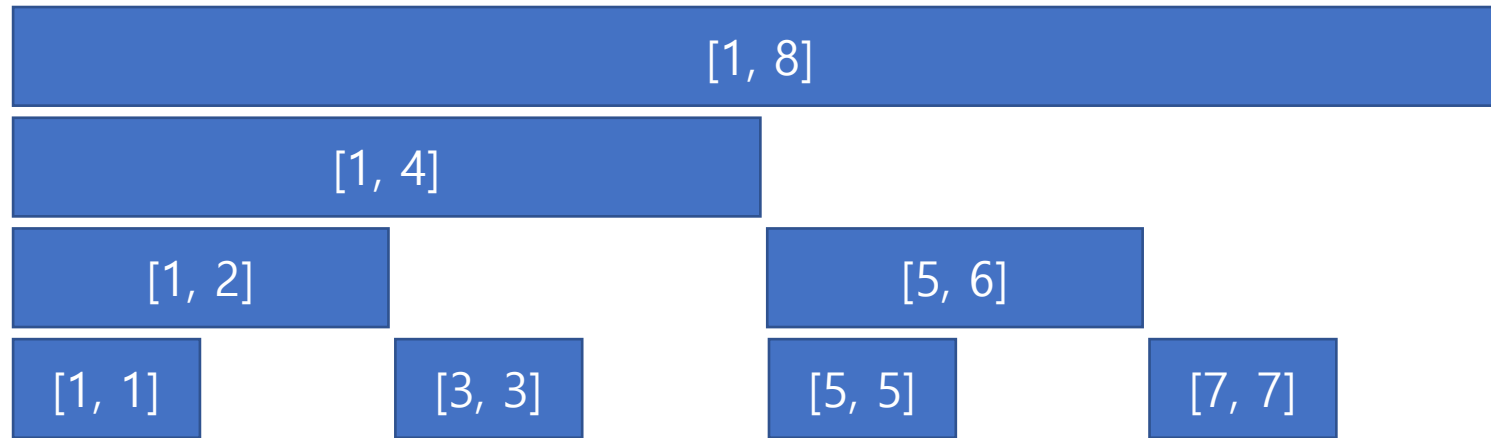
# Fenwick Tree(Binary Indexed Tree)



# Fenwick Tree(Binary Indexed Tree)

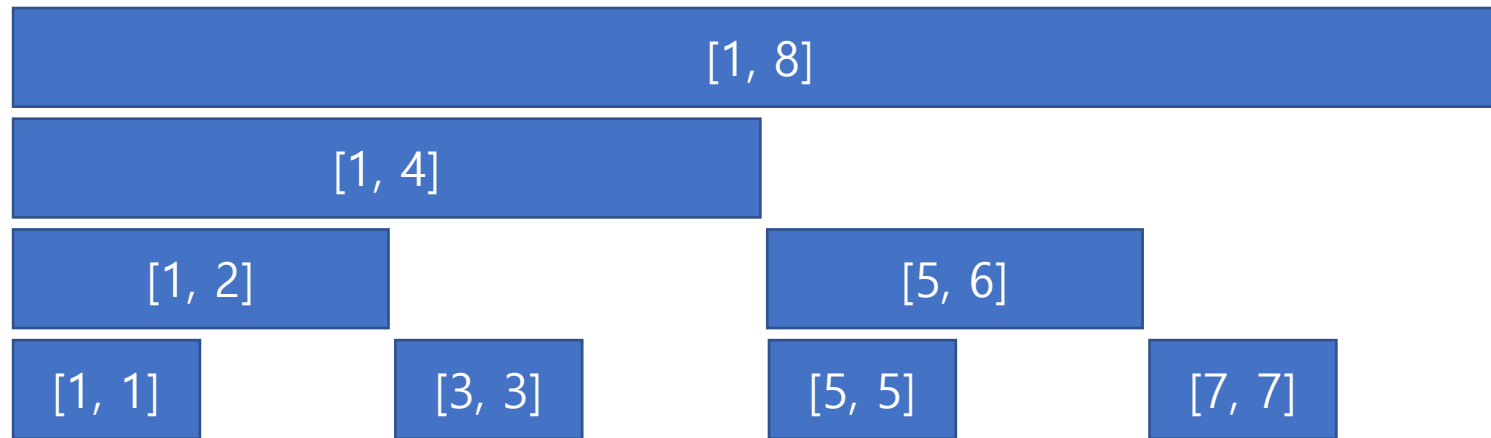


# Fenwick Tree(Binary Indexed Tree)



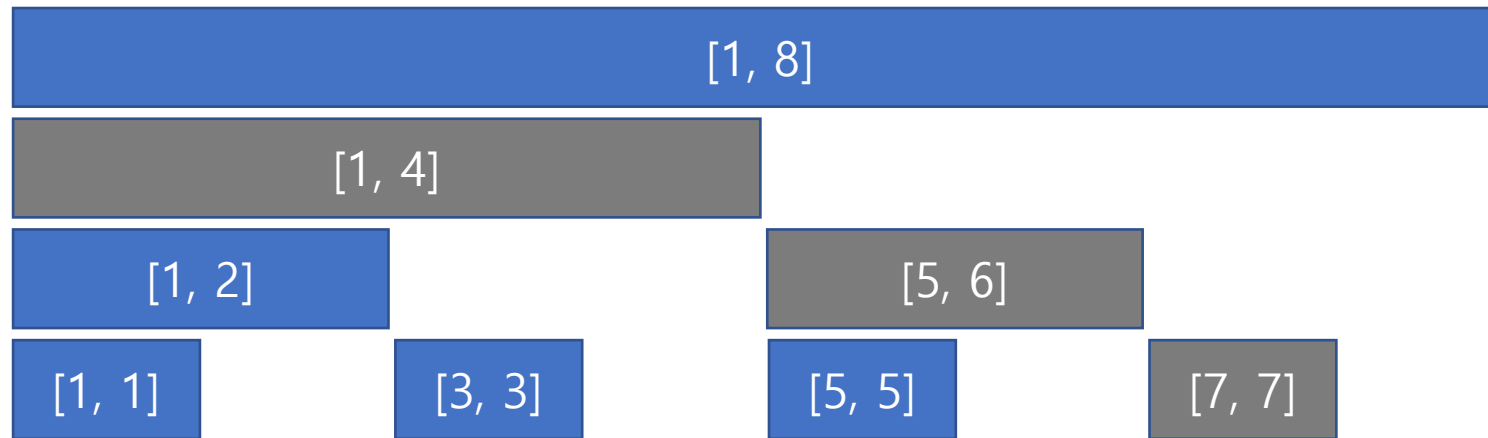
# Fenwick Tree(Binary Indexed Tree)

1~7 까지의 구간합



# Fenwick Tree(Binary Indexed Tree)

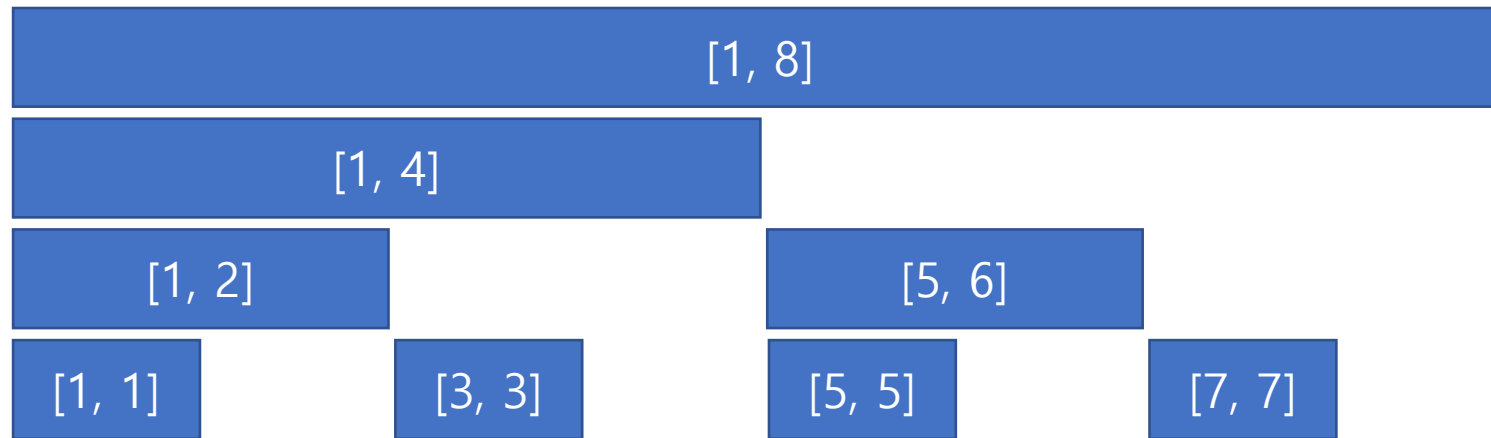
1~7 까지의 구간합





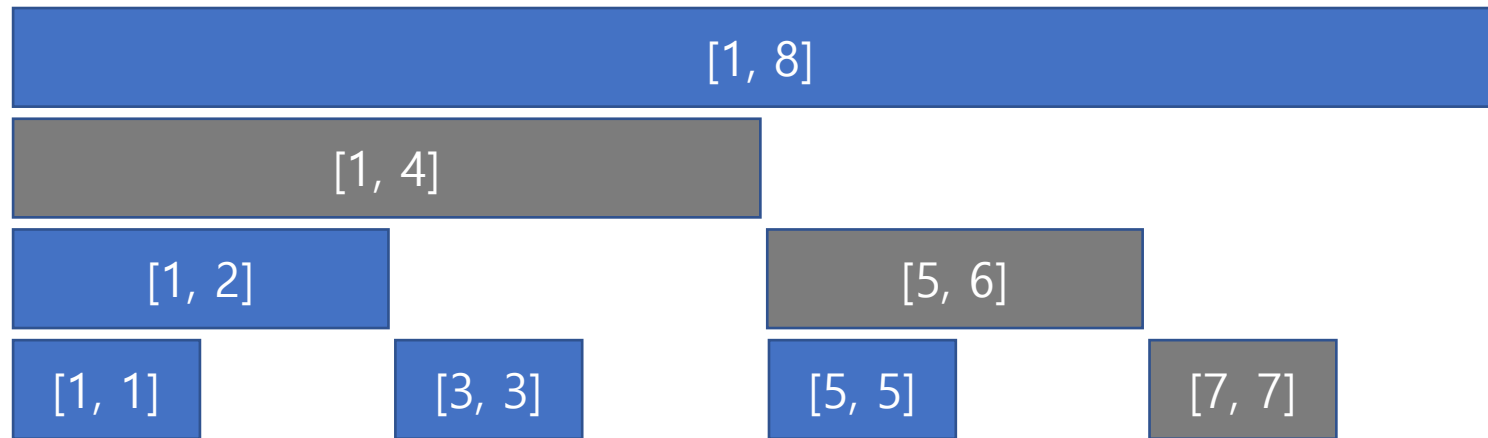
# Fenwick Tree(Binary Indexed Tree)

3~7 까지의 구간합



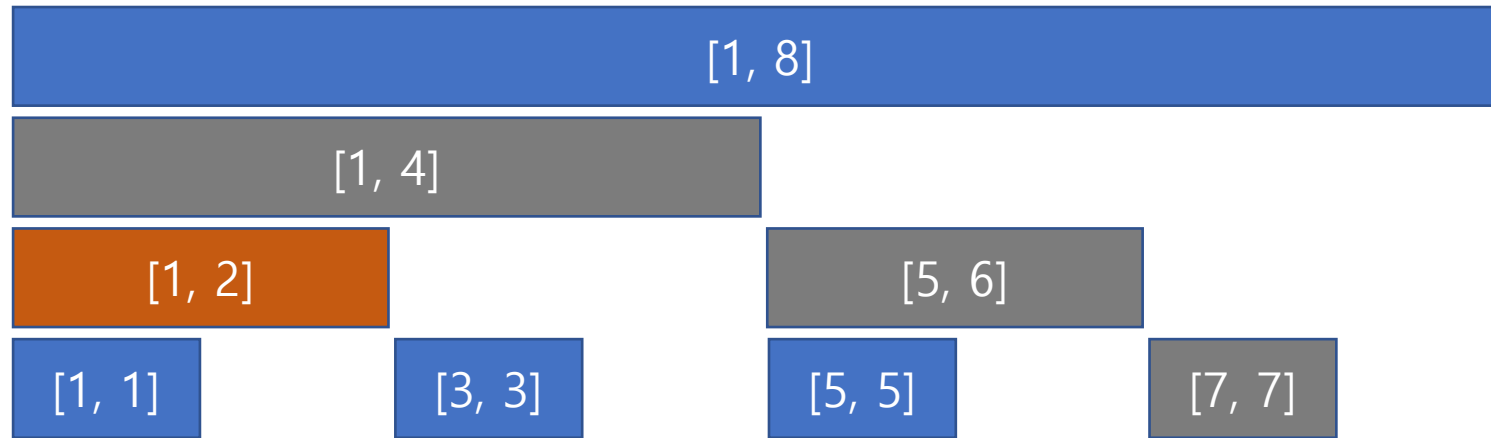
# Fenwick Tree(Binary Indexed Tree)

3~7 까지의 구간합



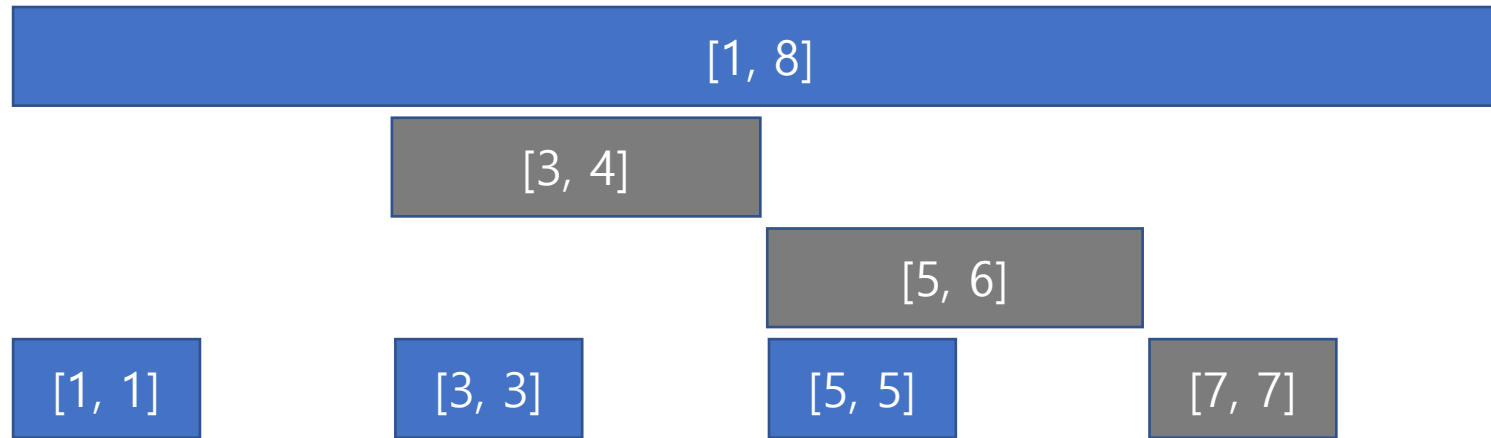
# Fenwick Tree(Binary Indexed Tree)

3~7 까지의 구간합

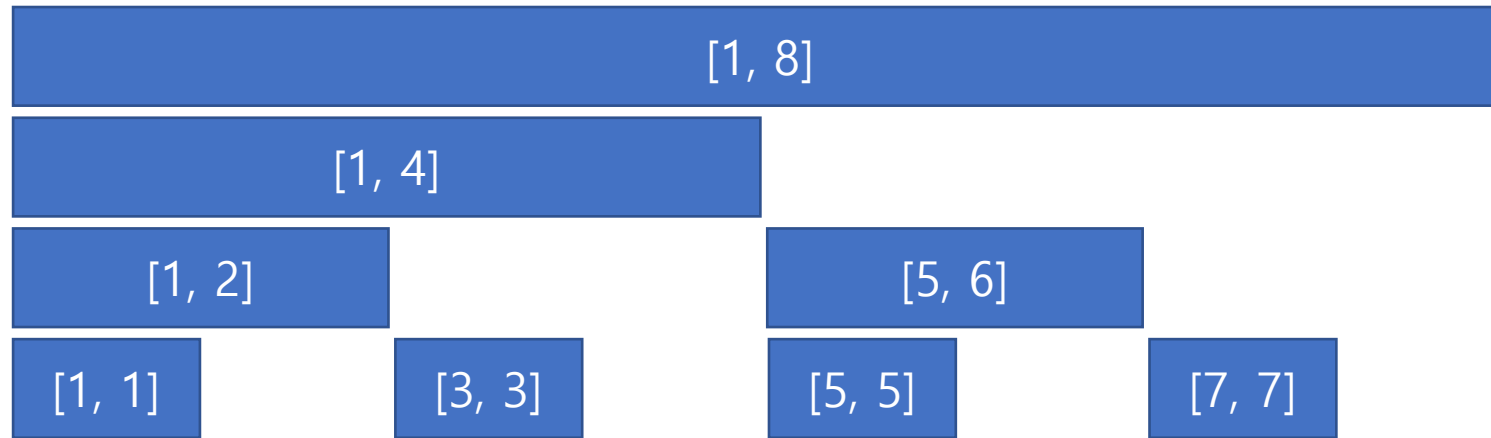


# Fenwick Tree(Binary Indexed Tree)

3~7 까지의 구간합

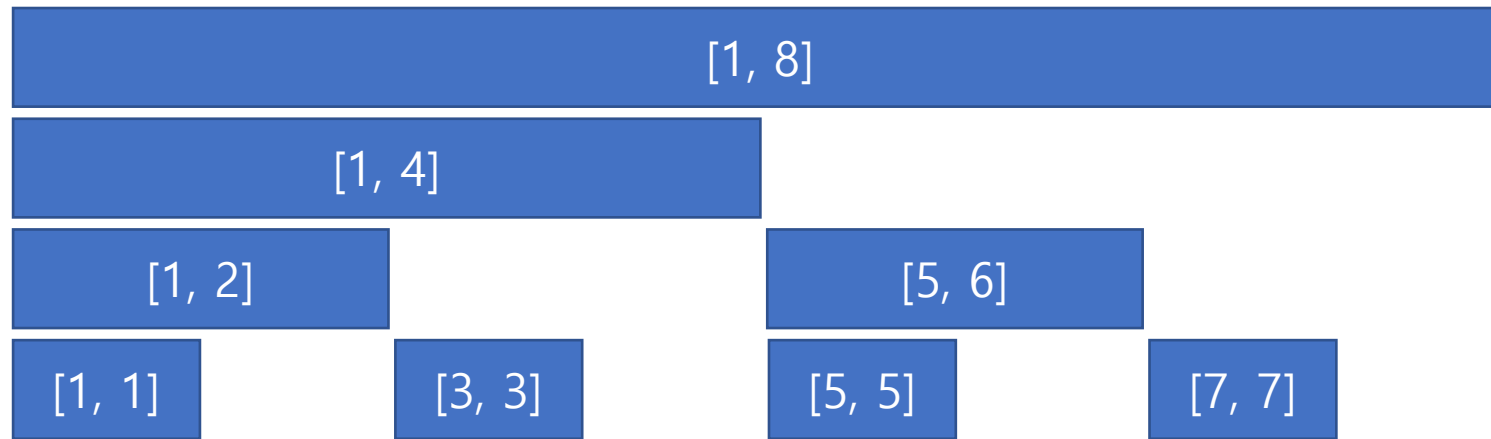


# Fenwick Tree(Binary Indexed Tree)



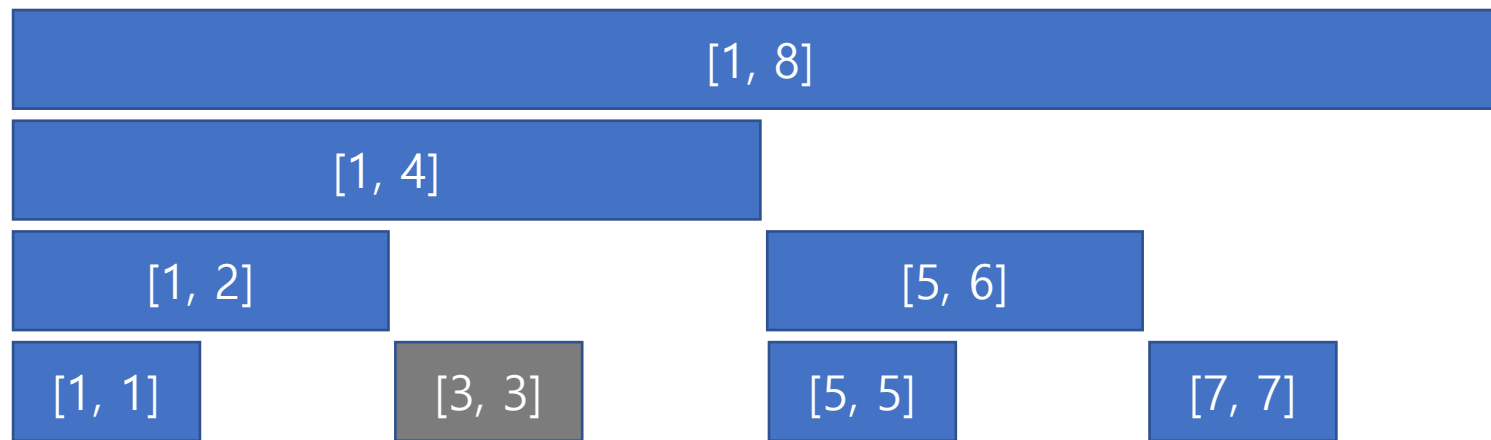
# Fenwick Tree(Binary Indexed Tree)

3번 인덱스 값 변경



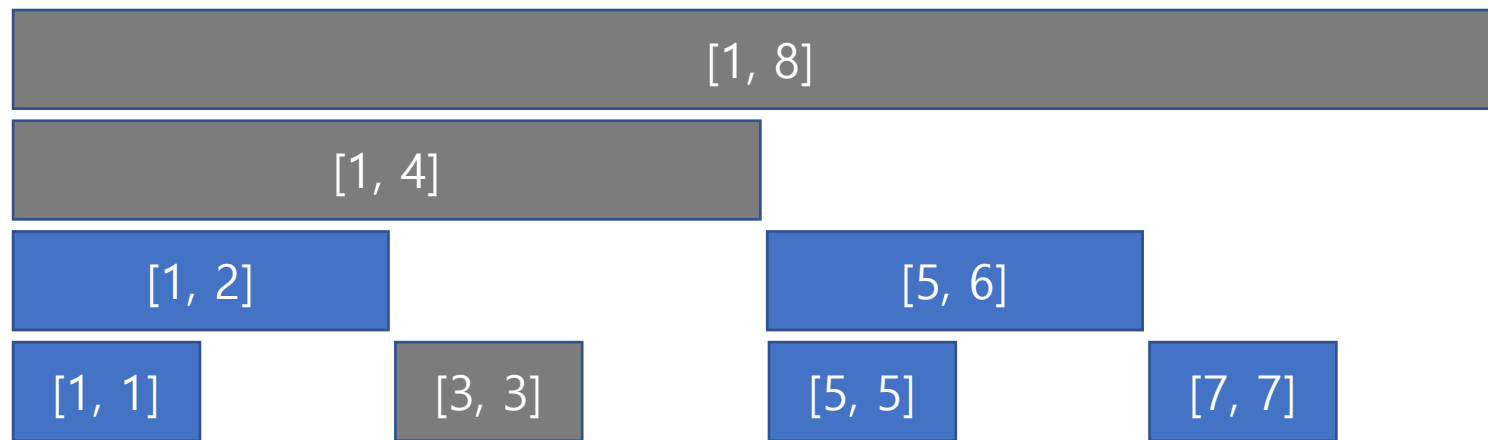
# Fenwick Tree(Binary Indexed Tree)

3번 인덱스 값 변경



# Fenwick Tree(Binary Indexed Tree)

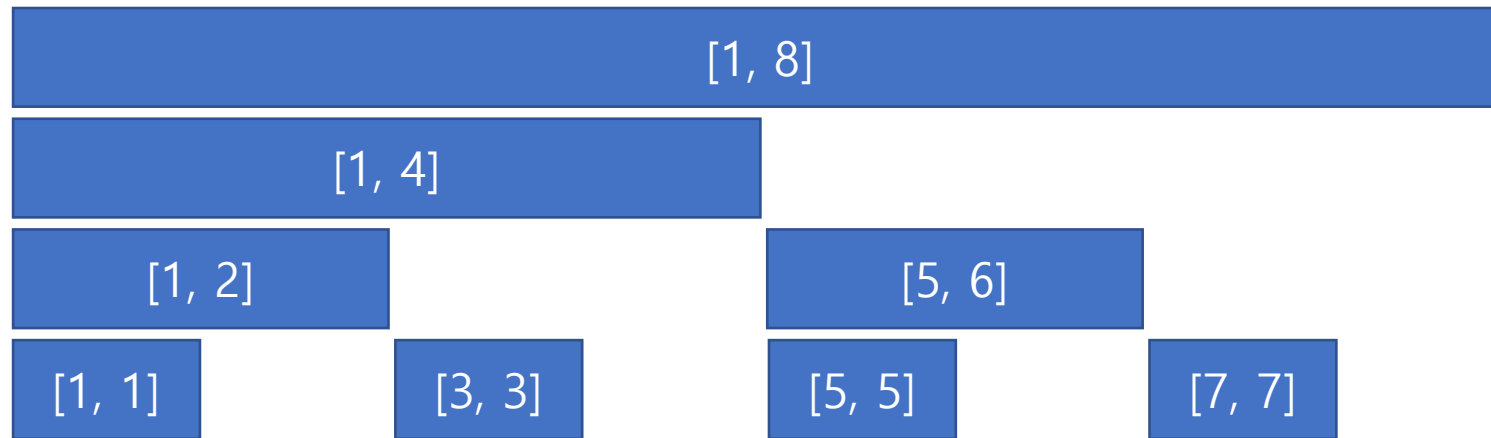
3번 인덱스 값 변경





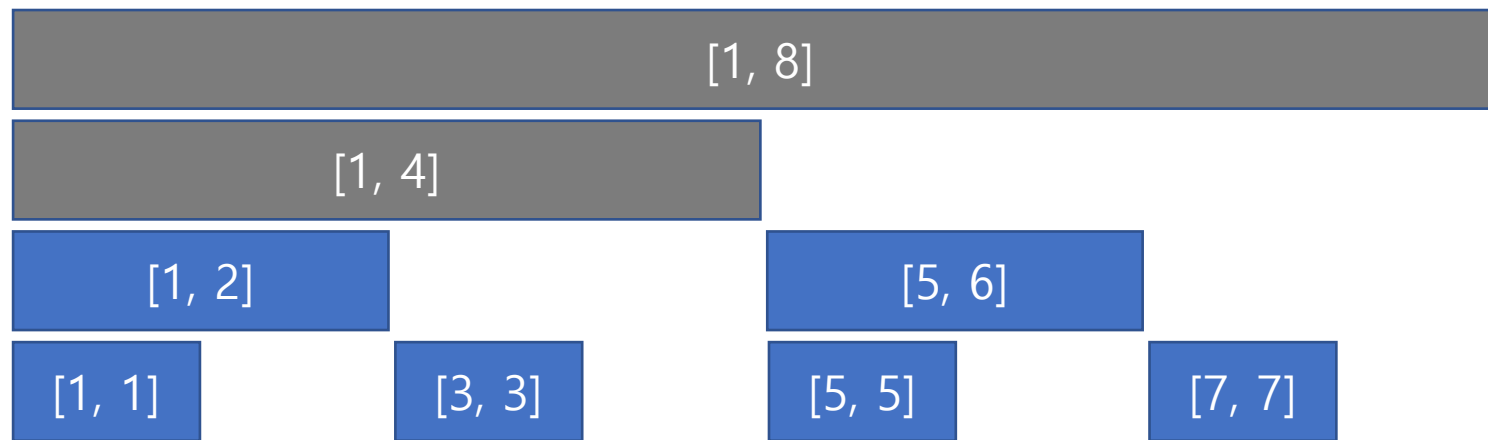
# Fenwick Tree(Binary Indexed Tree)

4번 인덱스 값 변경



# Fenwick Tree(Binary Indexed Tree)

4번 인덱스 값 변경

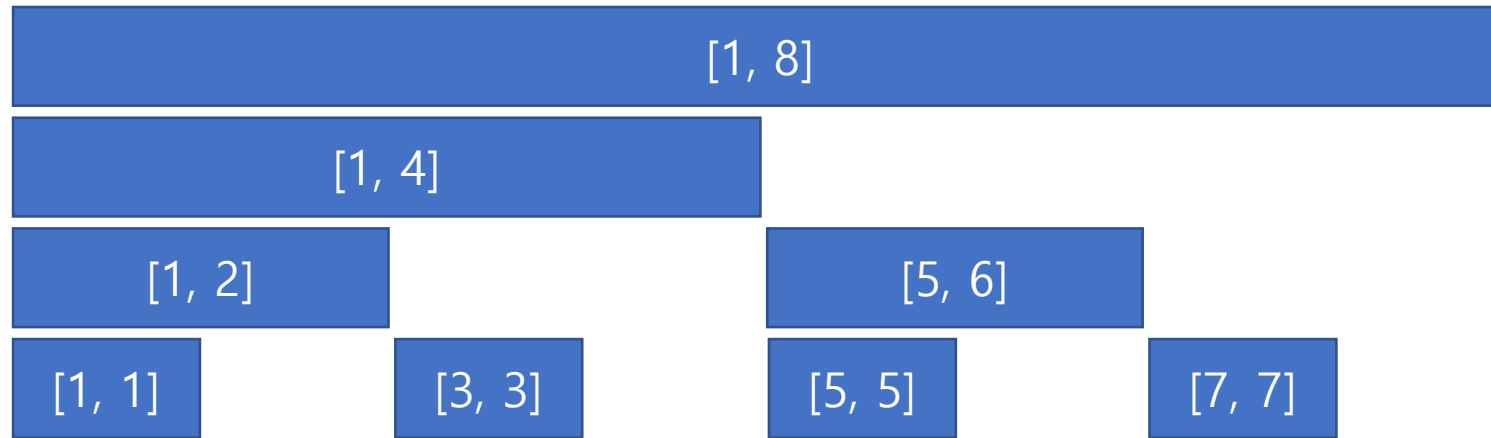




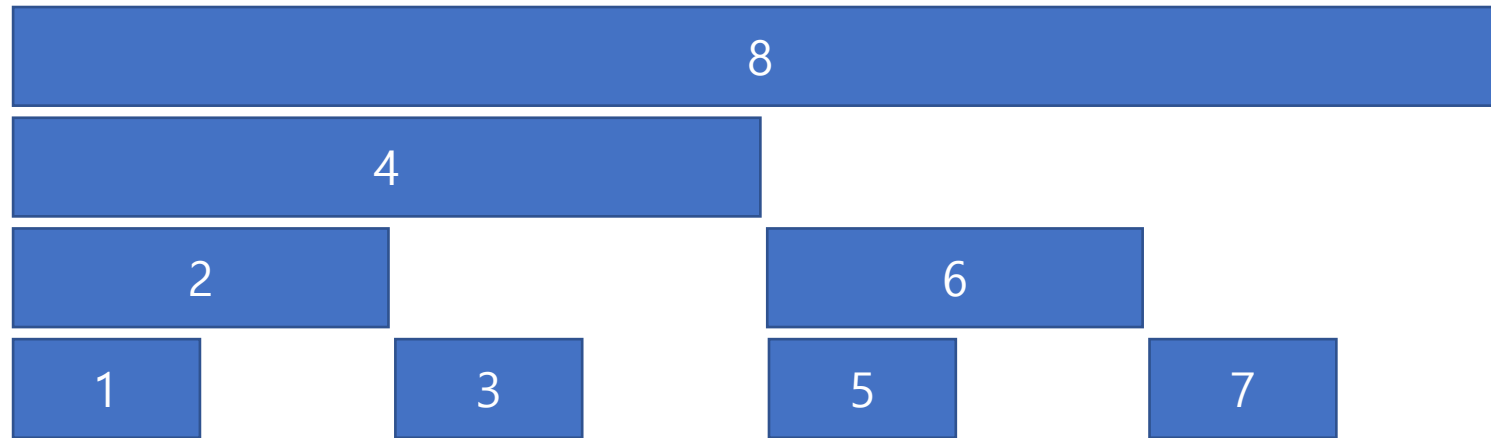
# Fenwick Tree(Binary Indexed Tree)

구간합, 업데이트 둘 다 가능한거 확인했고,  
공간복잡도 줄어든건 알겠는데,  
고작 이게 전부??

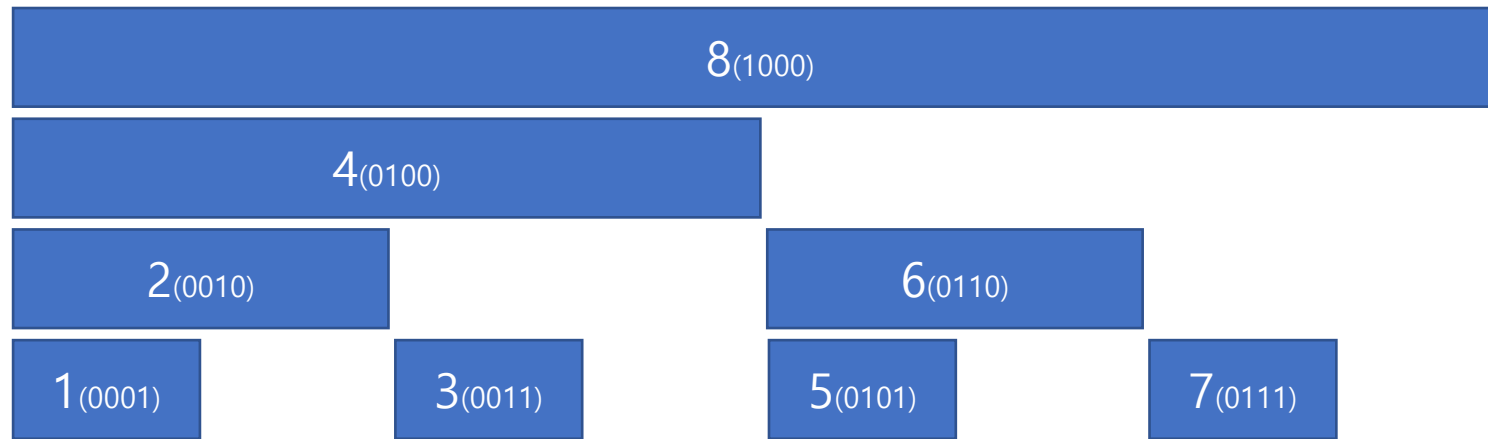
# Fenwick Tree(Binary Indexed Tree)



# Fenwick Tree(Binary Indexed Tree)

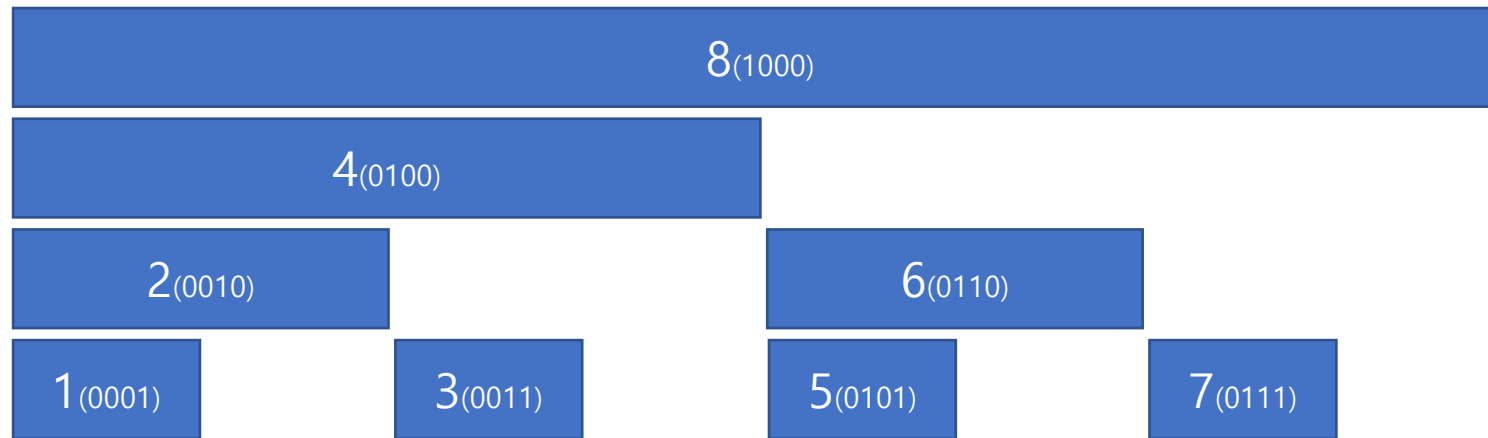


# Fenwick Tree(Binary Indexed Tree)



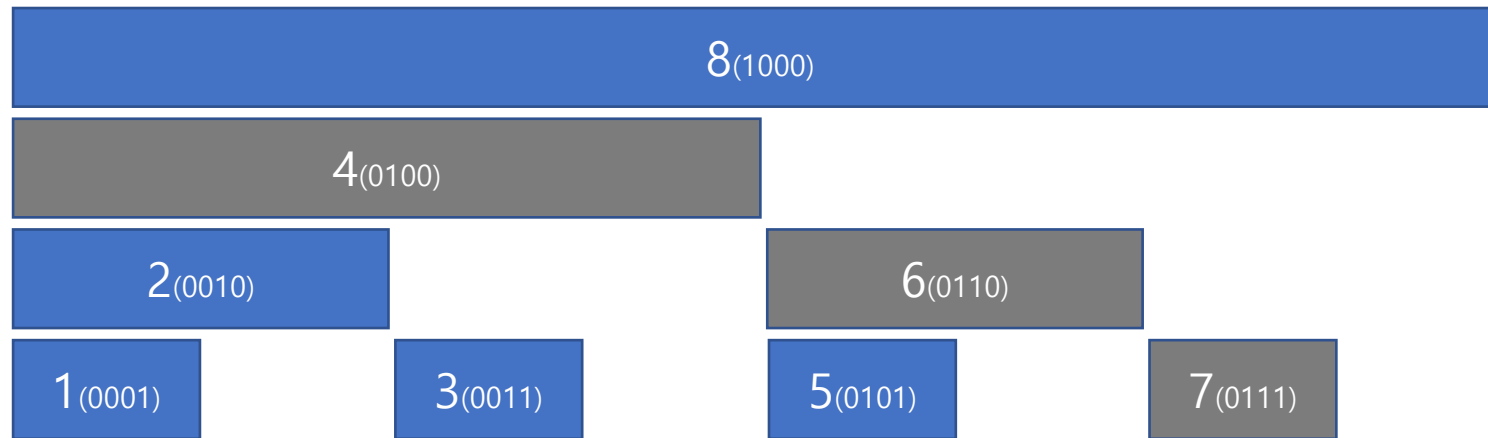
# Fenwick Tree(Binary Indexed Tree)

1~7 까지의 구간합



# Fenwick Tree(Binary Indexed Tree)

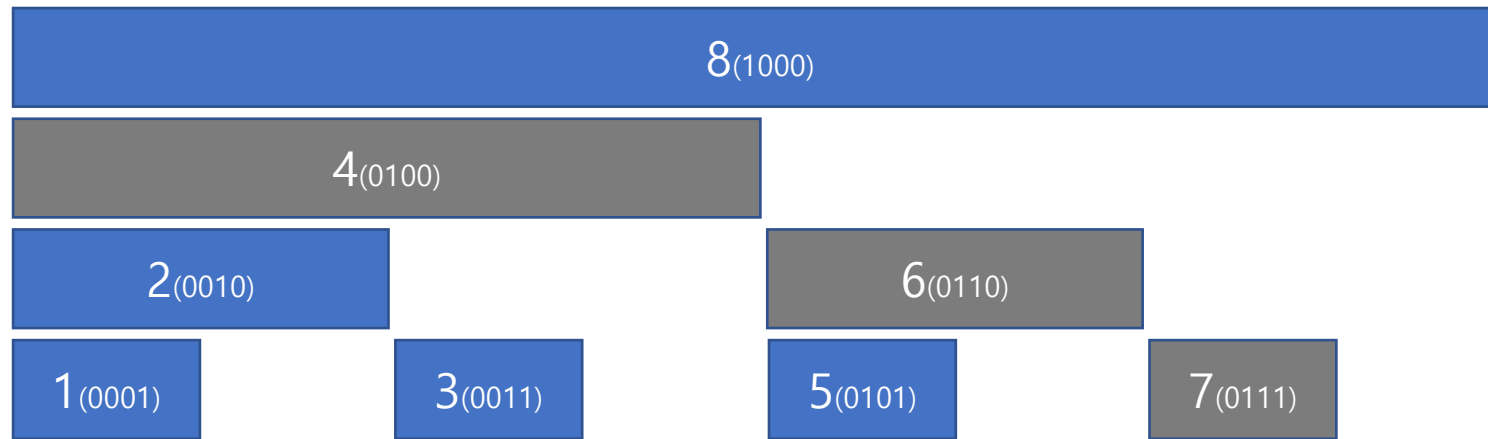
1~7 까지의 구간합





# Fenwick Tree(Binary Indexed Tree)

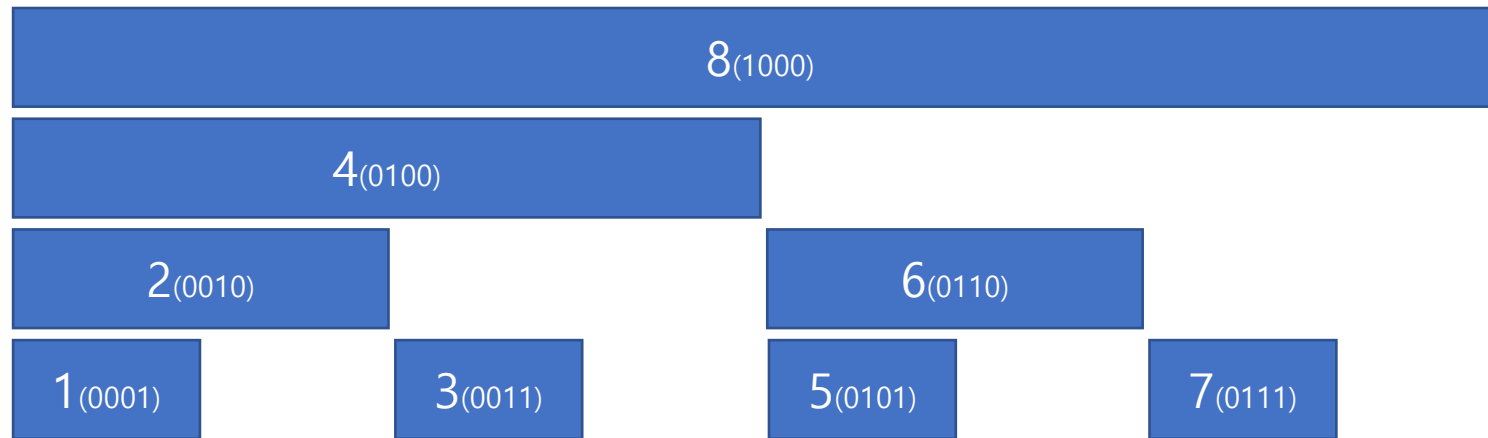
1~7 까지의 구간합



0111 -> 0110 -> 0100

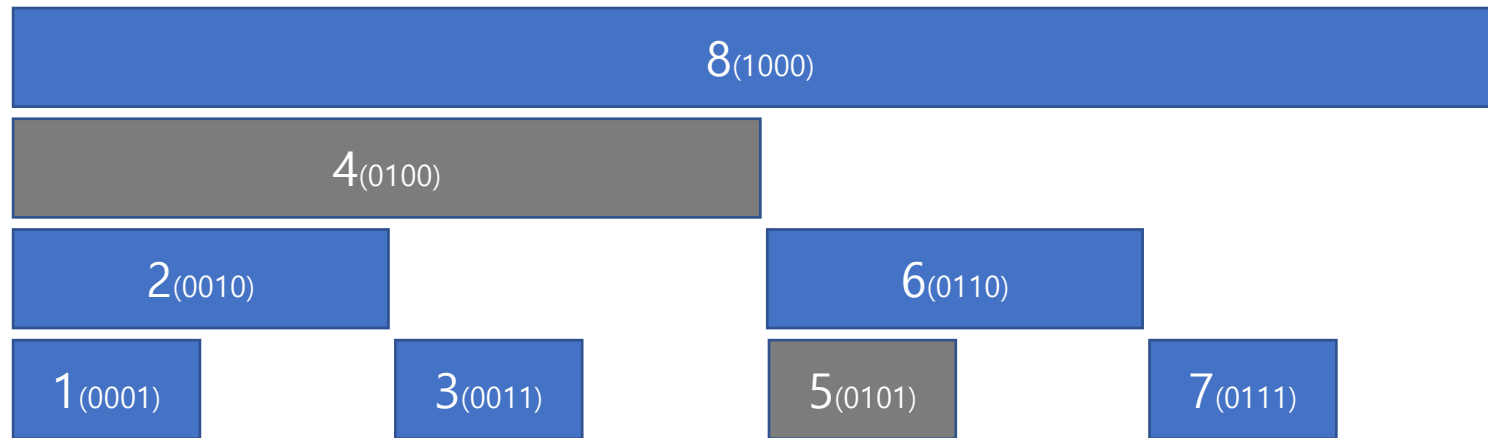
# Fenwick Tree(Binary Indexed Tree)

1~5 까지의 구간합



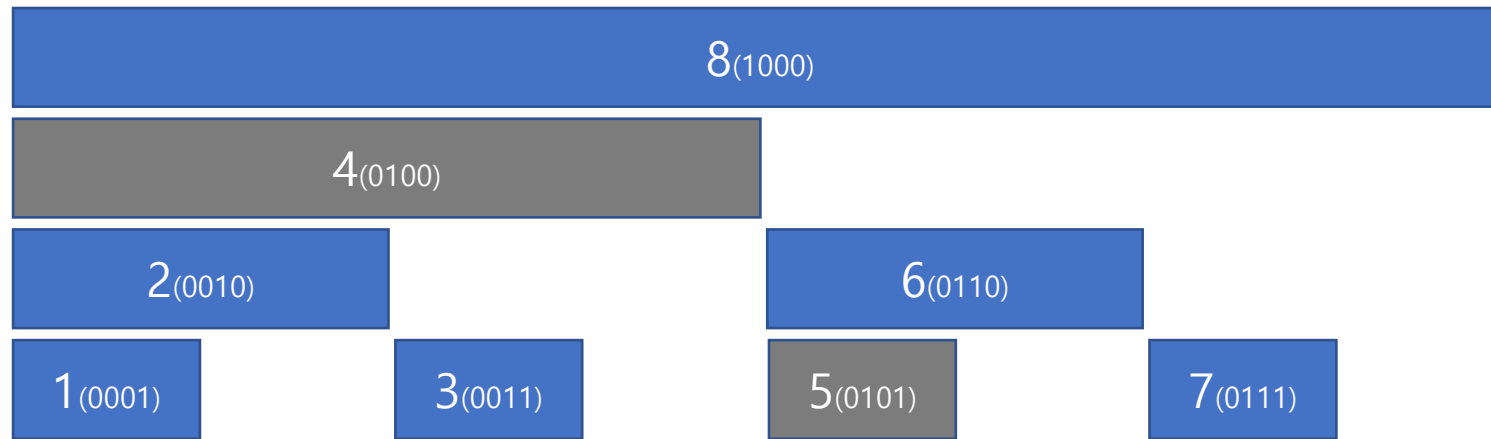
# Fenwick Tree(Binary Indexed Tree)

1~5 까지의 구간합



# Fenwick Tree(Binary Indexed Tree)

1~5 까지의 구간합



0101 -> 0100



# Fenwick Tree(Binary Indexed Tree)

가장 오른쪽에 있는 1(최하위 비트)가  
0으로 바뀌고 있다!!

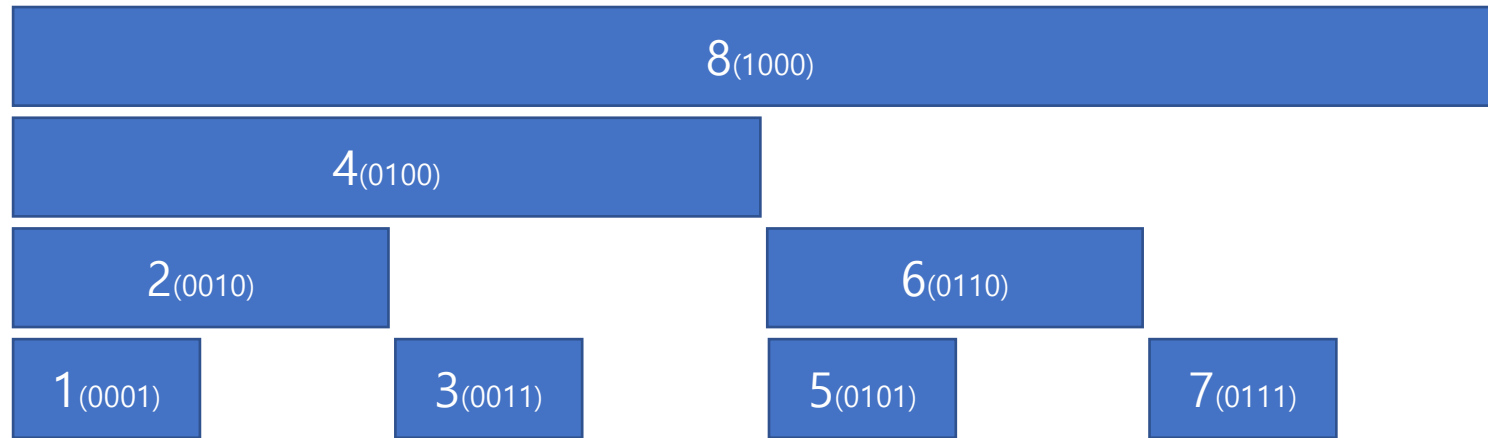
# Fenwick Tree(Binary Indexed Tree)

0111 -> 0110 -> 0100

0101 -> 0100

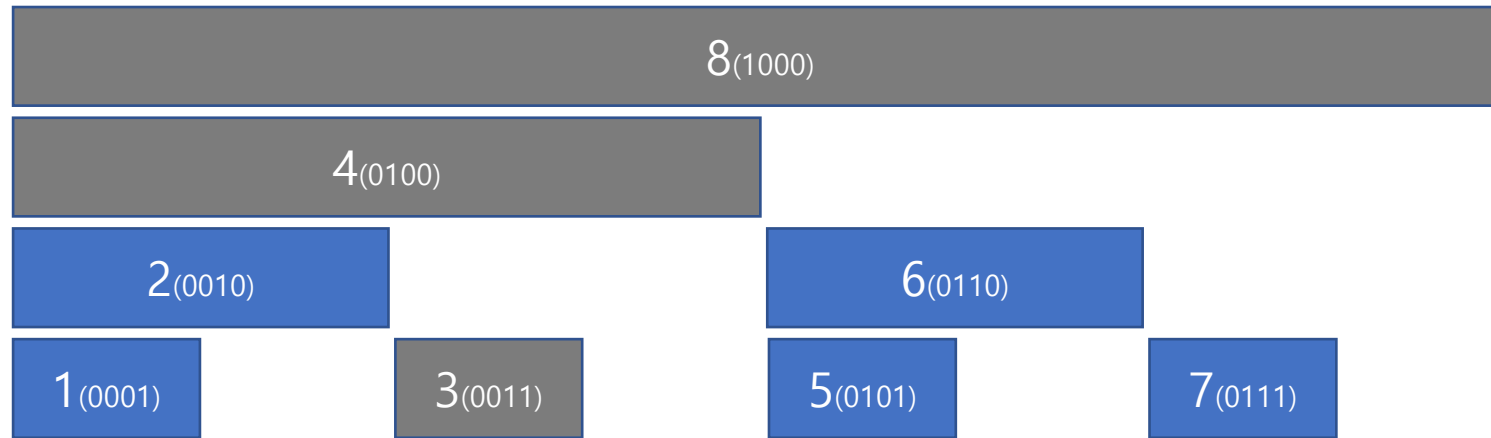
# Fenwick Tree(Binary Indexed Tree)

3번 index 업데이트



# Fenwick Tree(Binary Indexed Tree)

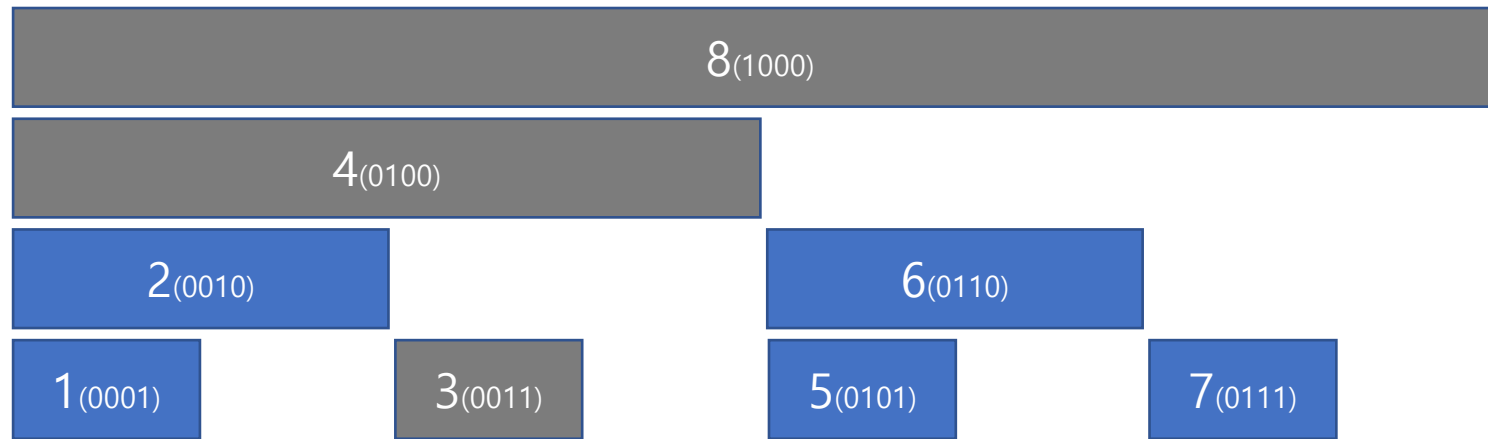
3번 index 업데이트





# Fenwick Tree(Binary Indexed Tree)

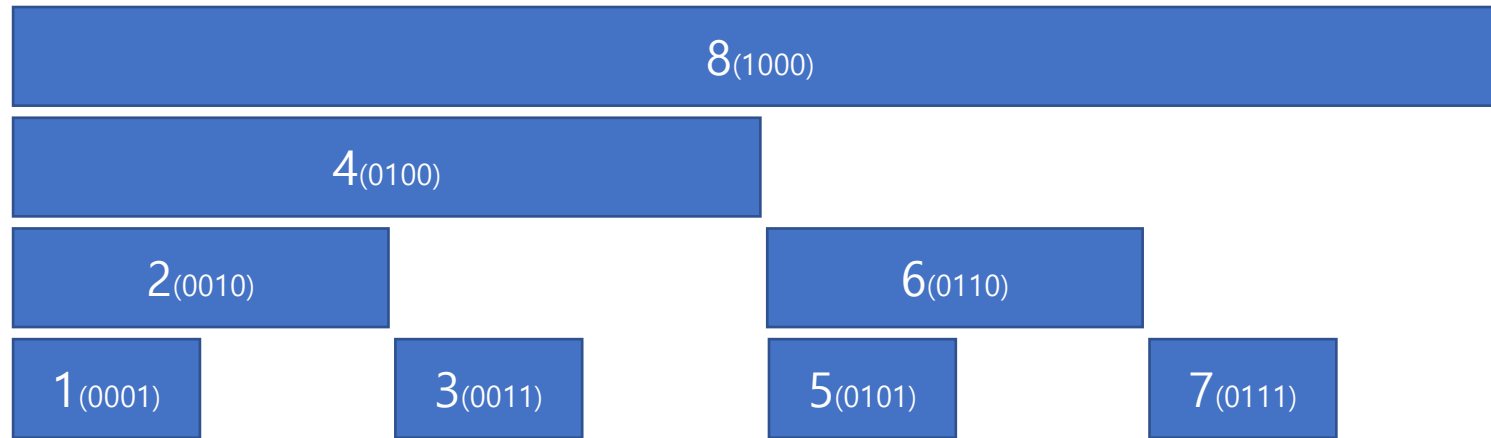
3번 index 업데이트



0011 -> 0100 -> 1000

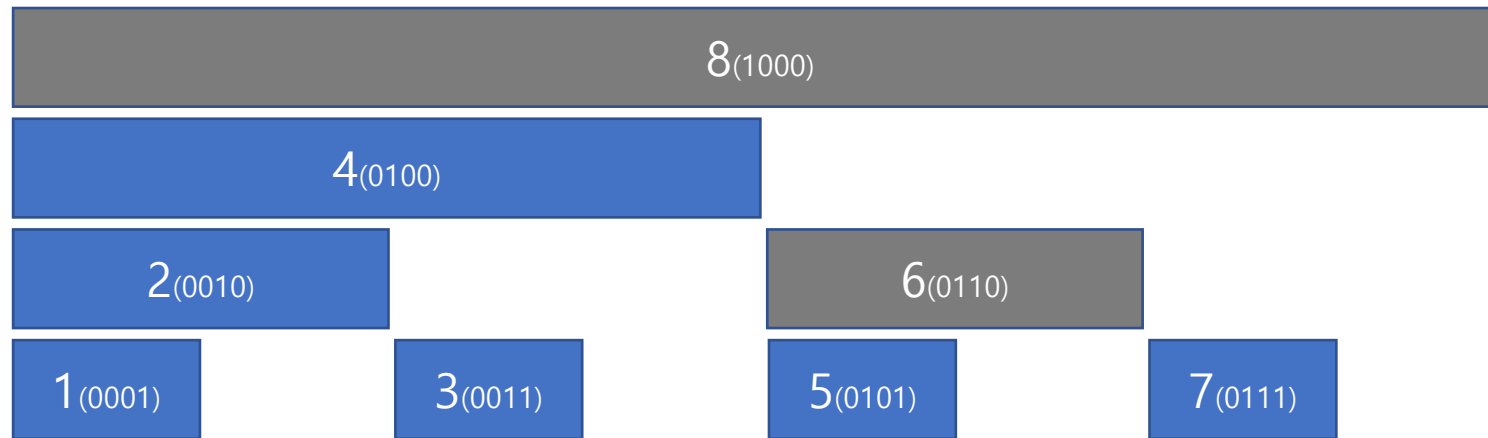
# Fenwick Tree(Binary Indexed Tree)

6번 index 업데이트



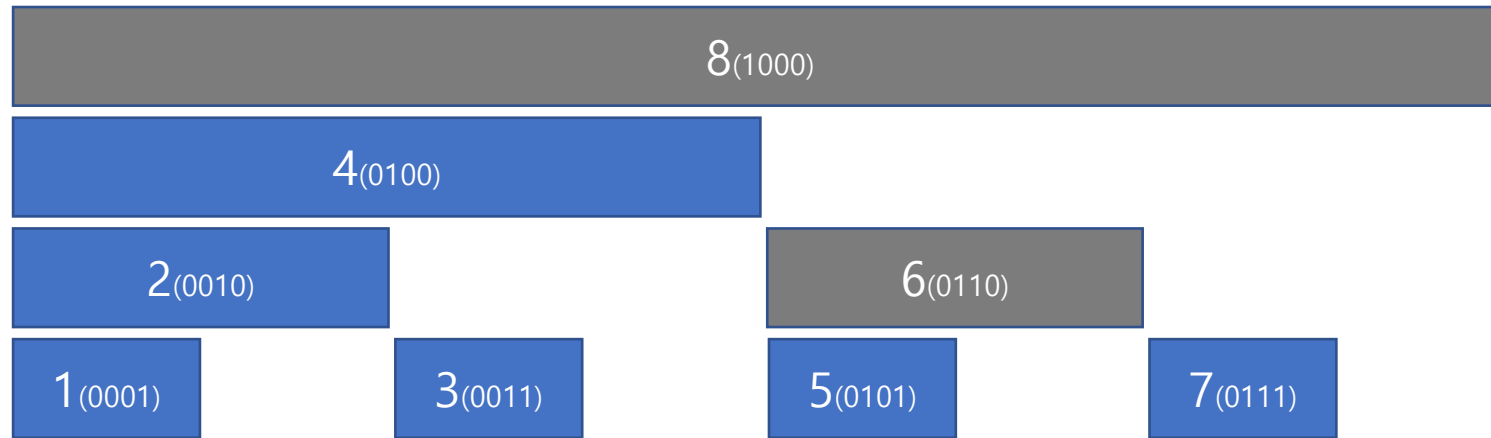
# Fenwick Tree(Binary Indexed Tree)

6번 index 업데이트



# Fenwick Tree(Binary Indexed Tree)

6번 index 업데이트



0110 -> 1000



# Fenwick Tree(Binary Indexed Tree)

가장 오른쪽 최하위 비트가  
1씩 더해지고 있다.

# Fenwick Tree(Binary Indexed Tree)

+ (0001)      + (0100)  
0011 -> 0100 -> 1000

+ (0010)  
0110 -> 1000



# Fenwick Tree(Binary Indexed Tree)

Fenwick Tree의 핵심, 최/하위 비트

# Fenwick Tree(Binary Indexed Tree)

```
1 void Update(int index, int diff)
2 {
3     while(index <= n)
4     {
5         Tree[index] += diff;
6         index += (index & -index);
7     }
8 }
9
10
11
12
13
14
15
```



# Fenwick Tree(Binary Indexed Tree)

```
1 void Update(int index, int diff)
2 {
3     while(index <= n)
4     {
5         Tree[index] += diff;
6         index += (index & -index);
7     }
8 }
```

Fenwick Tree는 업데이트를 할 때 원본과 변경 값의 차이를 더해주는 방식으로 진행된다.

# Fenwick Tree(Binary Indexed Tree)

```
1 void Update(int index, int diff)
2 {
3     while(index <= n)
4     {
5         Tree[index] += diff;
6         index += (index & -index);
7     }
8 }
```

최하위 비트는  $(i \& -i)$  연산으로 구할 수 있다.



# Fenwick Tree(Binary Indexed Tree)

$\text{index} += (\text{index} \& -\text{index})$



# Fenwick Tree(Binary Indexed Tree)

$\text{index} += (\text{index} \& -\text{index})$

컴퓨터는 음수를 표현할 때 2의 보수를 사용한다.

2의 보수 = 1의 보수 + 1

1의 보수 = 모든 비트를 뒤집은 수

# Fenwick Tree(Binary Indexed Tree)

Ex) 5의 2의 보수

$$5 = (0101)$$

$$1\text{의 보수} = (1010)$$

$$2\text{의 보수} = (1010) + (0001) = (1011)$$

# Fenwick Tree(Binary Indexed Tree)

Ex) (5 & -5)

2의 보수 = (1010) + (0001) = (1011)

(0101) & (1011) = (0001)

# Fenwick Tree(Binary Indexed Tree)

Ex) (6 & -6)

2의 보수 = (1001) + (0001) = (1010)

(0110) & (1010) = (0010)

# Fenwick Tree(Binary Indexed Tree)

```
1 int Sum(int index)
2 {
3     int sum = 0;
4     while (index > 0)
5     {
6         sum += Tree[index];
7         index -= (index & -index);
8     }
9     return sum;
10 }
11
12
13
14
15
```



# Fenwick Tree(Binary Indexed Tree)

```
1 int Sum(int index)
2 {
3     int sum = 0;
4     while (index > 0)
5     {
6         sum += Tree[index];
7         index -= (index & -index);
8     }
9     return sum;
10 }
11
12
13
14
15
```

최하위 비트를 제거해주는 과정

# Fenwick Tree(Binary Indexed Tree)

```
1
2     Tree.resize(n + 1);
3
4     for(int i = 1; i <= n; i++)
5     {
6         cin >> d[i];
7         Update(i, d[i]);
8     }
9
10
11
12
13
14
15
```

Fenwick Tree를 초기화 할 때는 Update를 이용한다.

# Fenwick Tree(Binary Indexed Tree)

```
1
2     cin >> L >> R;
3     cout << Sum(R) - Sum(L - 1);
4
5     cin >> index >> x;
6     Update(index, x - d[index]);
7     d[index] = x;
8 }
```

구간합을 구할 땐 L, R 이 주어지면,  
 $\text{Sum}(R) - \text{Sum}(L-1)$  모양으로 사용한다.

업데이트를 할 땐 두 수의 차이를 더해준다.



# 연습 문제

2042 : 구간 합 구하기  
12837 : 가계부(Hard)

# 과제

- 11505 : 구간 곱 구하기
- 12015 : 가장 긴 증가하는 부분 수열 2
- 1275 : 커피숍2
- 3745 : 오름세
- 3006 : 터보소트
- 1280 : 나무 심기
- 3653 : 영화 수집
- 9345 : 디지털 비디오 디스크

끗!

