29th

Prefix Sum

Range Sum

- 배열이 주어지고 주어진 구간에 존재하는 수들을 전부 다 더해라
- [l, r] 구간이 주어진 경우 arr[l] + arr[l+1] + … + arr[r]로 구할 수 있다
- 배열에 최대 N개의 원소가 있을 수 있으므로 O(N)의 시간 복잡도를 따른다

Range Sum

- 이러한 구간합을 Q번 주어진다면 어떨까?
- 단순히 더하는 것을 Q번 반복한다면 O(NQ)의 시간 복잡도를 따를 것이다

- 10만개의 원소가 주어지고 10만개의 쿼리가 주어진다면 100억번의 연산이 필요할 것이다

Range Sum

- 전처리를 통하여 시간을 줄여보자
- 미리 일정 구간을 계산해둔다면 계산하는 과정을 줄일 수 있다

- 그렇다면 어느 구간을 계산해야 하는가?

Prefix Sum

- 전처리를 통하여 i번째 칸에 1번째 원소부터 i번째 원소까지 더한 값을 저장한다
- sum[i] = arr[1] + arr[2] + ... + arr[i];

Prefix Sum

- i번째까지 더해둔 값을 사용해서 구간 합을 구해보자
- [l, r]구간의 합은 arr[l] + arr[l + 1] + … + arr[r]이었다
- 미리 구해둔 값을 사용한다면 [l, r]구간의 합을 sum[r] – arr[1] - … - arr[l-1]로 나타낼 수 있다

Prefix Sum

- sum[r] – arr[1] - … - arr[l - 1]을 다시 정리하면 sum[r] - (arr[1] + … + arr[l - 1])로 나타낼 수 있다
- 괄호 안의 식은 1부터 l - 1까지 더한 것이므로 sum으로 표현할 수 있다
- 따라서 [l, r]구간의 합은 sum[r] - sum[l - 1]로 나타낼 수 있다

Prefix Sum

1 + … + arr[l-1] sum[l - 1]

1 + … + arr[r] sum[r]

1	**…**	**l-1**	**l**	**l-1**	**…**	**r-1**	**r**	**r+1**	**…**

Prefix Sum

1 + … + arr[l-1] | arr[l] + … + arr[r] | sum[r]

1	…	l-1	l	l-1	…	r-1	r	r+1	…

Prefix Sum

- [1,1]을 구하는 것을 생각해보자
- 1번째 원소만 더하면 되므로 배열의 1번째의 인덱스를 0으로 사용한다면 arr[0]으로 구할 수 있다
- 하지만 앞선 식을 적용한다면 sum[0] - sum[-1]이므로 배열의 범위를 벗어난다
- 따라서 일반적으로 sum[0]을 0으로 놔두고 인덱스를 1부터 사용한다

Prefix Sum

```
#define MAX_N 100001

int sum[MAX_N];
int n;

for (int i = 1; i <= n; i++) {
    cin >> sum[i];
    sum[i] = sum[i - 1] + sum[i];
}
```

Prefix Sum

- sum배열과 arr배열을 따로 나눌 필요가 없다
- 기존 입력 숫자 하나는 [i, i] 구간의 구간합과 동일하다
- 따라서 구간 합을 구할 수 있다면 기존의 입력을 기억할 필요가 없다
- 따라서 배열 하나로 사용하면 된다

Prefix Sum

- l, r의 구간 합을 물어보는 경우 sum[r] - sum[l - 1]로 구할 수 있다
- 전처리 과정에서 O(N)의 시간 복잡도를 요구하고 쿼리는 뺄셈 하나로 처리가 가능하므로 O(N + Q)의 시간 복잡도를 따른다

Prefix Sum

```
#define MAX_N 100001

int sum[MAX_N];
int n, q;

cin >> n >> q;
for (int i = 1; i <= n; i++) {
    cin >> sum[i];
    sum[i] = sum[i - 1] + sum[i];
}

int l, r;
while (q--) {
    cin >> l >> r;
    cout << sum[r] - sum[l - 1] << '\n';
}
```

Prefix Sum

- 구간 합 뿐만 아니라 구간의 복구가 가능하다면 다른 방식으로도 접근이 가능하다
- 구간 곱 또는 구간 XOR 등을 같은 방식으로 구현할 수 있다

Prefix Sum

- 구간 곱의 경우, 첫번째 원소부터 i번째 원소까지 모두 곱한 값을 기억하고 있는다
- [l, r]구간에 해당하는 구간 곱을 알고 싶은 경우 r까지 곱한 값에서 l-1까지 곱한 값을 나누면 된다
- 앞선 값에 계속해서 곱하므로 0번째 배열의 초기값은 1이어야 한다

Prefix Sum

```
#define MAX_N 100001

int sum[MAX_N];
int n, q;

sum[0] = 1;
cin >> n >> q;
for (int i = 1; i <= n; i++) {
    cin >> sum[i];
    sum[i] = sum[i - 1] * sum[i];
}

int l, r;
while (q--) {
    cin >> l >> r;
    cout << sum[r] / sum[l - 1] << '\n';
}
```

Prefix Sum

- XOR의 경우 1번 XOR한 경우 본인과 값이 동일하며, 2번 XOR한 경우 0으로 돌아간다
- 이것을 이용하여 곱셈에서 나눗셈을 해 원하는 구간의 값을 없애는 것처럼 다시 한번 XOR해 원하는 구간의 값을 없앨 수 있다
- [l, r]구간에 해당하는 구간 XOR을 알고 싶은 경우 r까지 XOR한 값에서 l-1까지 XOR한 값을 XOR하면 된다
- 0번째 배열의 초기 값은 0이다

Prefix Sum

```
#define MAX_N 100001

int sum[MAX_N];
int n, q;

cin >> n >> q;
for (int i = 1; i <= n; i++) {
    cin >> sum[i];
    sum[i] = sum[i - 1] ^ sum[i];
}

int l, r;
while (q--) {
    cin >> l >> r;
    cout << sum[r] ^ sum[l - 1] << '\n';
}
```

Prefix Sum

- 누적 합을 사용할 때는 자료형에 주의해야한다
- 마지막 값의 경우 1번째부터 N번째까지 모든 값을 더하거나 곱한 값을 저장하고 있다
- 즉, 값이 매우 커지므로 사용하려는 범위 안에 포함되는지 생각하고 변수를 선언해야한다

Prefix Sum

- 원소가 100,000개이며 각 원소는 최대 100,000의 값을 가진다면 최대 누적 합은 10,000,000,000이다
- int의 범위가 2,147,483,647이므로 int의 범위를 넘어간다
- 다음과 같은 경우 long long으로 sum배열을 만들어야 한다

문제

- 구간 합 구하기 4 BOJ 11659
- 구간 합 구하기 5 BOJ 11660
- 낚시 BOJ 30461

Segment Tree

Prefix Sum with Update

- 구간 합을 빠르게 구하기 위해 누적 합에 대해 배웠다
- 다음과 같이 2가지를 수행해야 하는 알고리즘을 생각해보자

- [l, r]구간에 해당하는 구간합을 구하여라
- 몇 번째 숫자를 value로 바꾸어라

Prefix Sum with Update

- 숫자 값을 바꿔야하는 경우 기존에 구해둔 누적 합을 다시 구해야한다
- 만약 첫번째 숫자가 바뀐 경우 모든 배열의 값을 다시 구해야 한다
- 숫자 값이 바뀌는 연산이 자주 들어오는 경우를 위한 알고리즘을 생각해봐야 한다
- ex) 구간 합 구하기 BOJ 2042

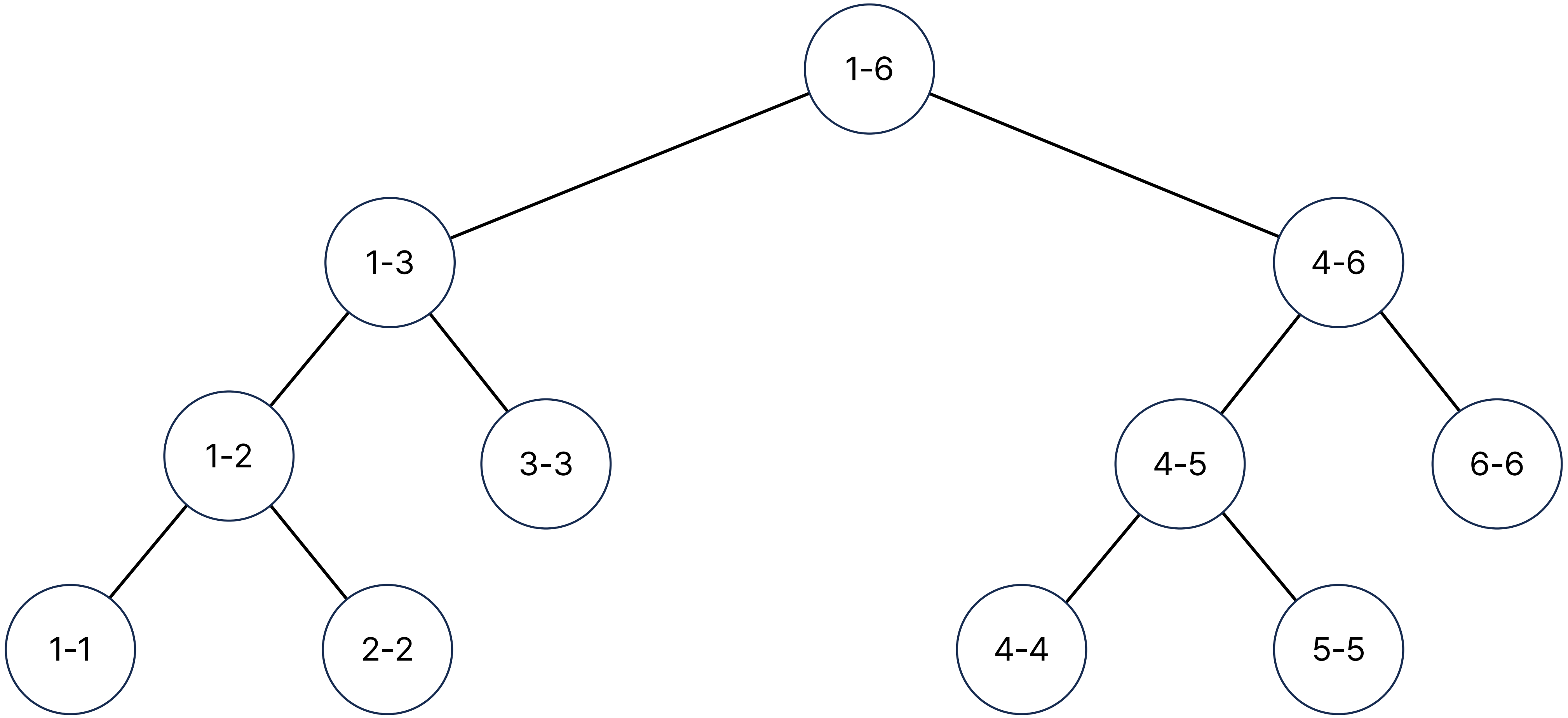
Prefix Sum with Update

- 앞에서부터 전체를 더하는 것이 아닌 몇 개의 블럭으로 나누는 방법을 생각할 수 있다
- 어떠한 값이 바뀐다면 해당 값이 포함된 블럭만 바꾸는 경우, 어느 정도 타협점을 생각할 수 있다
- 대표적인 방법이 제곱근 분할법, 세그먼트 트리이다

Segment Tree

- 세그먼트 트리는 구간이 1인 노드를 N개, 2인 노드를 N/2개, 4인 노드를 N/4개, … ,
  N인 노드를 1개 가지는 형태이다
- 세그먼트 트리는 이진 트리 형태이다
- 루트는 1부터 N까지의 구간을 관리한다
- 자식으로 내려갈 때 부모의 구간을 절반씩 관리한다

Segment Tree



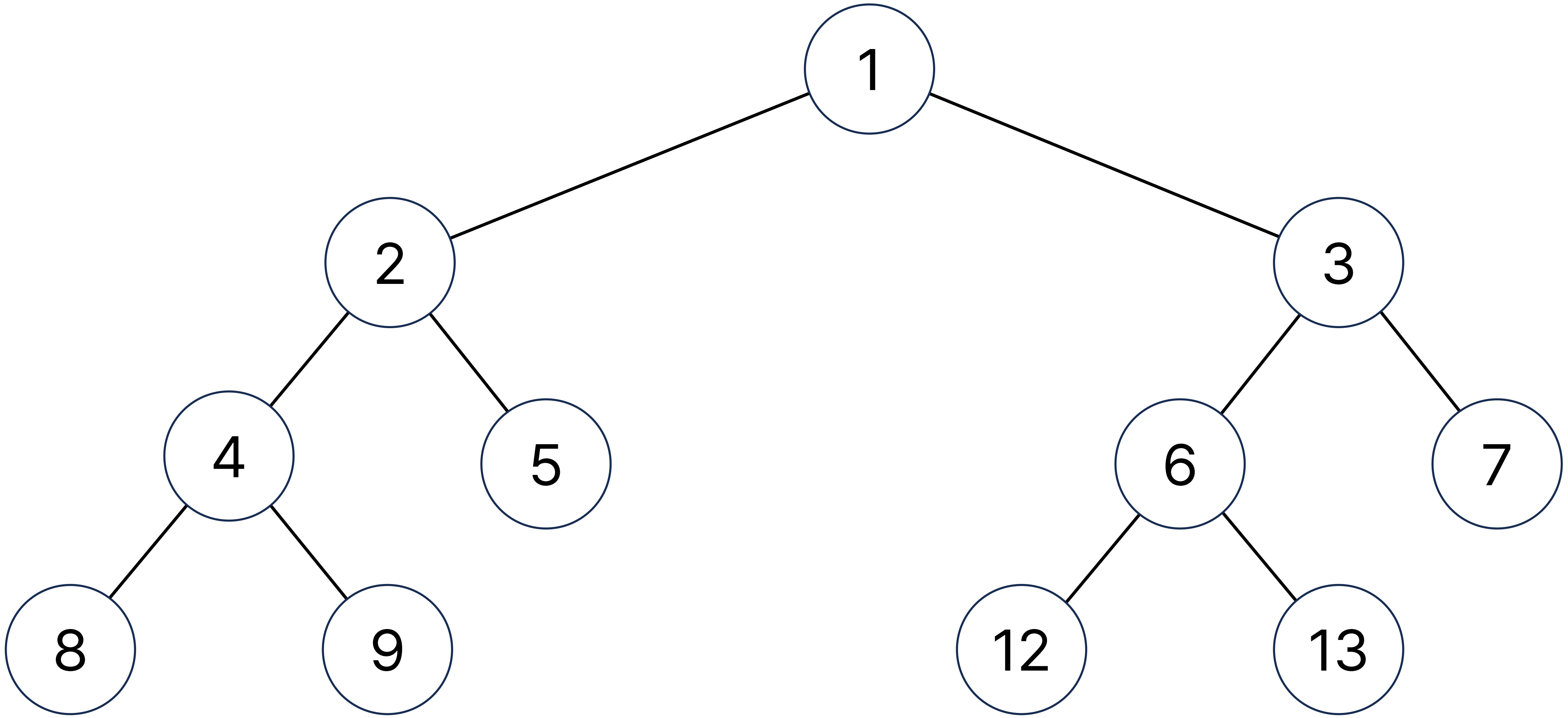
Segment Tree

- 각 깊이별로 관리하는 구간의 길이가 절반이 된다
- 따라서 트리의 전체 깊이는 $\log_2 N$을 따른다
- 누적 합의 경우 업데이트 시 최악의 경우 N개의 값을 모두 업데이트 해야하므로 O(N)의 시간복잡도를 따른다
- 세그먼트 트리는 하나의 노드가 포함된 노드가 $\log_2 N$개 이므로 O(log N)의 시간복잡도에 업데이트가 가능하다

Segment Tree Init

- 재귀로 세그먼트 트리를 구현하는 경우 일반적으로 노드의 4배를 할당하면 된다
  10만 개의 원소 -> 40만 개의 노드
- 인덱스의 경우 힙과 동일한 방식으로 설정한다
- 루트 노드의 인덱스 1, 왼쪽 자식은 부모 노드 인덱스의 2배, 오른쪽 자식은 부모 노드 인덱스의 2배 + 1로 설정한다

Segment Index



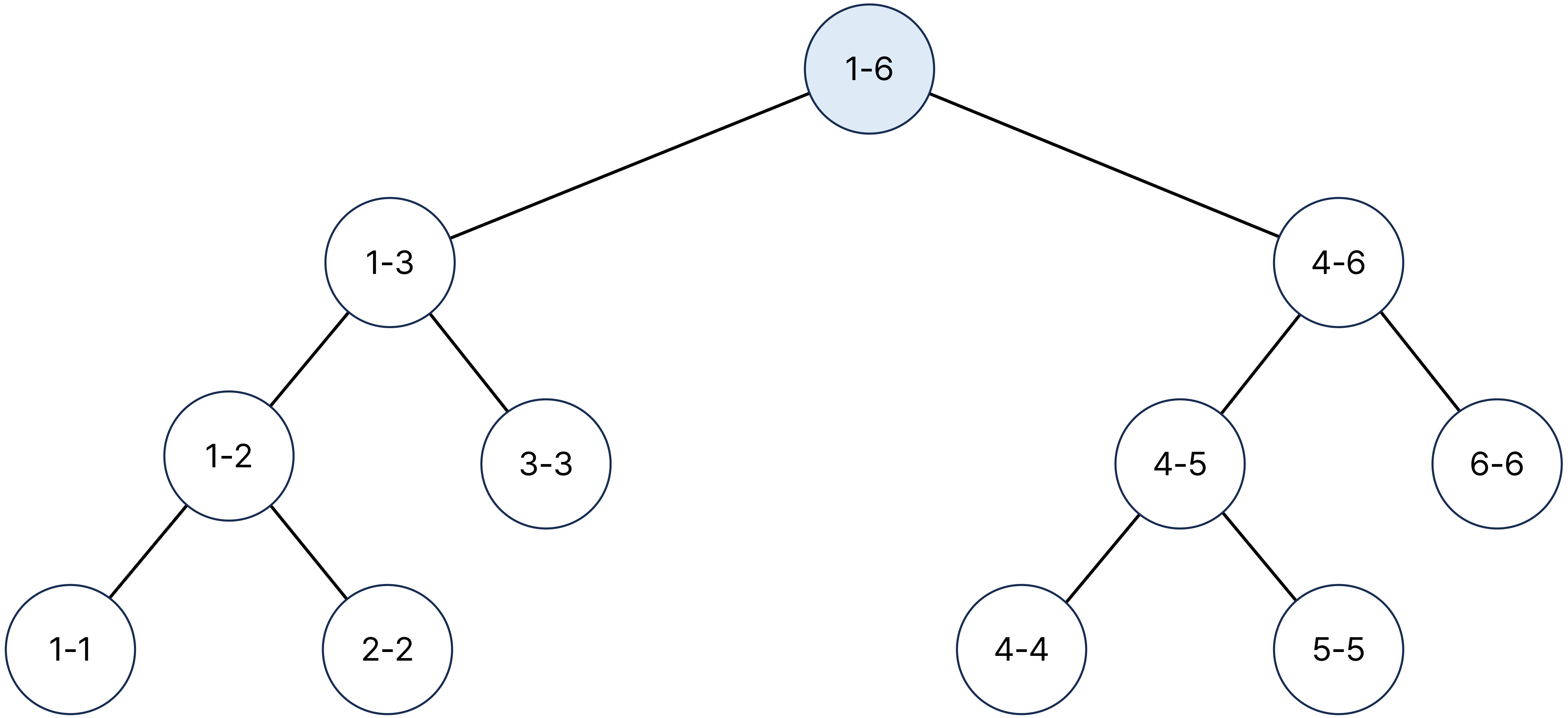
Segment Tree Init

```
int tree[MAX_NODE * 4];
int num[MAX_NODE];

int n;
cin >> n;
for (int i = 1; i <= n; i++)
    cin >> num[i];

tree_init(1, n, 1);
```

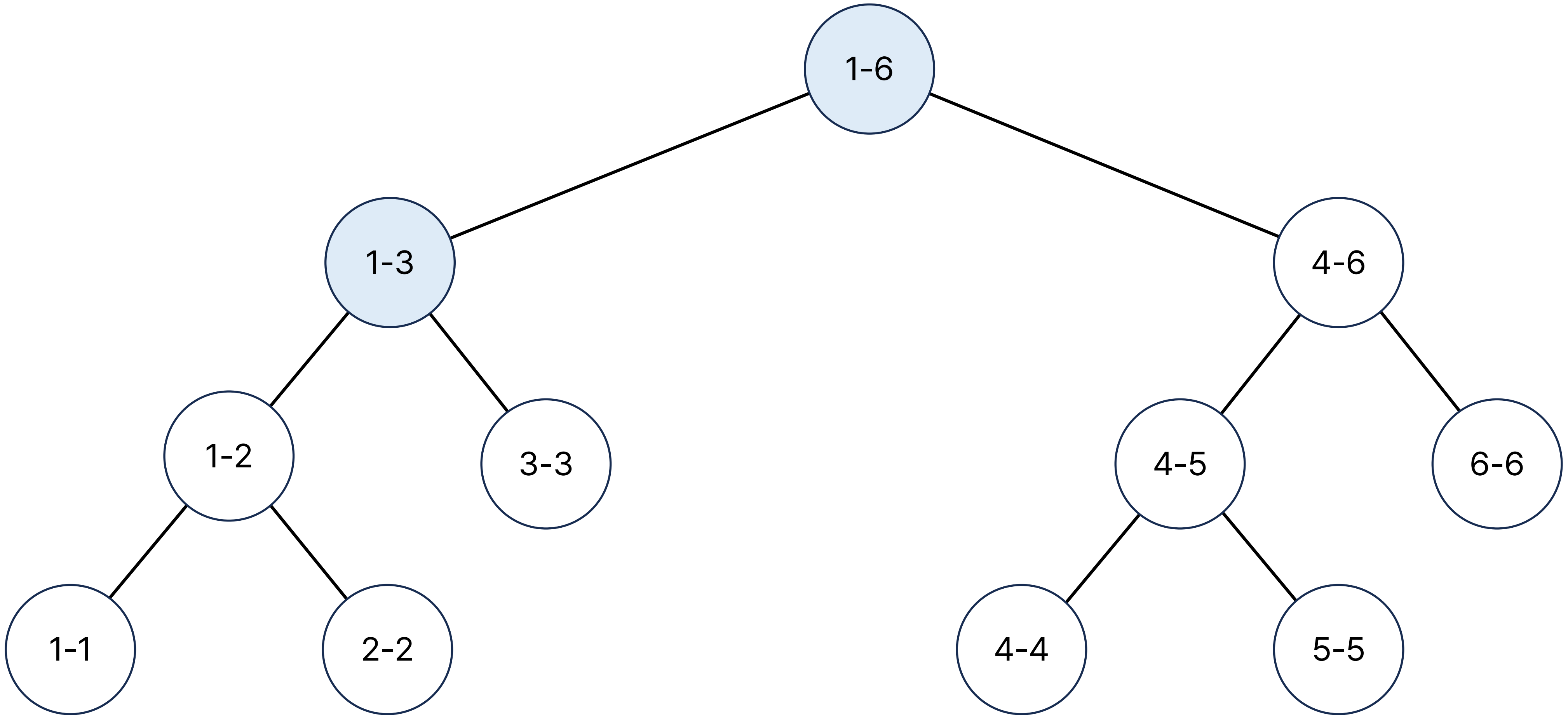
Segment Tree Init

```
void tree_init(int start, int end, int index) {
    if (start == end) {
        tree[index] = num[start];
        return;
    }

    tree_init(start, (start + end) / 2, index * 2);
    tree_init((start + end) / 2 + 1, end, index * 2 + 1);
    tree[index] = tree[index * 2] + tree[index * 2 + 1];
}
```

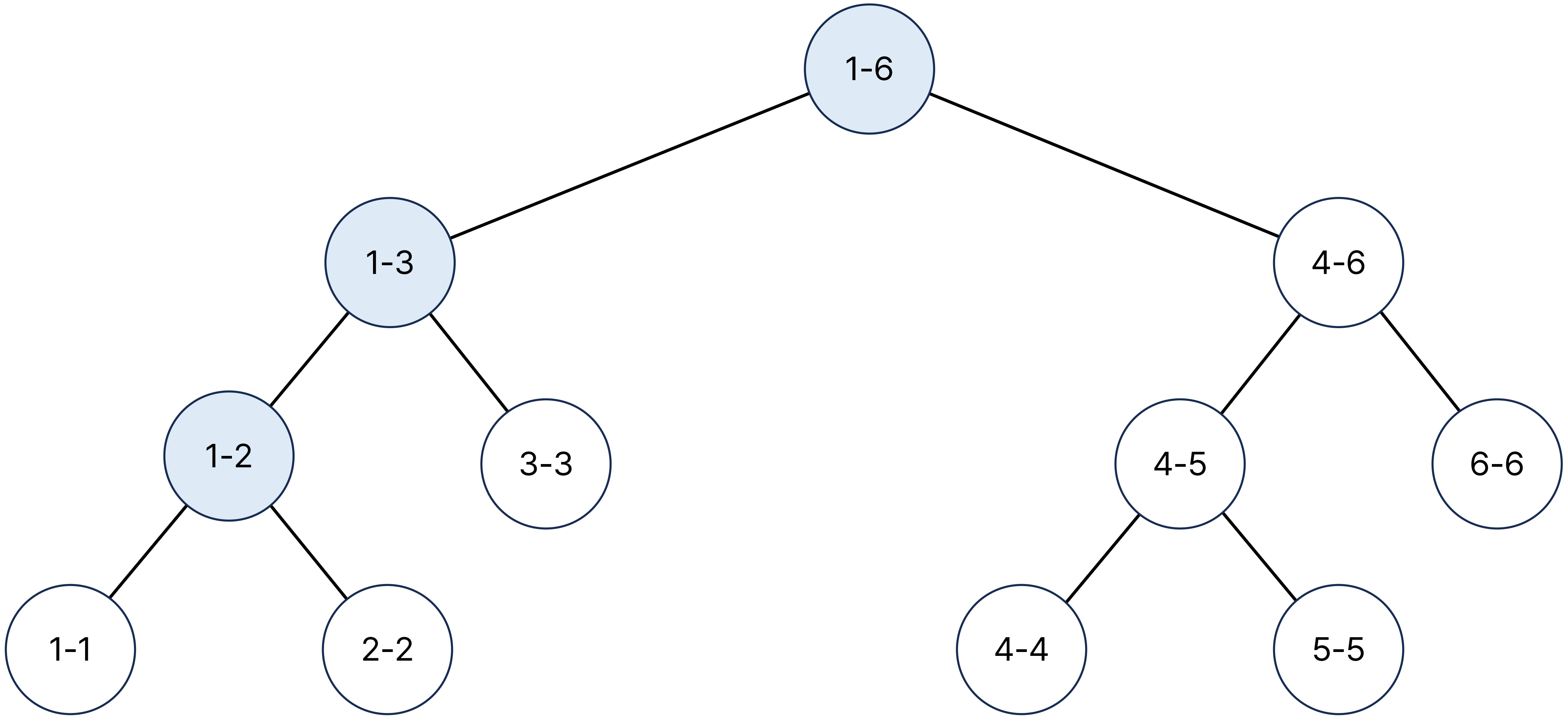
Segment Tree Init



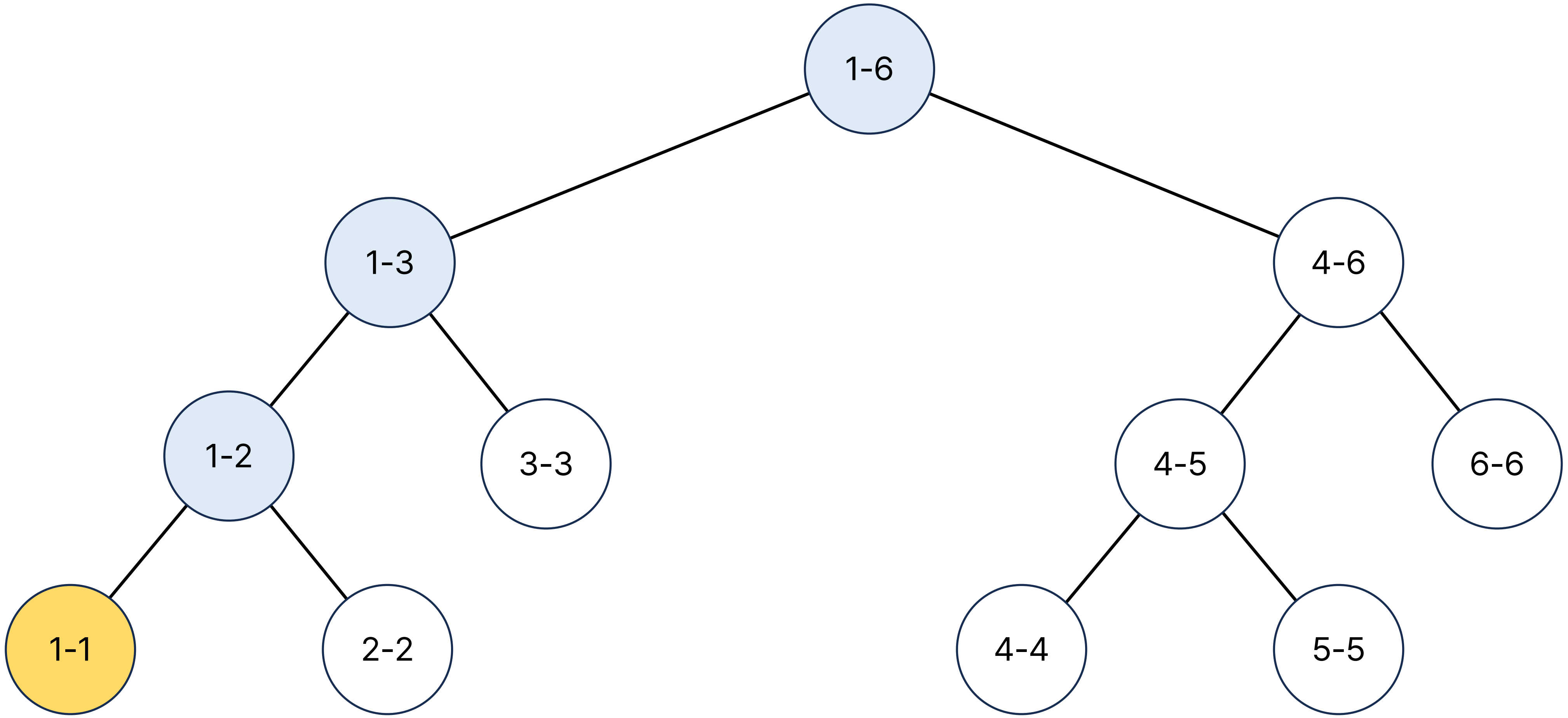
Segment Tree Init



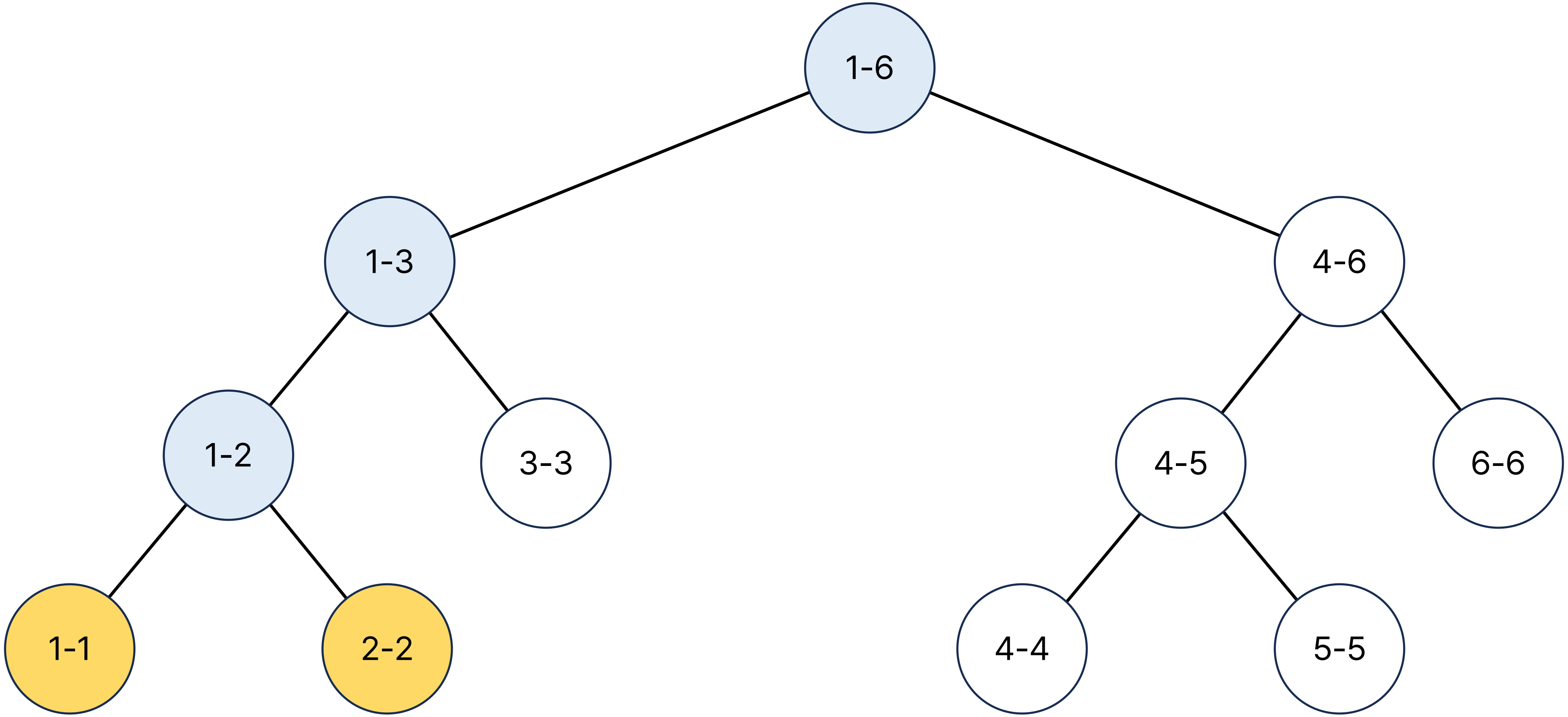
Segment Tree Init



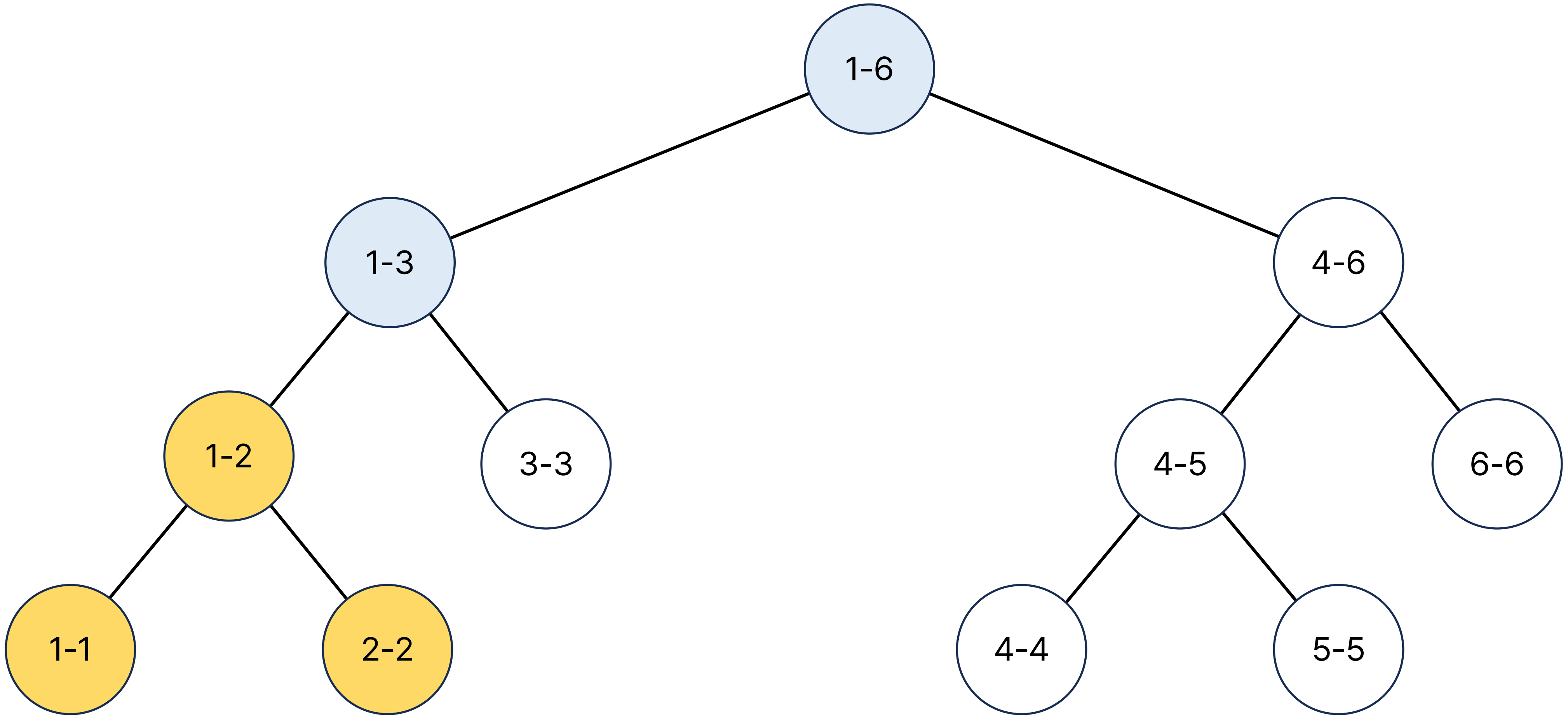
Segment Tree Init



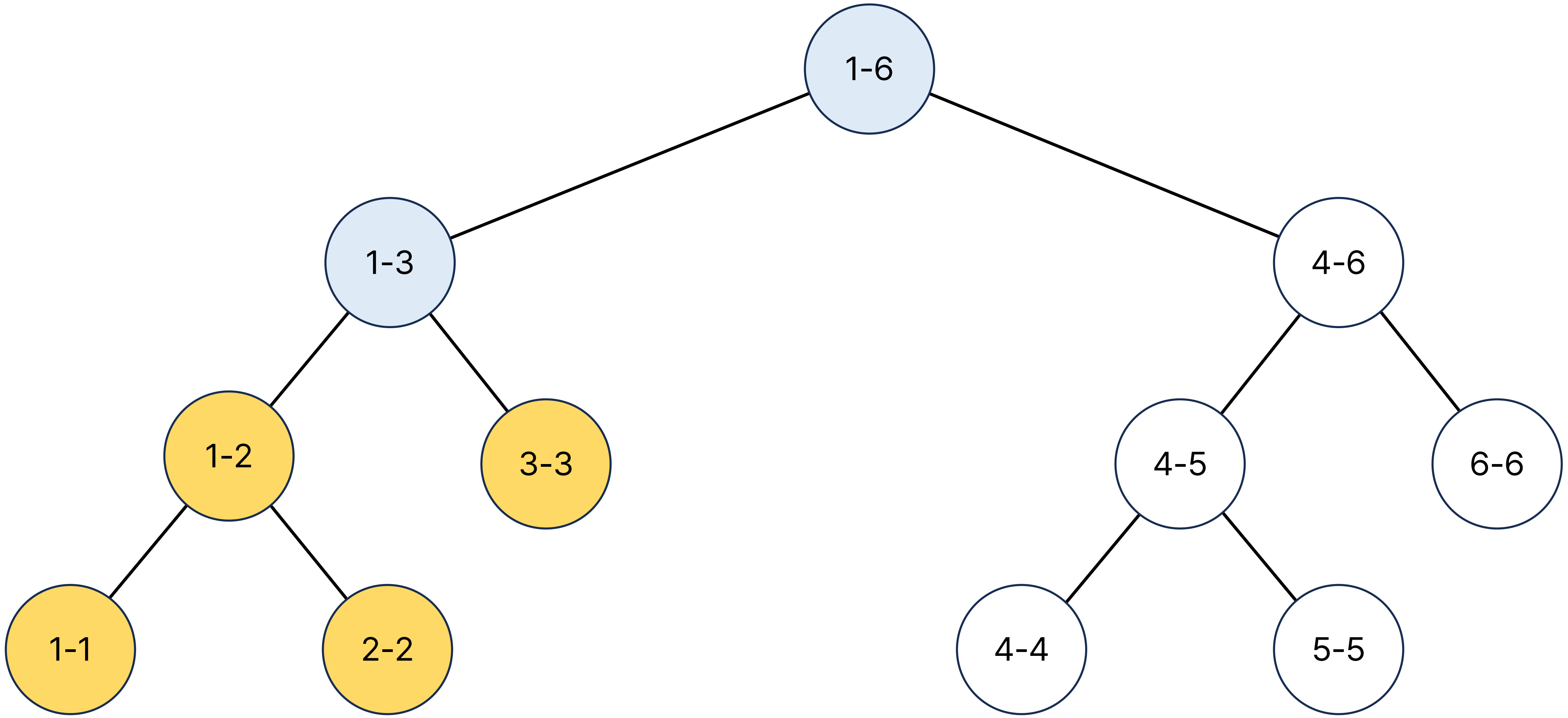
Segment Tree Init



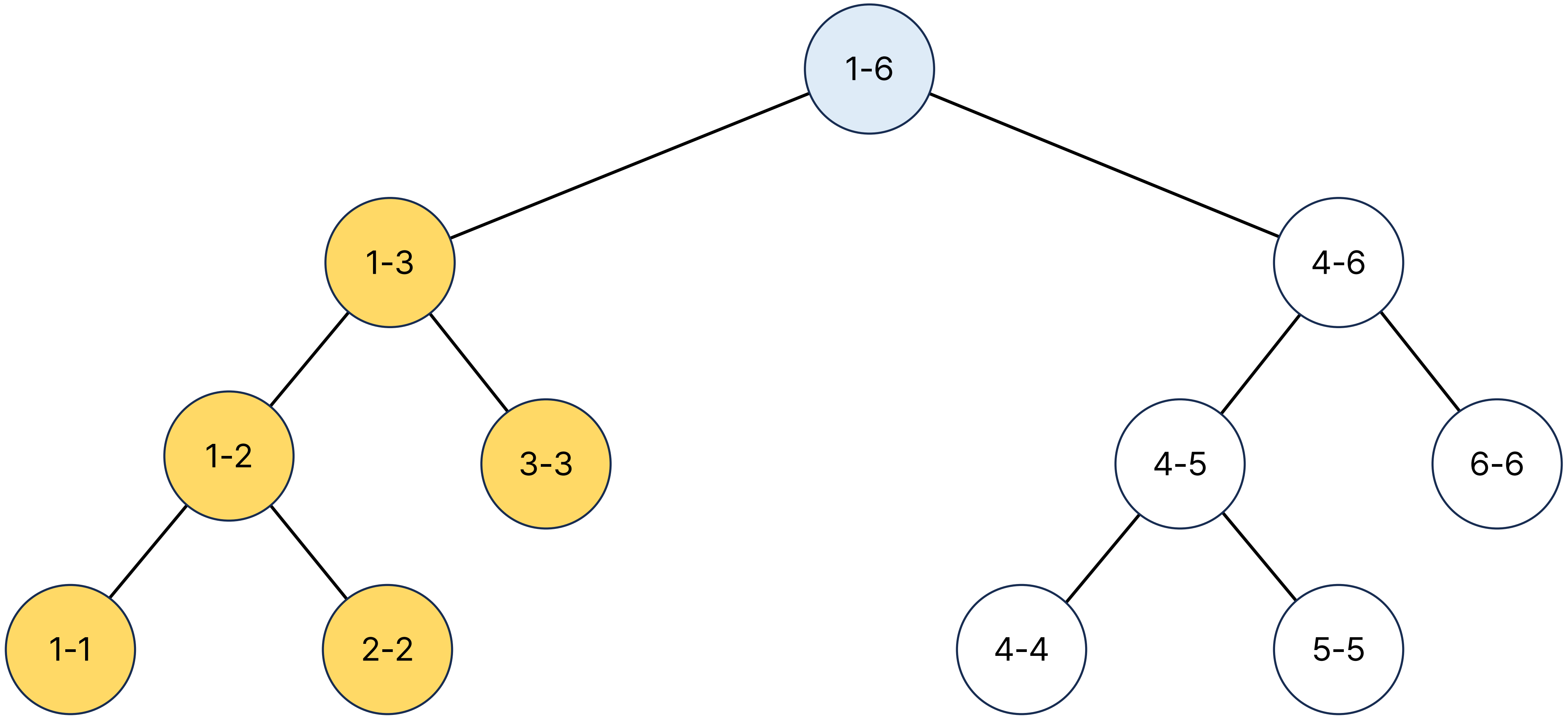
Segment Tree Init



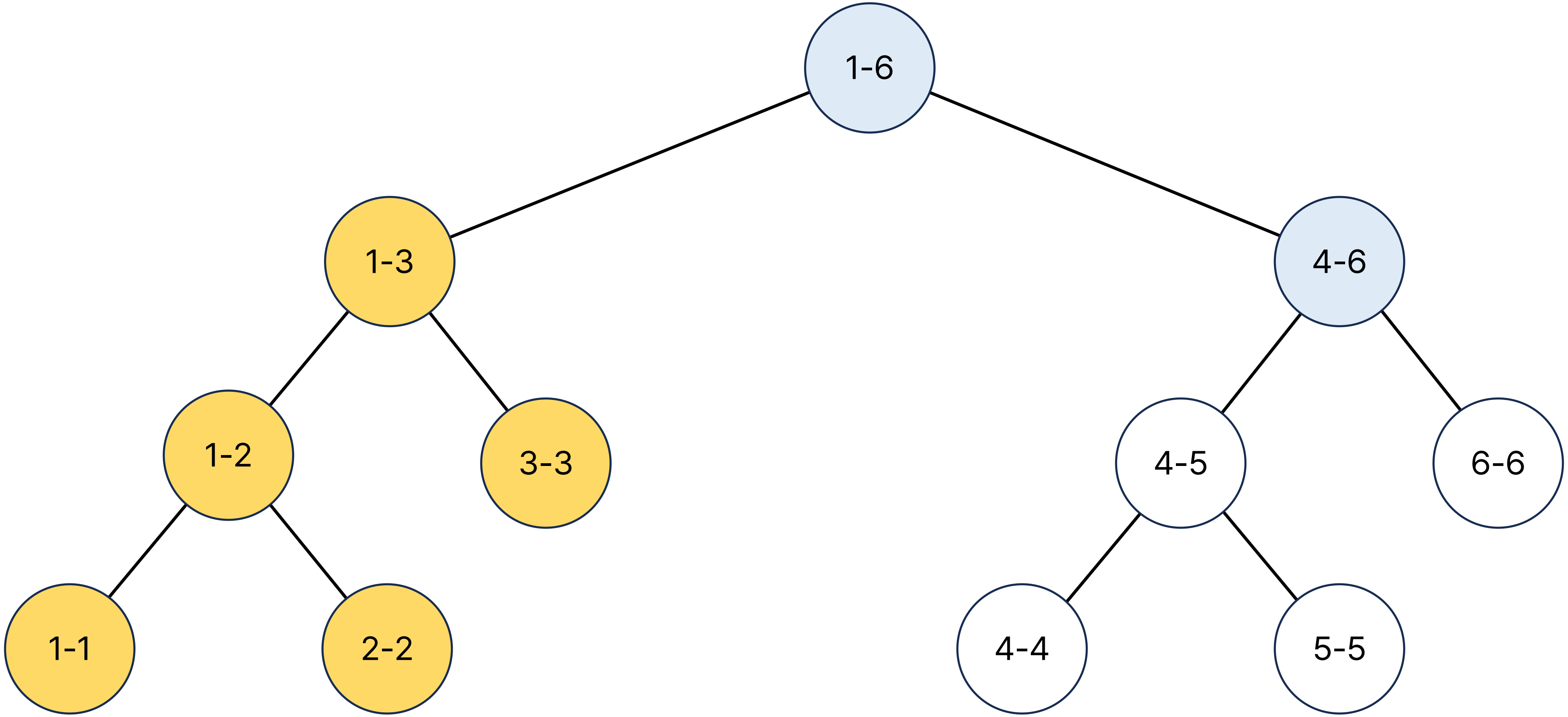
Segment Tree Init



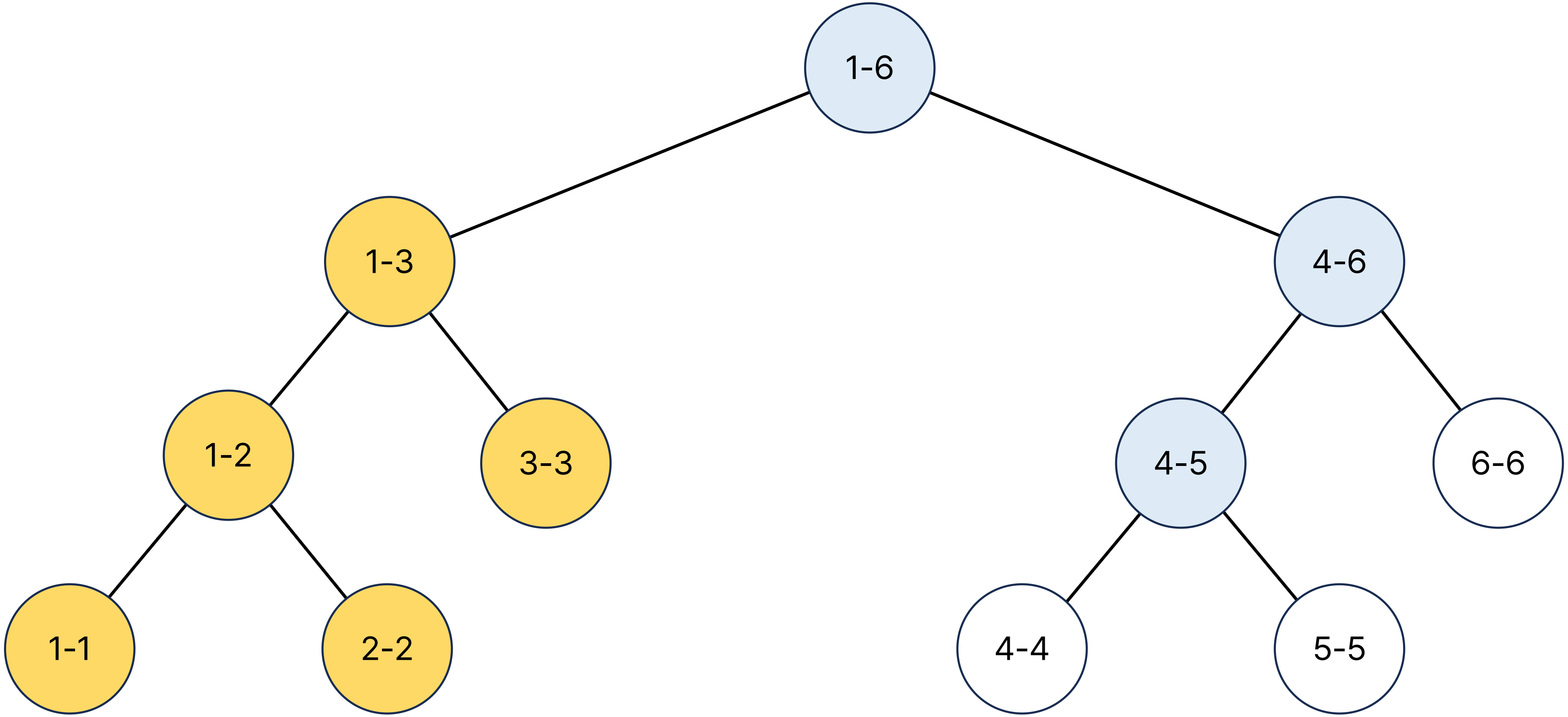
Segment Tree Init



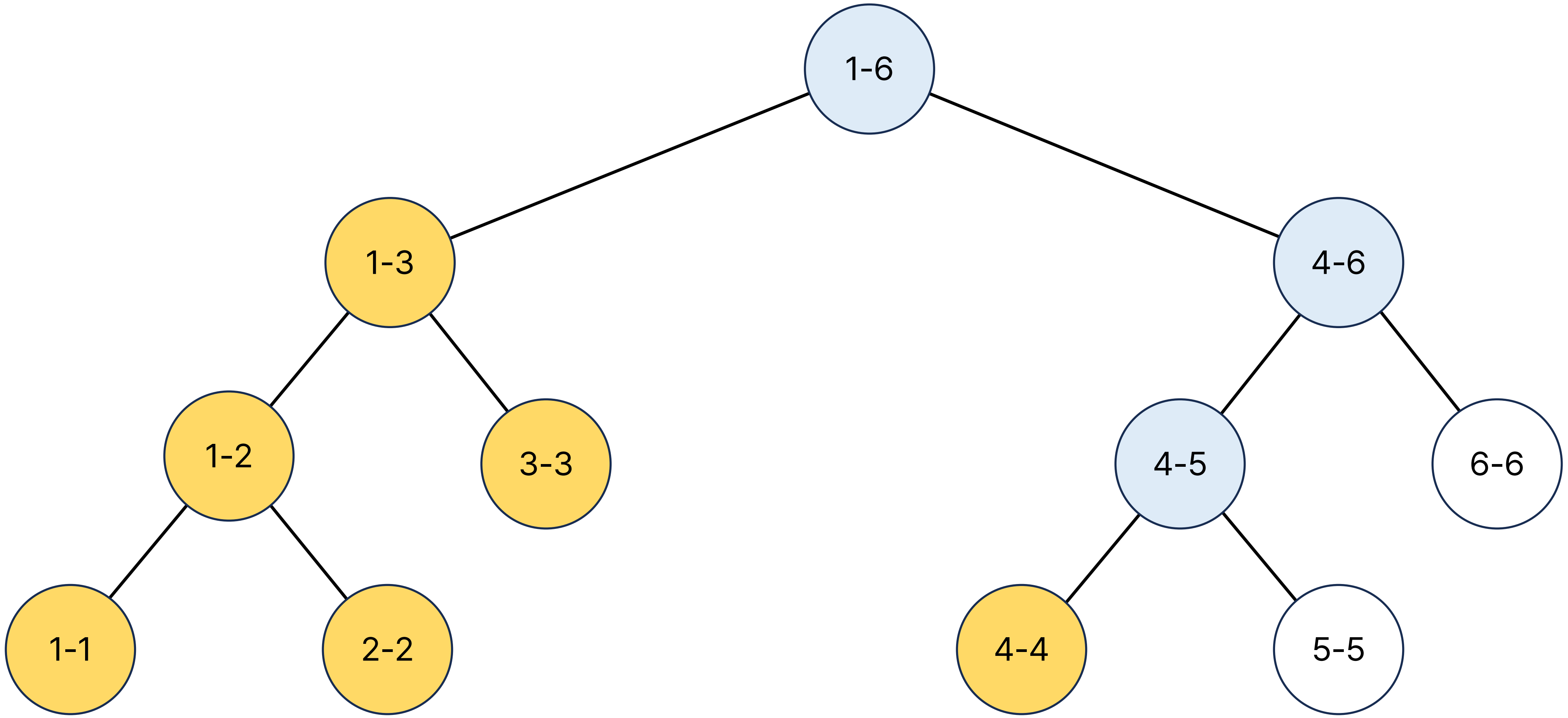
Segment Tree Init



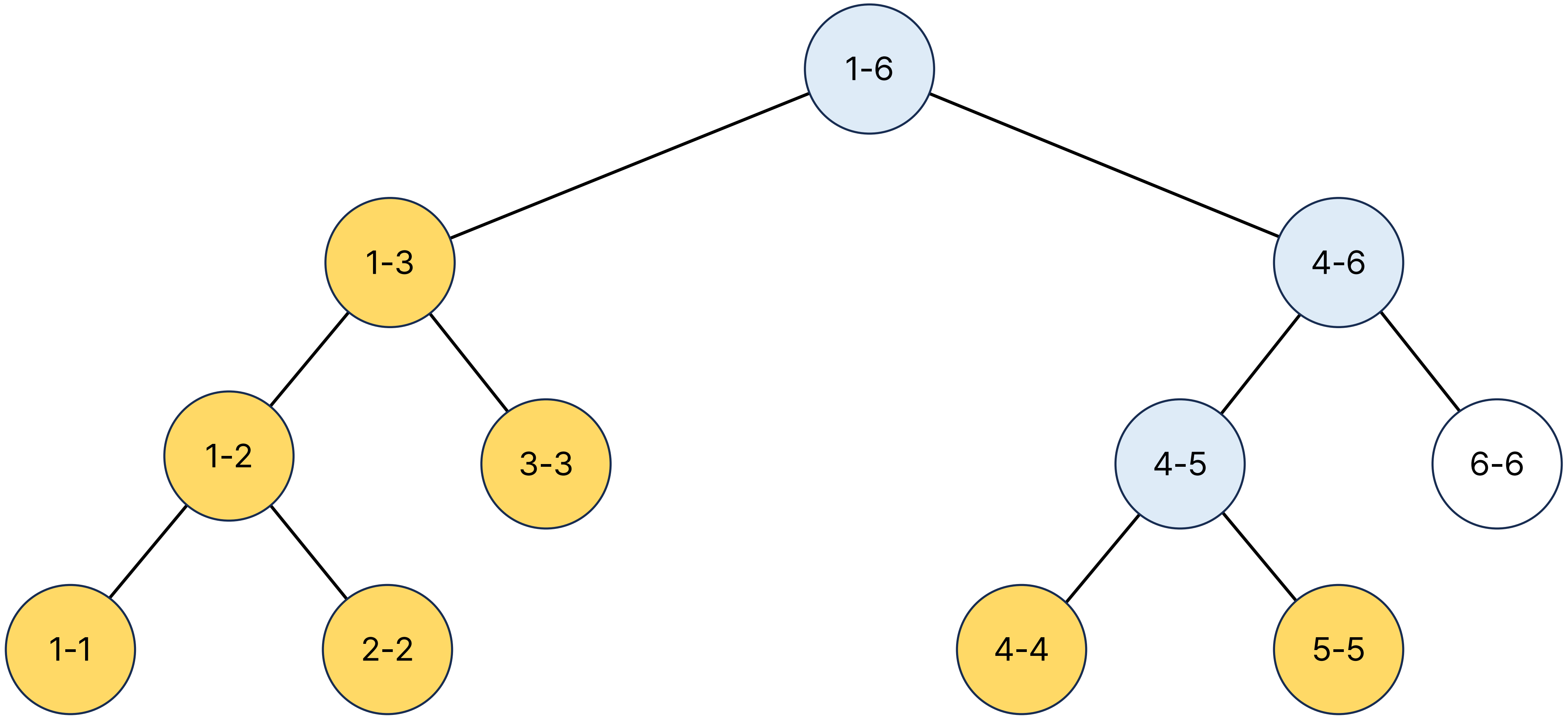
Segment Tree Init



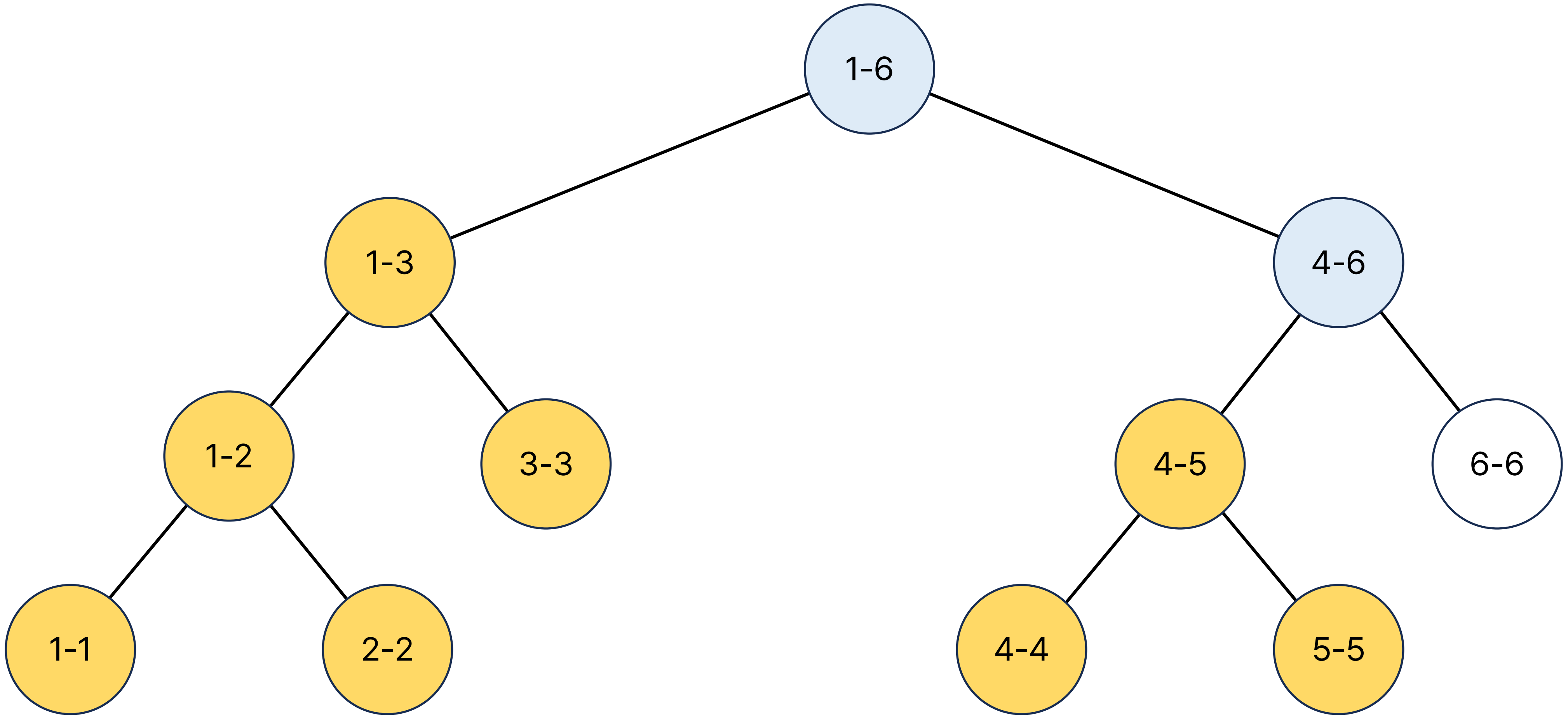
Segment Tree Init



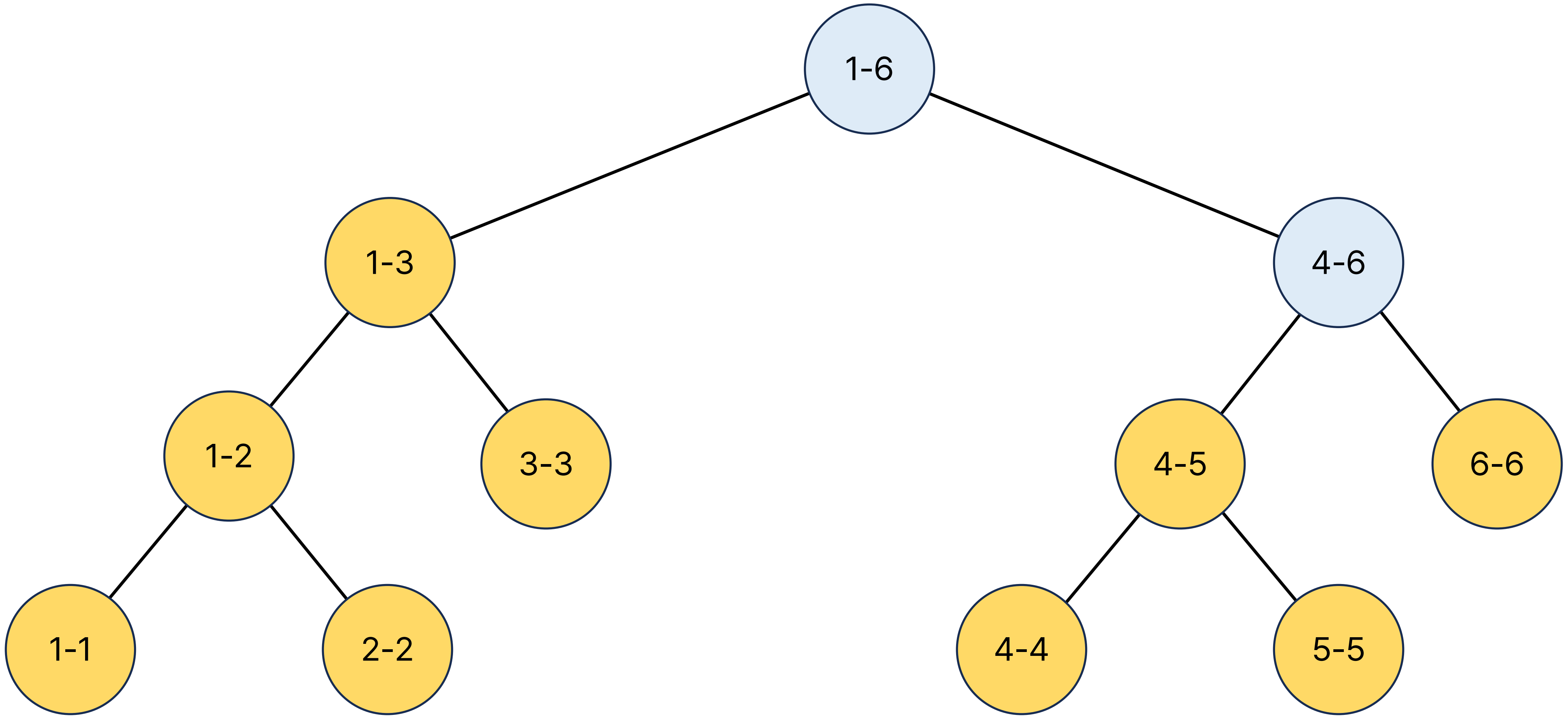
Segment Tree Init



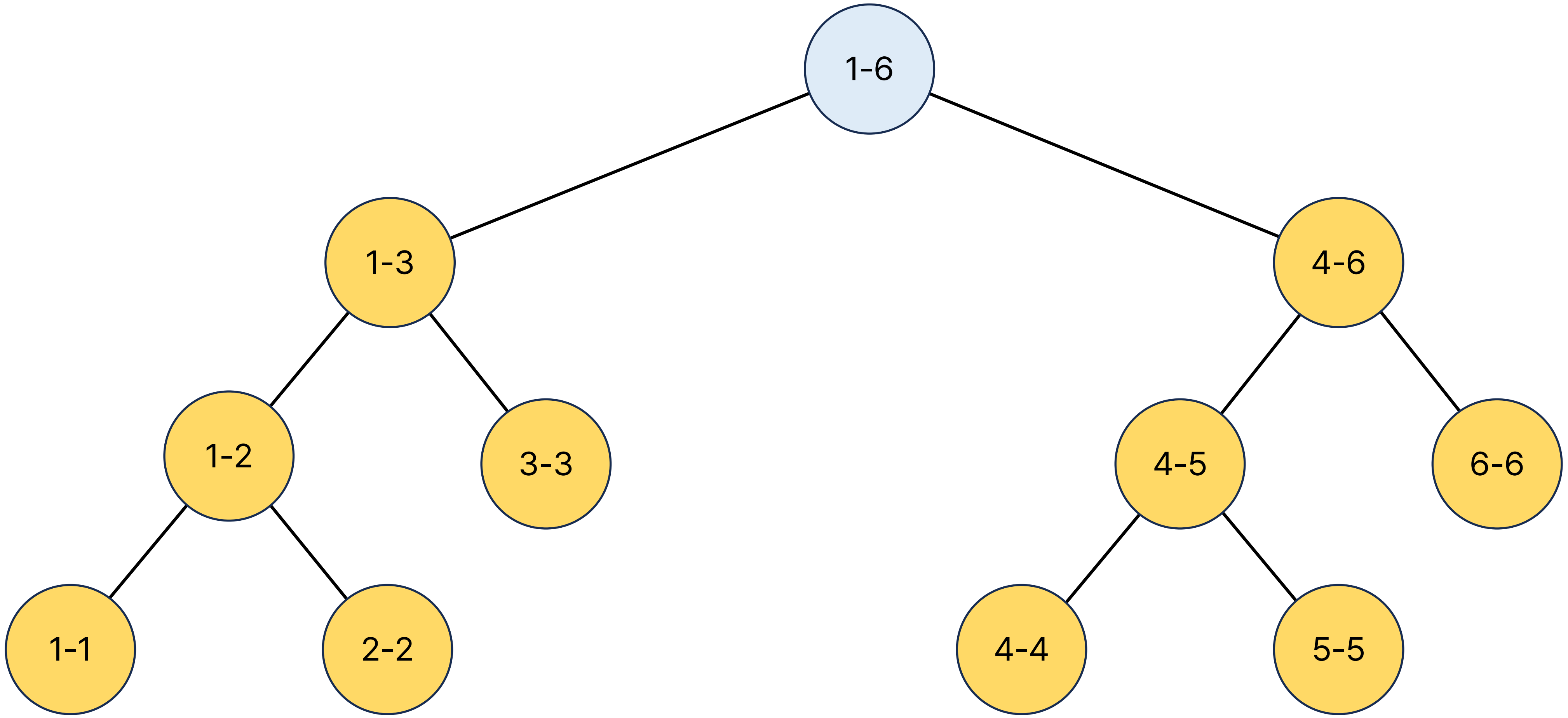
Segment Tree Init



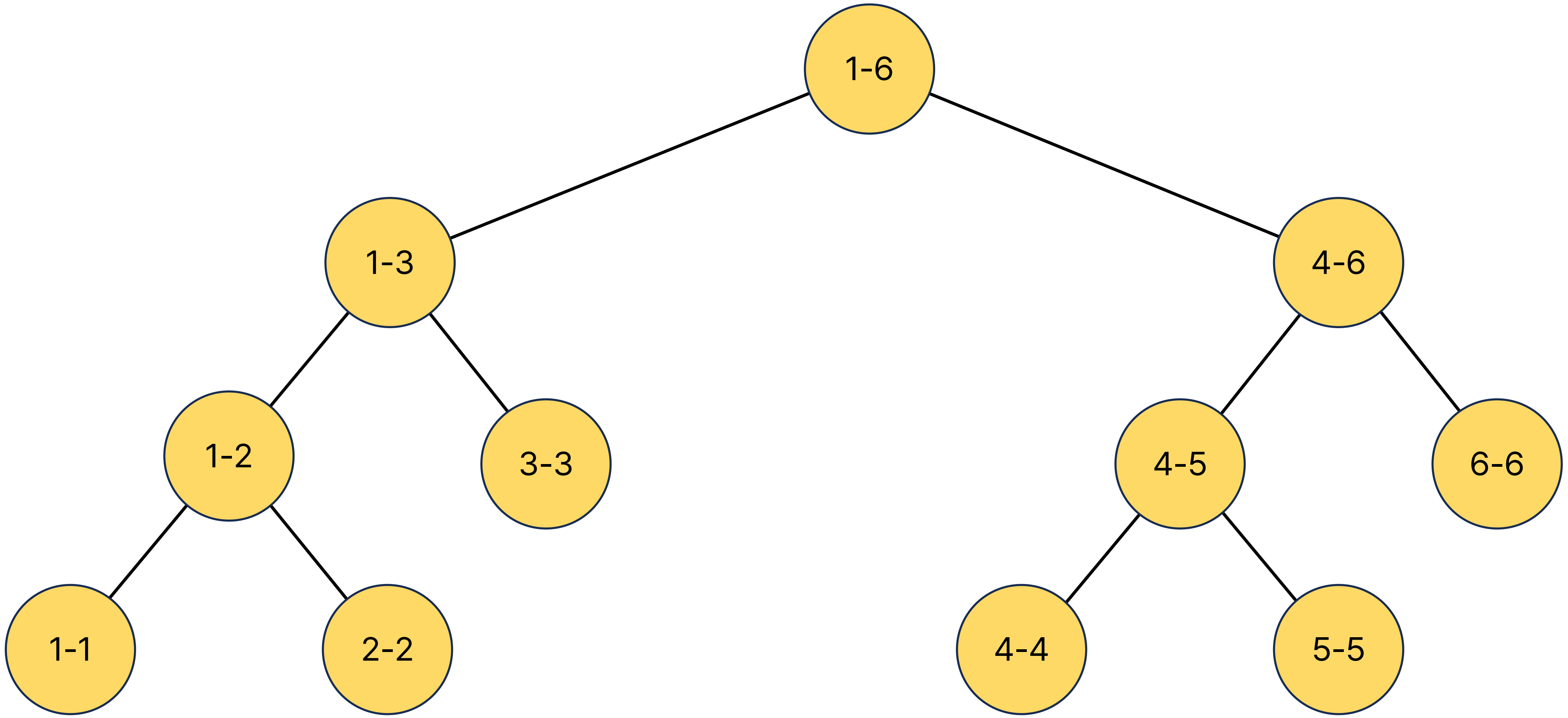
Segment Tree Init



Segment Tree Init



Segment Tree Init



Segment Tree Query

- 특정 구간 합을 구하기 위하여 어떤 노드들을 더해야 하는지 구별해야 한다
- [left, right] 구간 합을 구하라 했을 때 노드의 범위 [start, end]가 구간에 완전히 포함되는 경우 left <= start <= end <= right 인 경우, 해당 노드가 결과에 포함된다
- 반대로 완전히 포함되지 않는 경우 end < left 또는 right < start인 경우 해당 노드는 결과에 포함되지 않는다
- 구간이 걸쳐 있는 경우 자식 노드에 포함되는 노드와 포함되지 않는 노드가 공존한다. 이 때 자식 노드로 내려가서 다시 판단한다

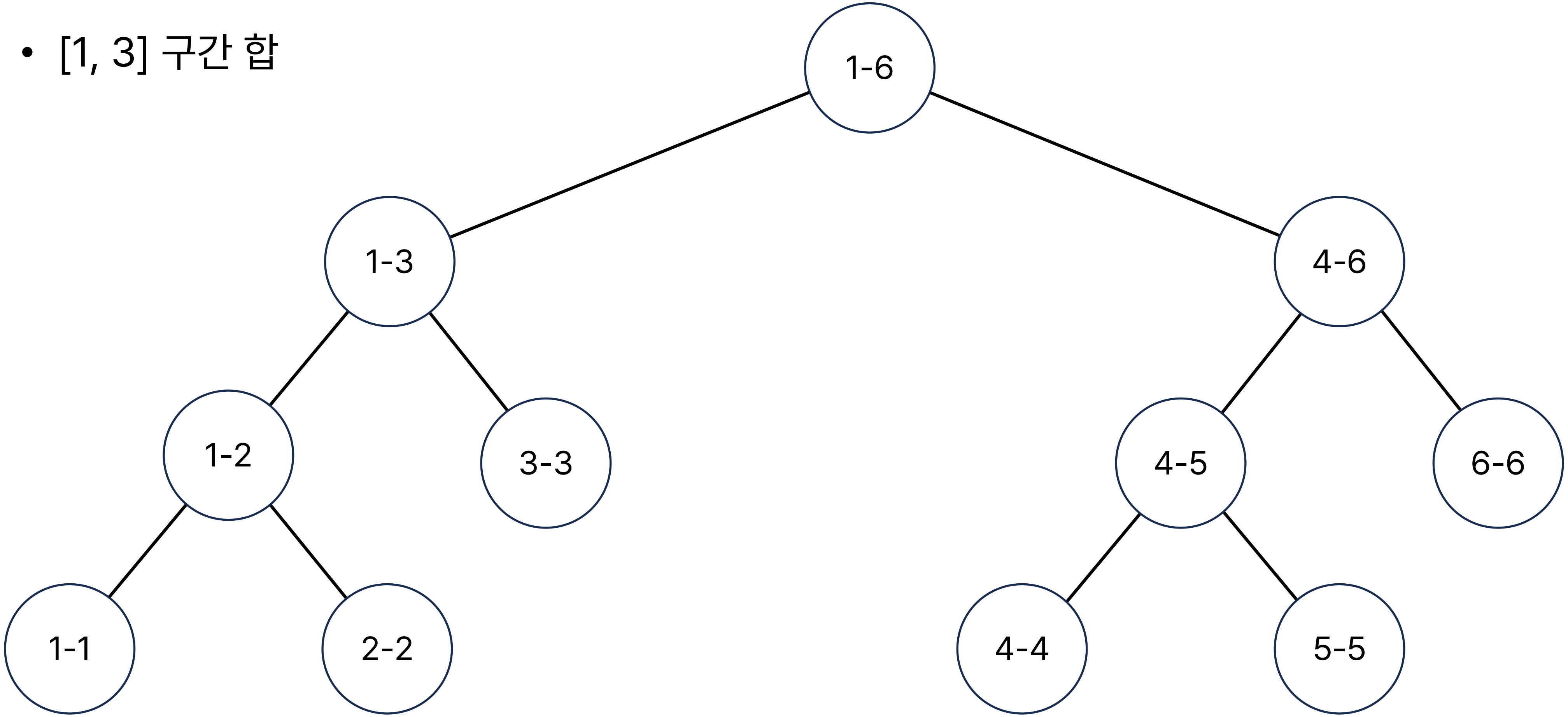
Segment Tree Query

- 구간이 완전히 포함되지 않은 경우, 0을 리턴한다
- 구간이 완전히 포함되는 경우, 해당 노드가 가진 구간 합을 리턴한다
- 이 두가지 경우가 아닌 경우 구간이 겹쳐 있는 경우이다

  이런 경우 자식 노드들에게 다시 쿼리를 보내 돌아오는 값을 합쳐서 리턴한다

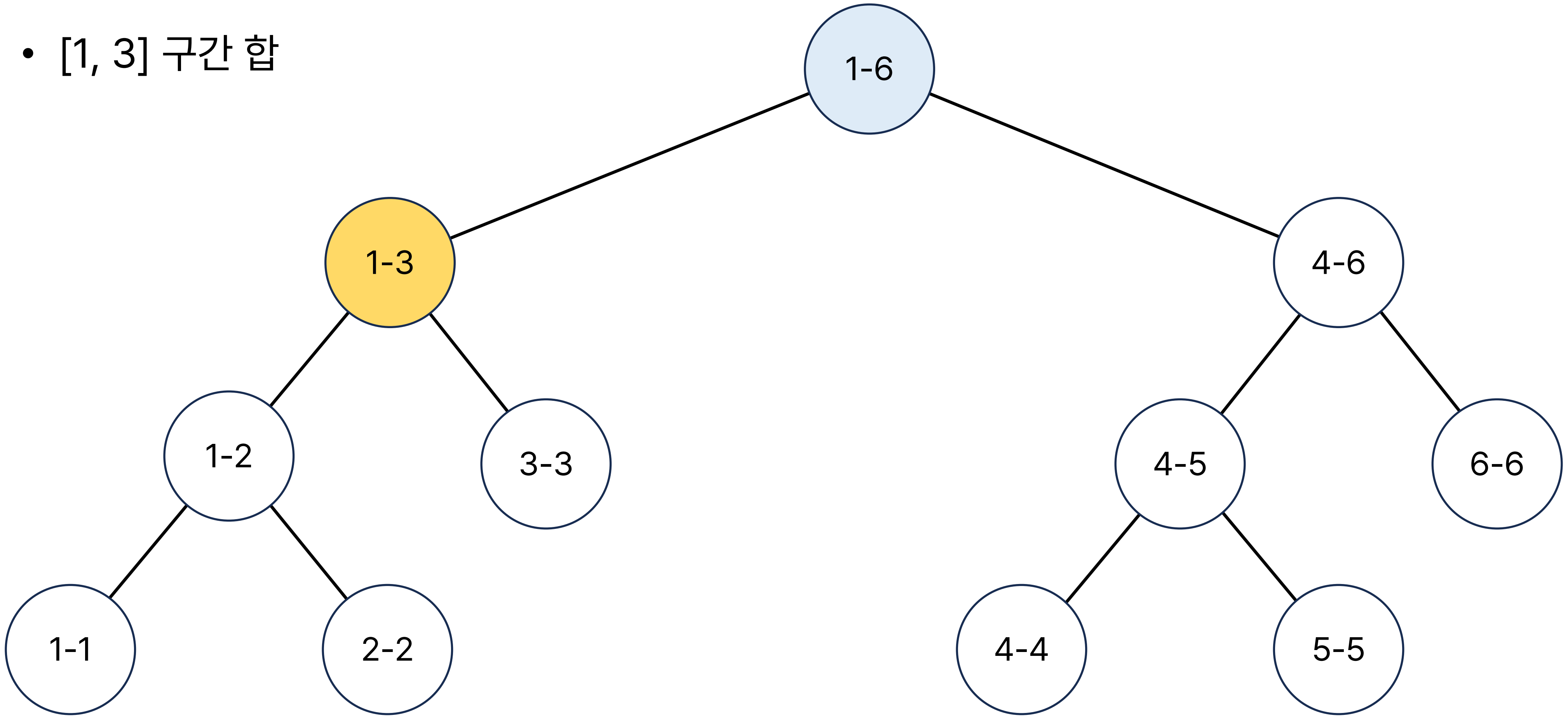
Segment Tree Query

- [1, 3] 구간 합



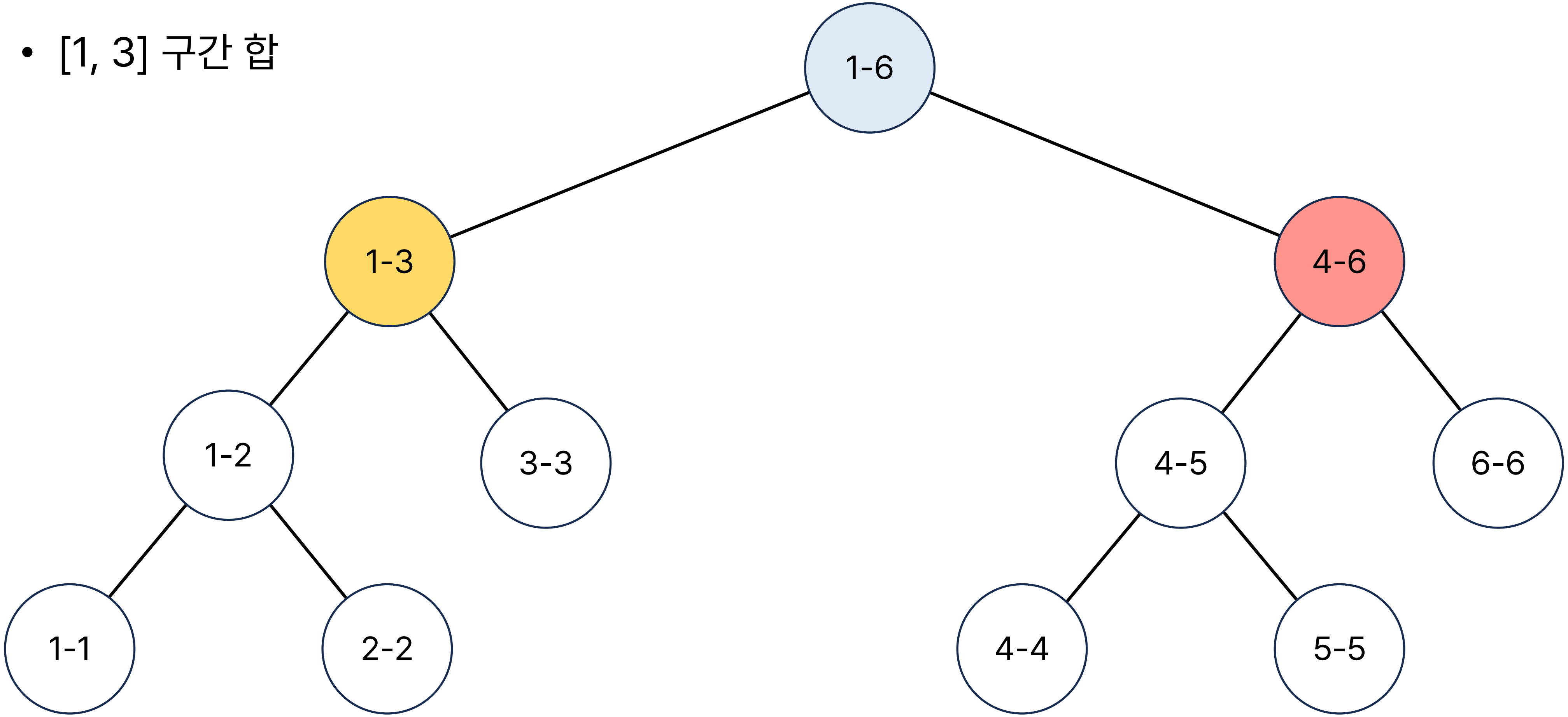
Segment Tree Query

- [1, 3] 구간 합



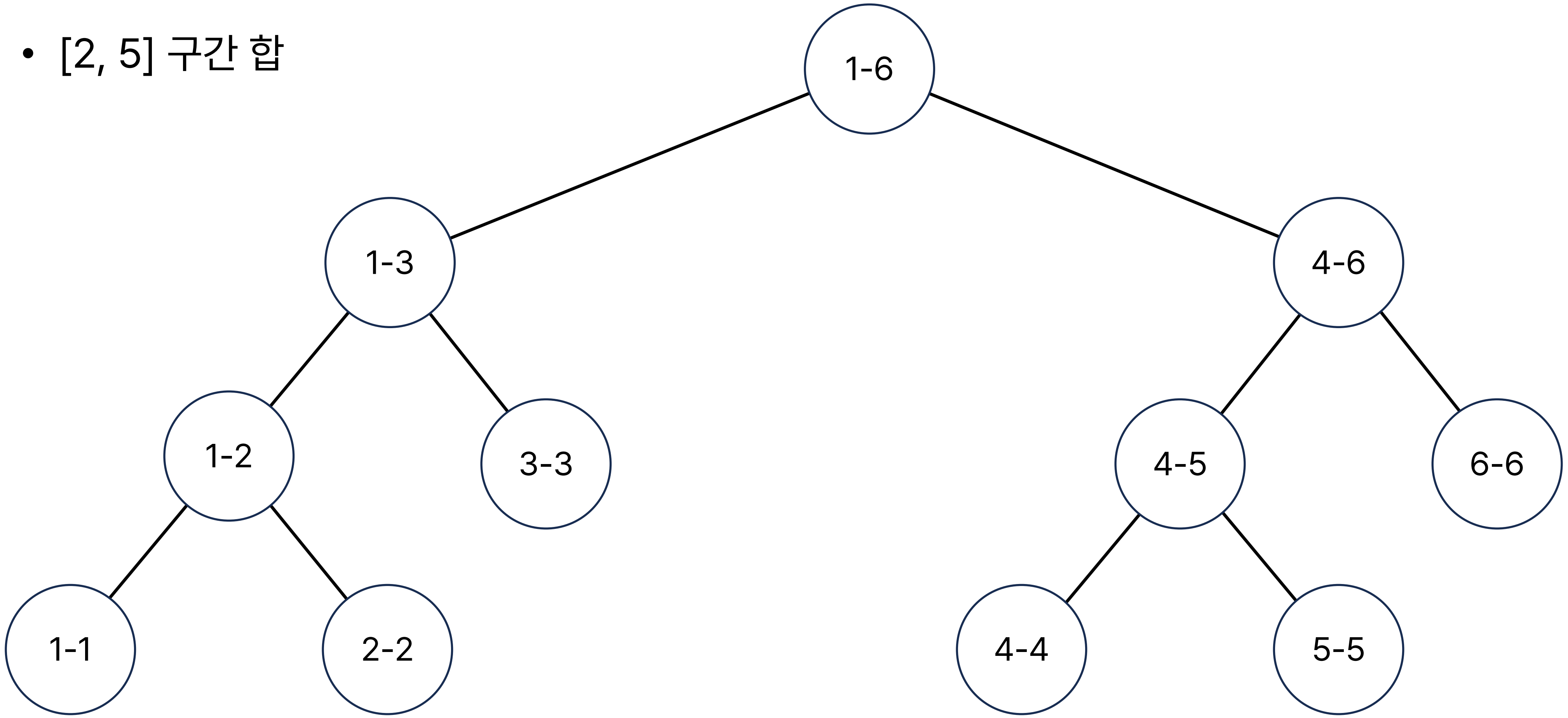
Segment Tree Query

- [1, 3] 구간 합



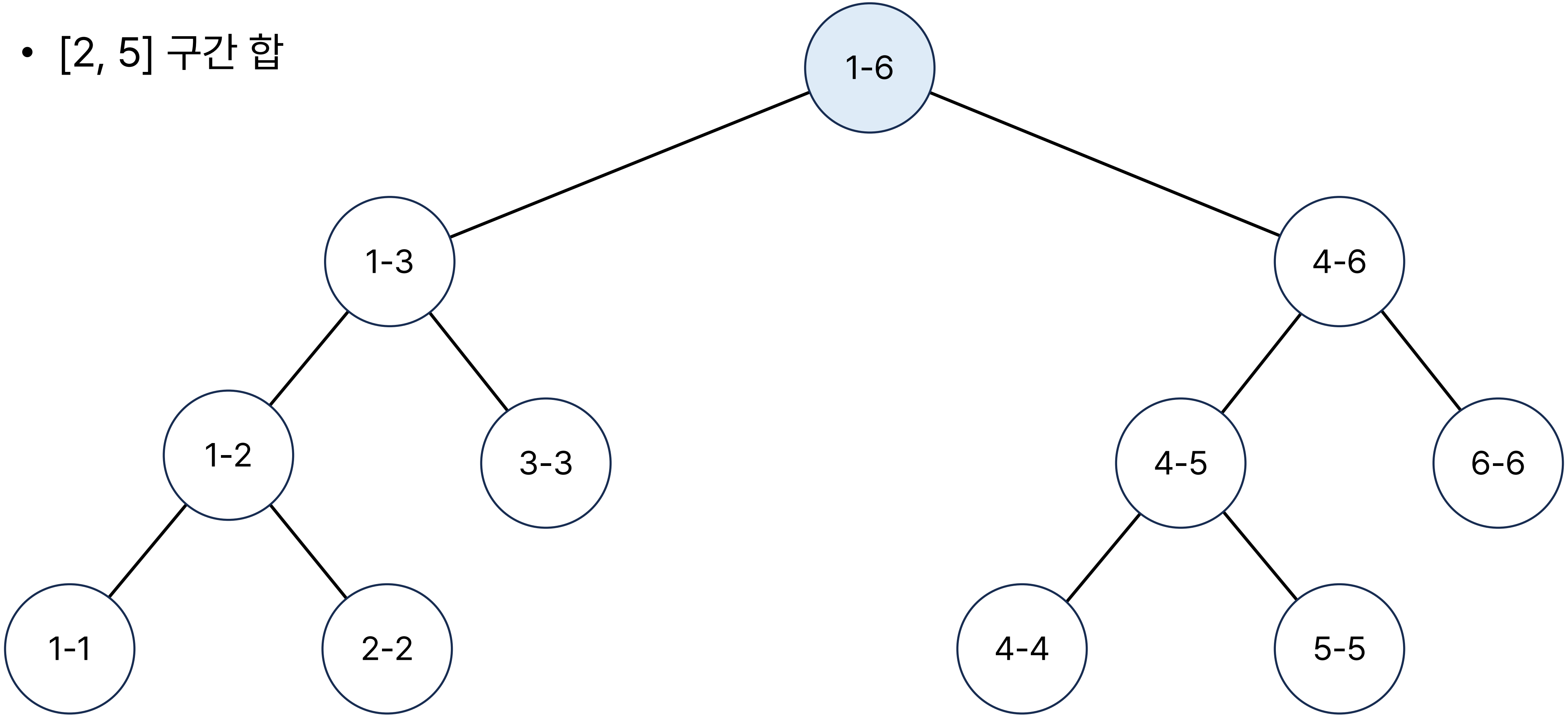
Segment Tree Query

- [2, 5] 구간 합



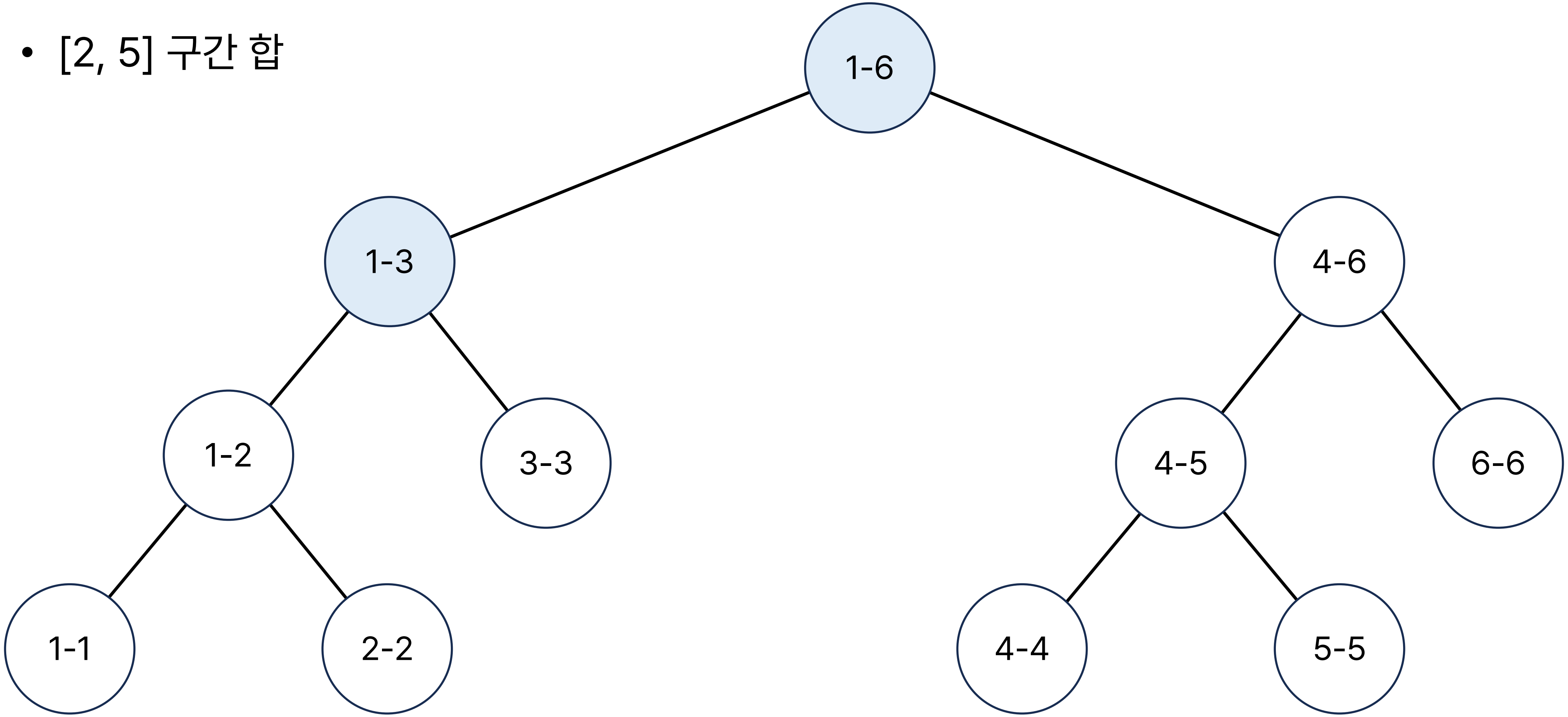
Segment Tree Query

- [2, 5] 구간 합



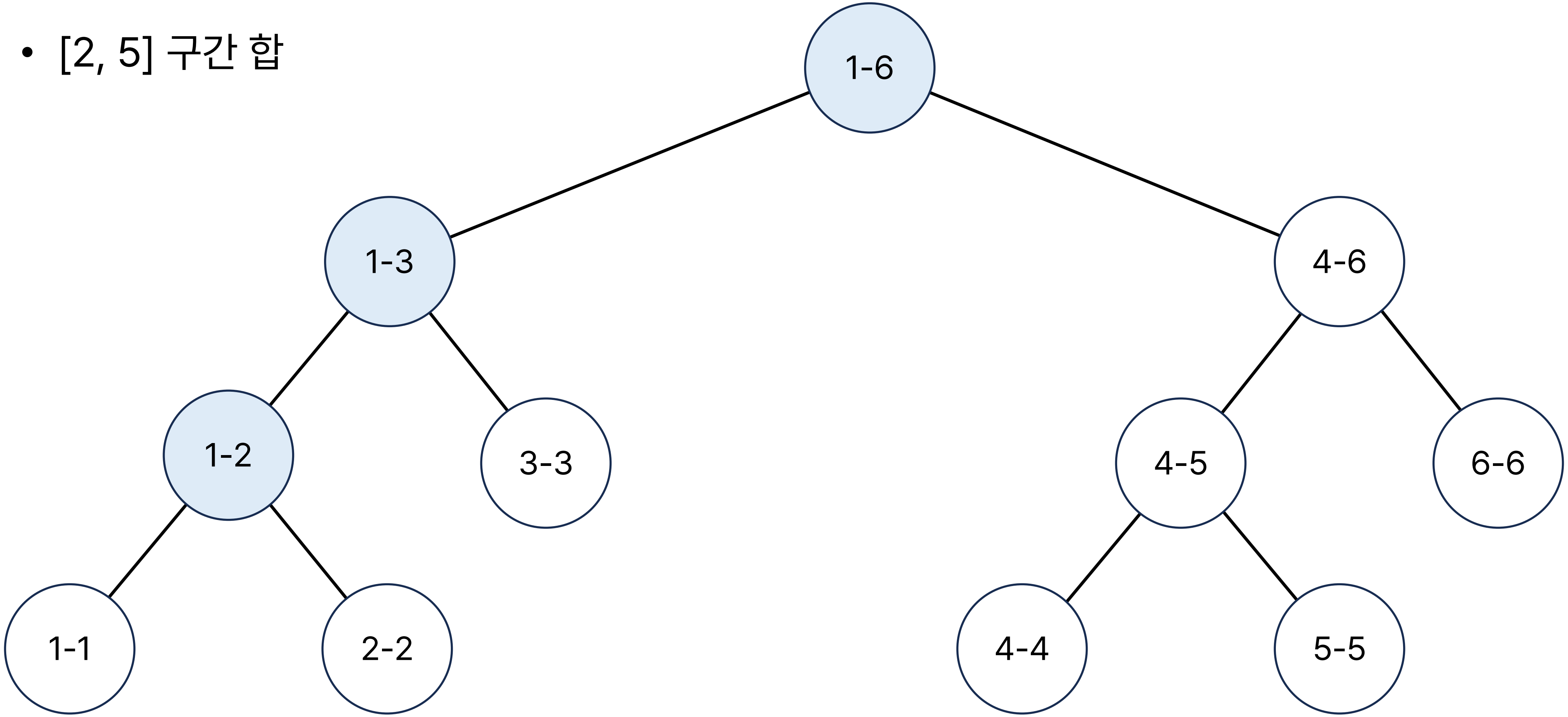
Segment Tree Query

- [2, 5] 구간 합



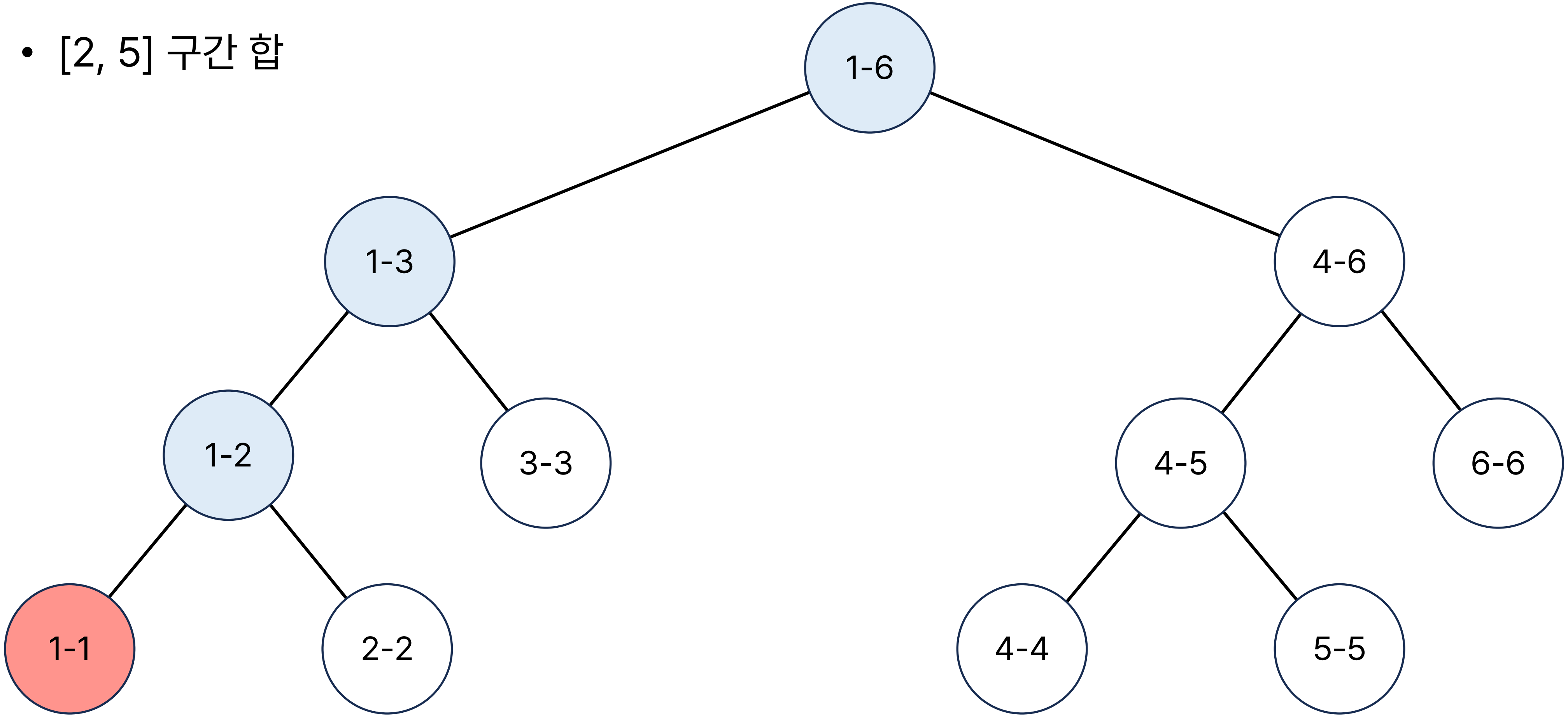
Segment Tree Query

- [2, 5] 구간 합



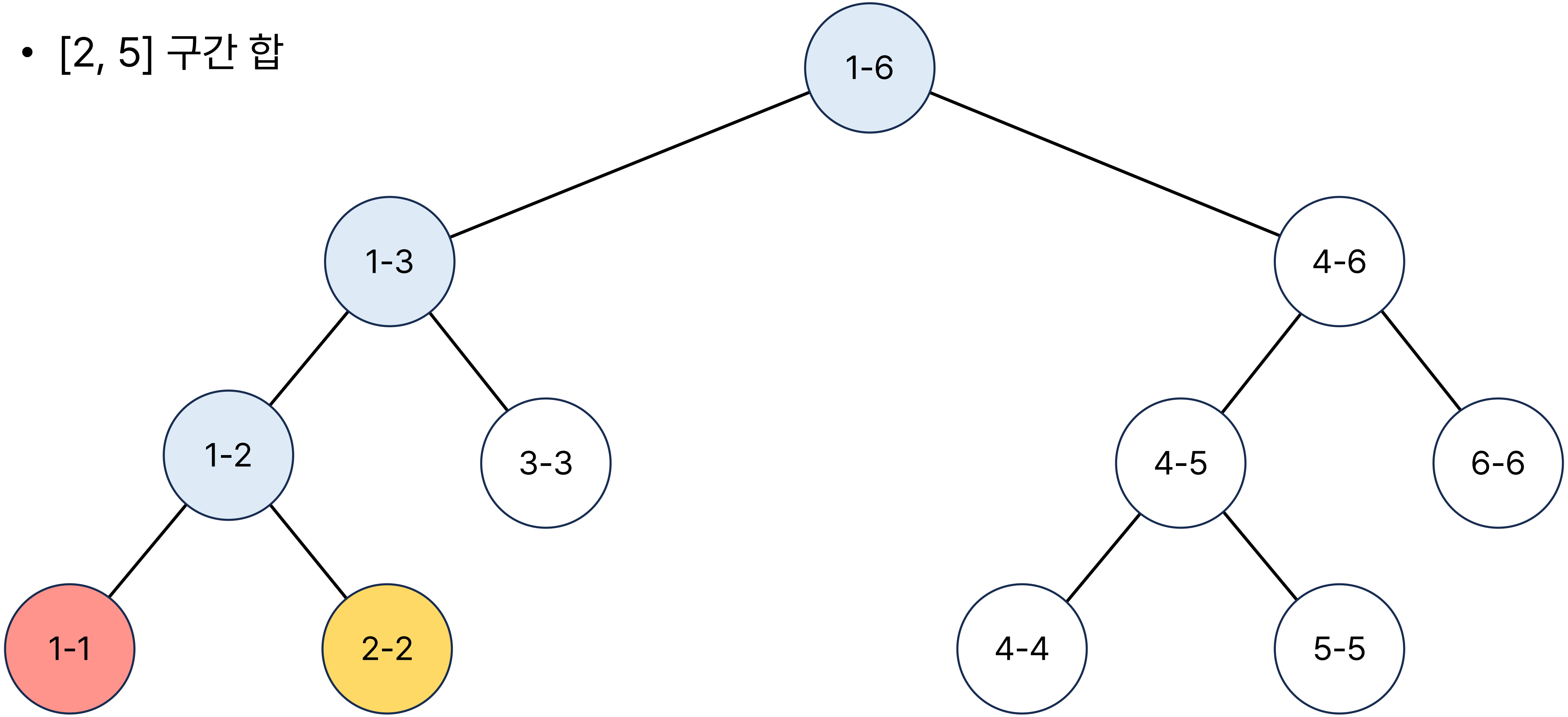
Segment Tree Query

- [2, 5] 구간 합



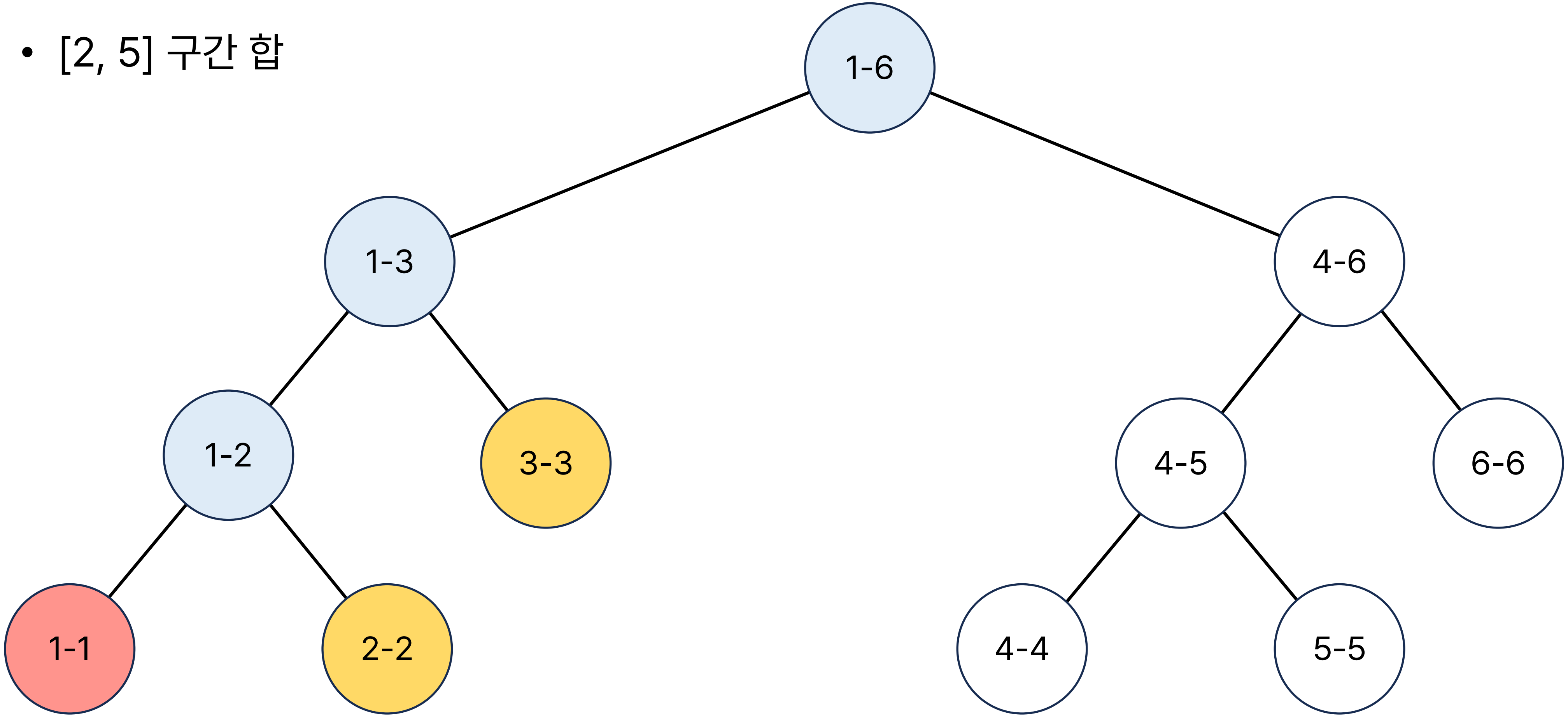
Segment Tree Query

- [2, 5] 구간 합



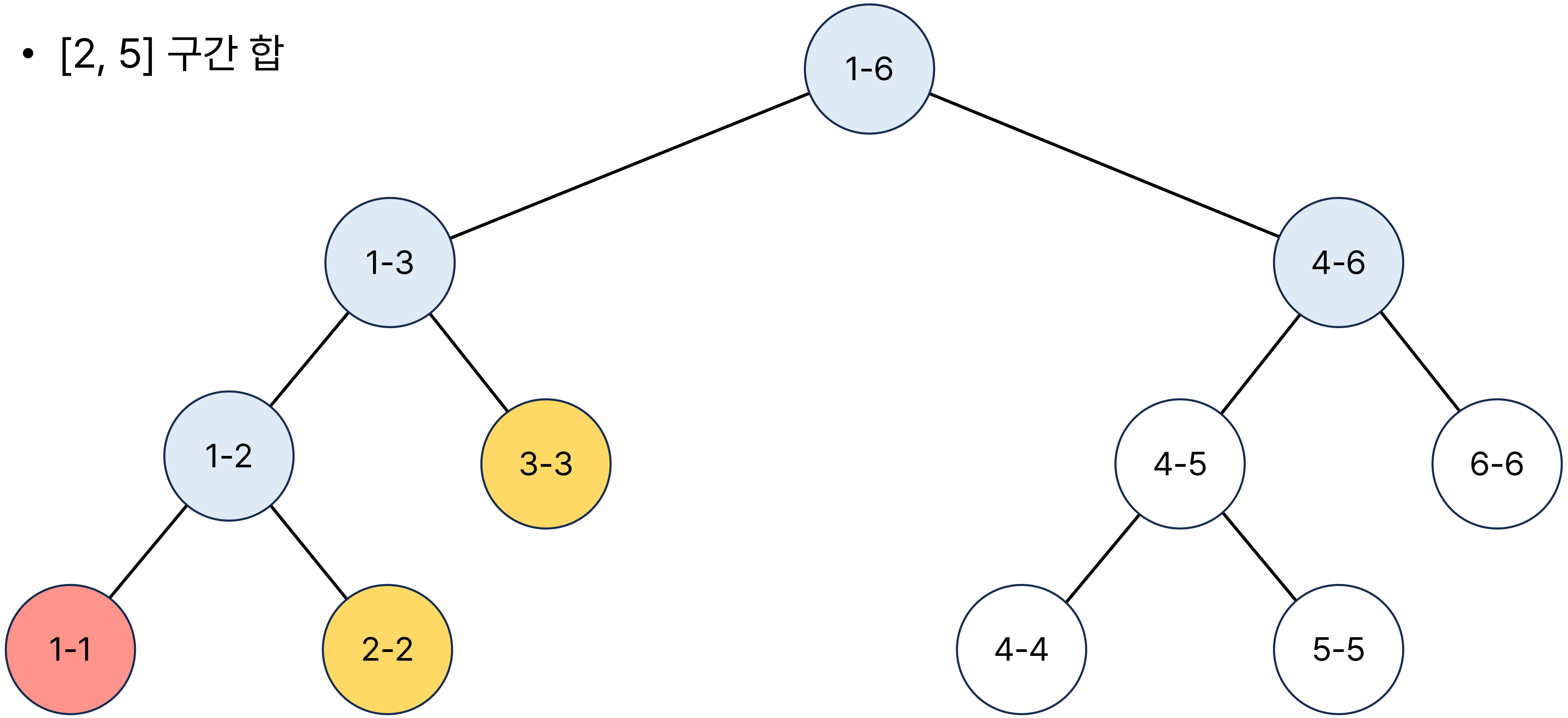
Segment Tree Query

- [2, 5] 구간 합



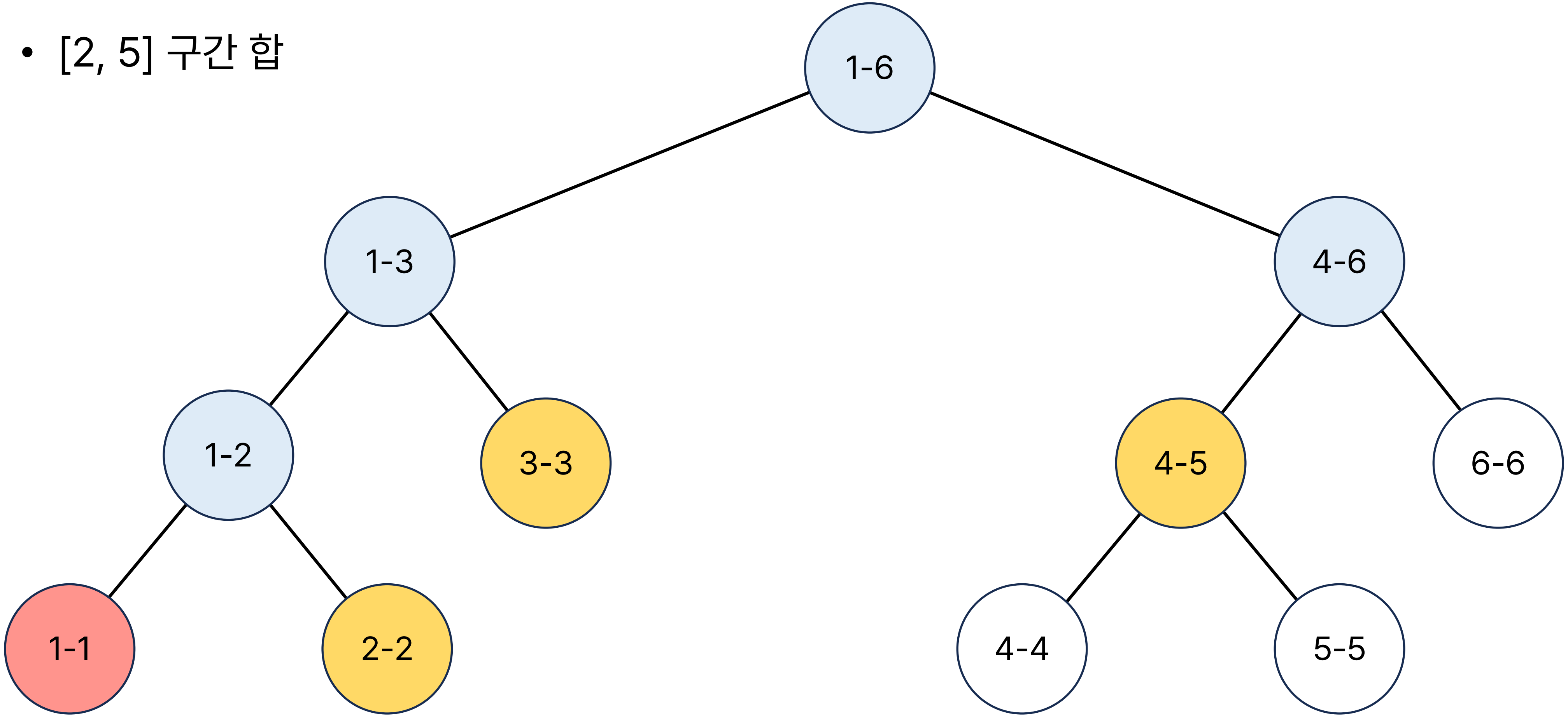
Segment Tree Query

- [2, 5] 구간 합



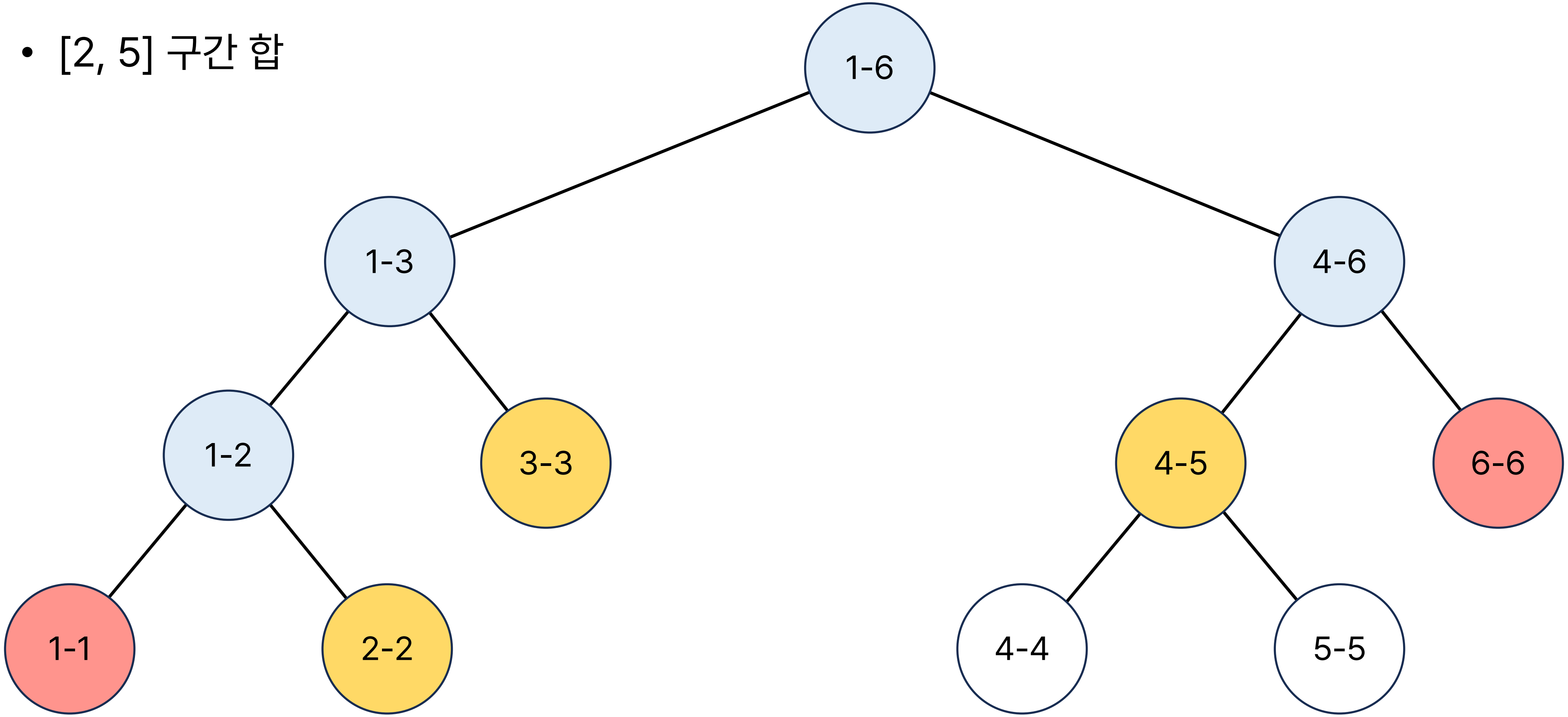
Segment Tree Query

- [2, 5] 구간 합



Segment Tree Query

- [2, 5] 구간 합



Segment Tree Query

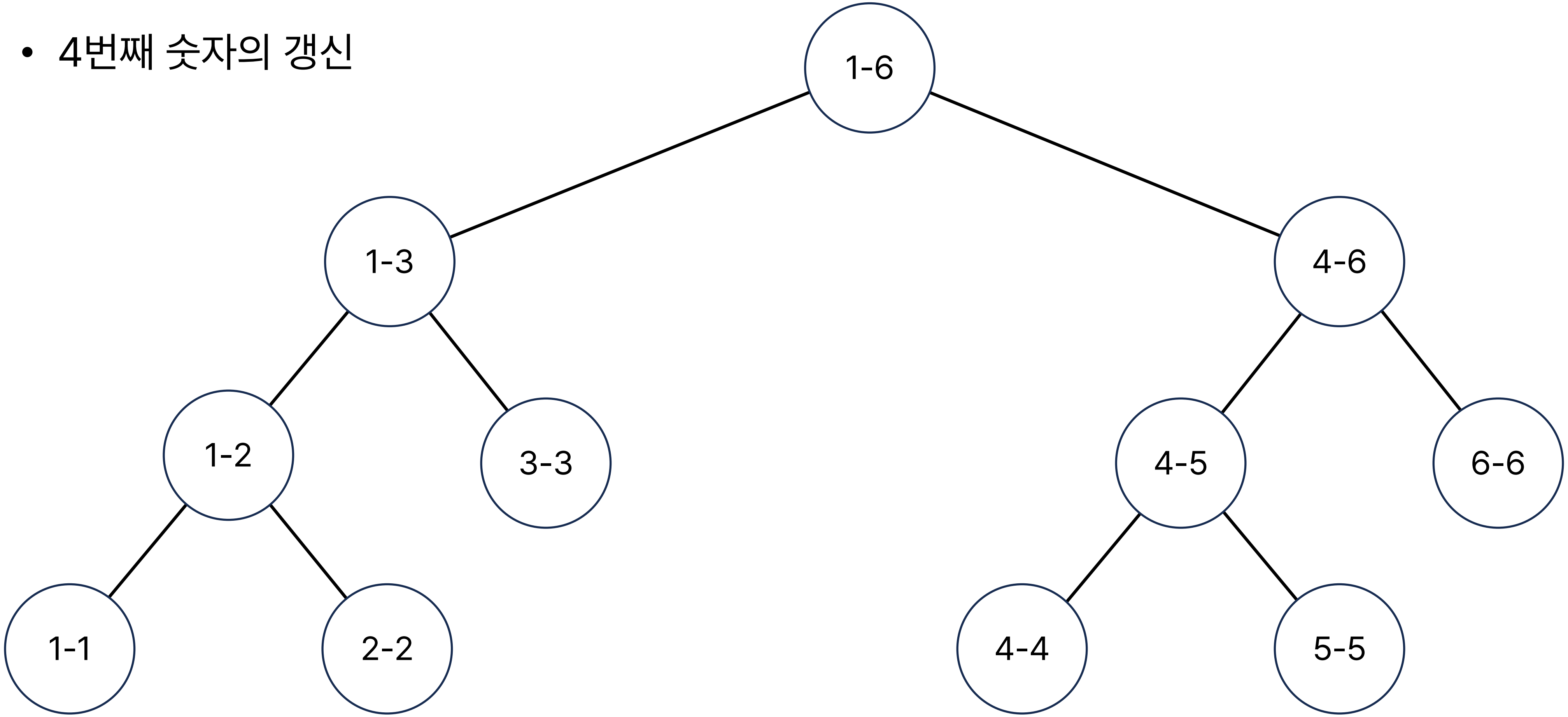
```
int query(int left, int right, int start, int end, int index) {
    if (end < left or right < start)
        return 0;
    if (left <= start and end <= right)
        return tree[index];
    return query(left, right, start, (start + end) / 2, index * 2)
         + query(left, right, (start + end) / 2 + 1, end, index * 2 + 1);
}
```

Segment Tree Update

- K번째 숫자의 값을 바꾼다면 관리하는 구간에 K번째 노드가 포함된 노드들은 누적 합을 갱신해야 한다
- 구간을 살펴보면서 쿼리와 동일하게 구간에 K번째 숫자가 포함되어 있는 경우 자식 중에도 K번째 숫자가 포함된 구간이 존재하므로 계속해서 갱신한다
- 반대로 구간에 K번째 숫자가 포함되지 않은 경우 자식 노드들에도 K번째 숫자가 포함되어 있지 않으므로 갱신을 멈춘다

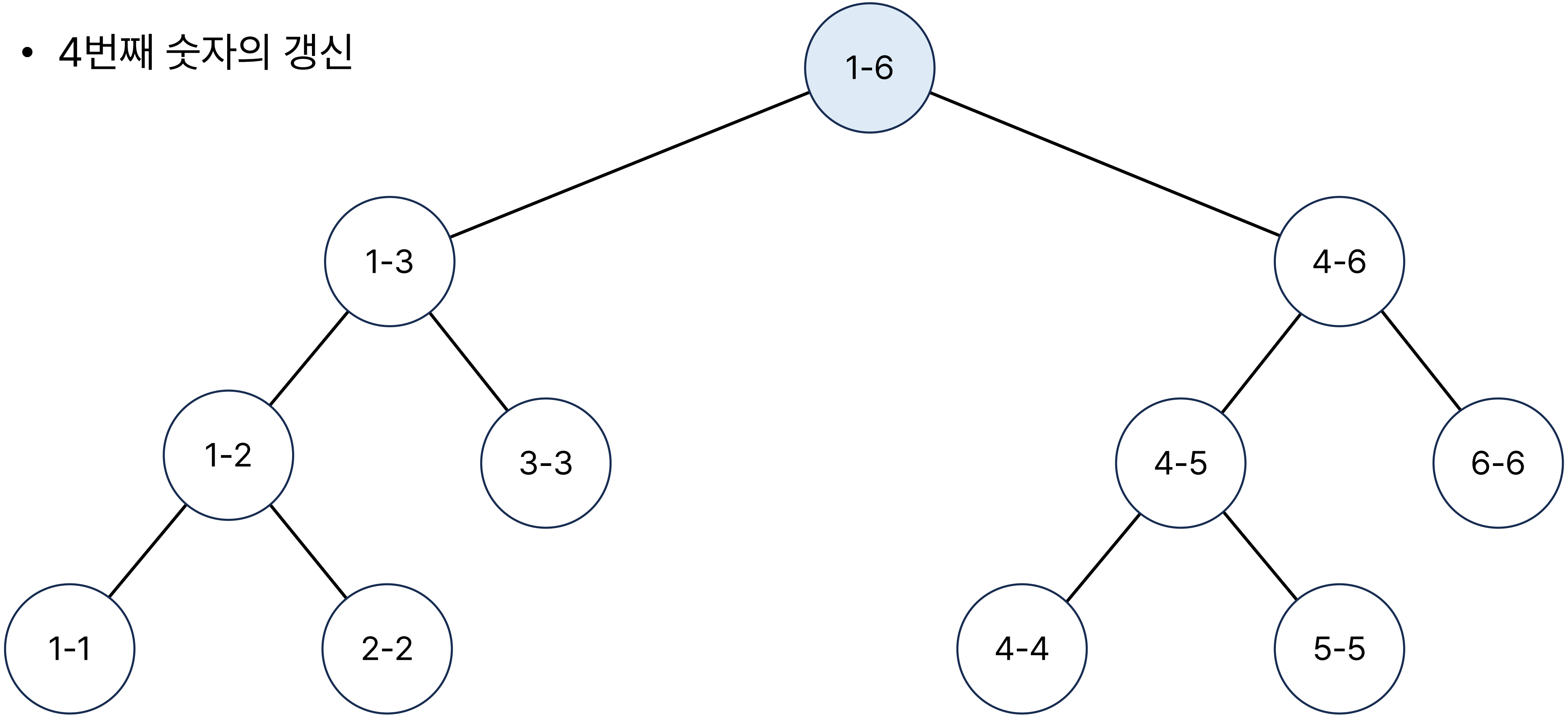
Segment Tree Update

- 4번째 숫자의 갱신



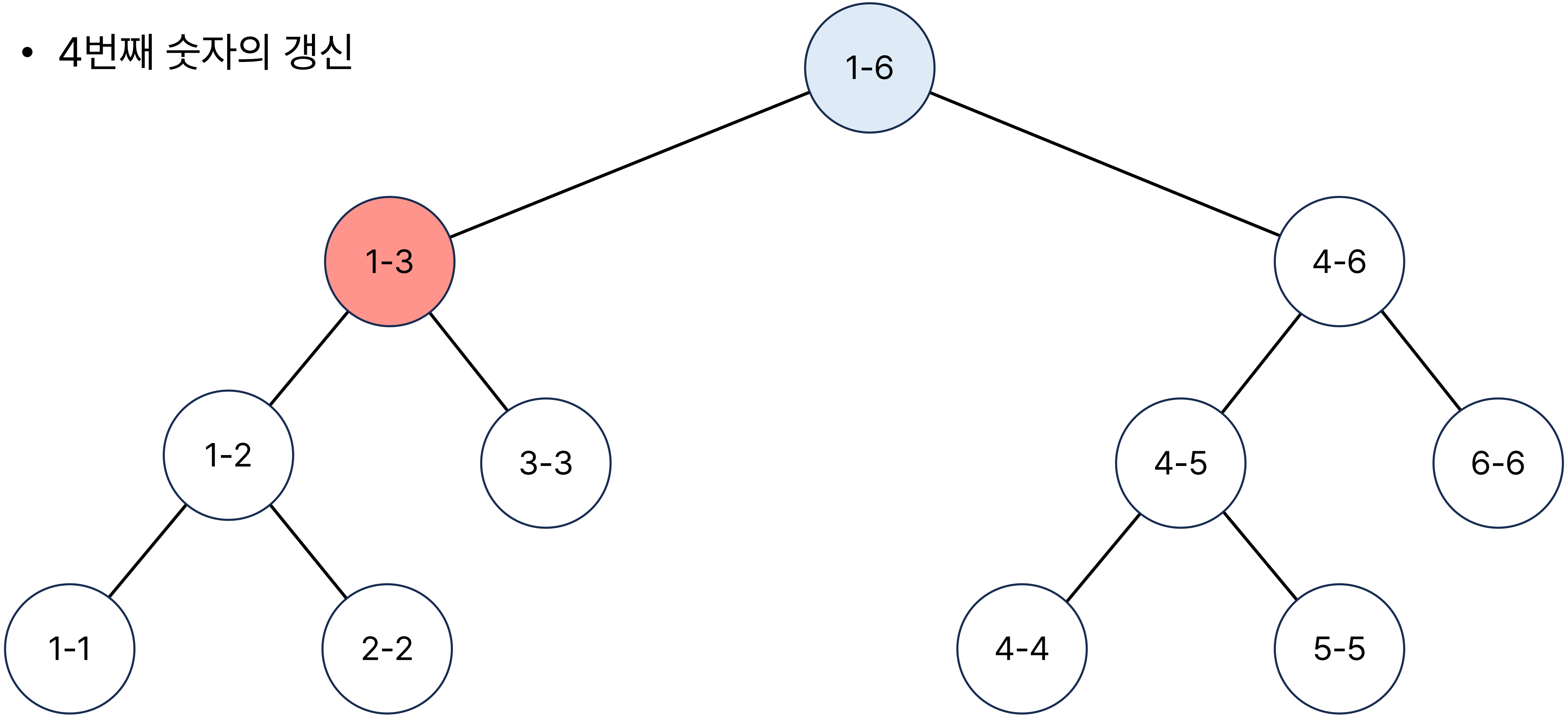
Segment Tree Update

- 4번째 숫자의 갱신



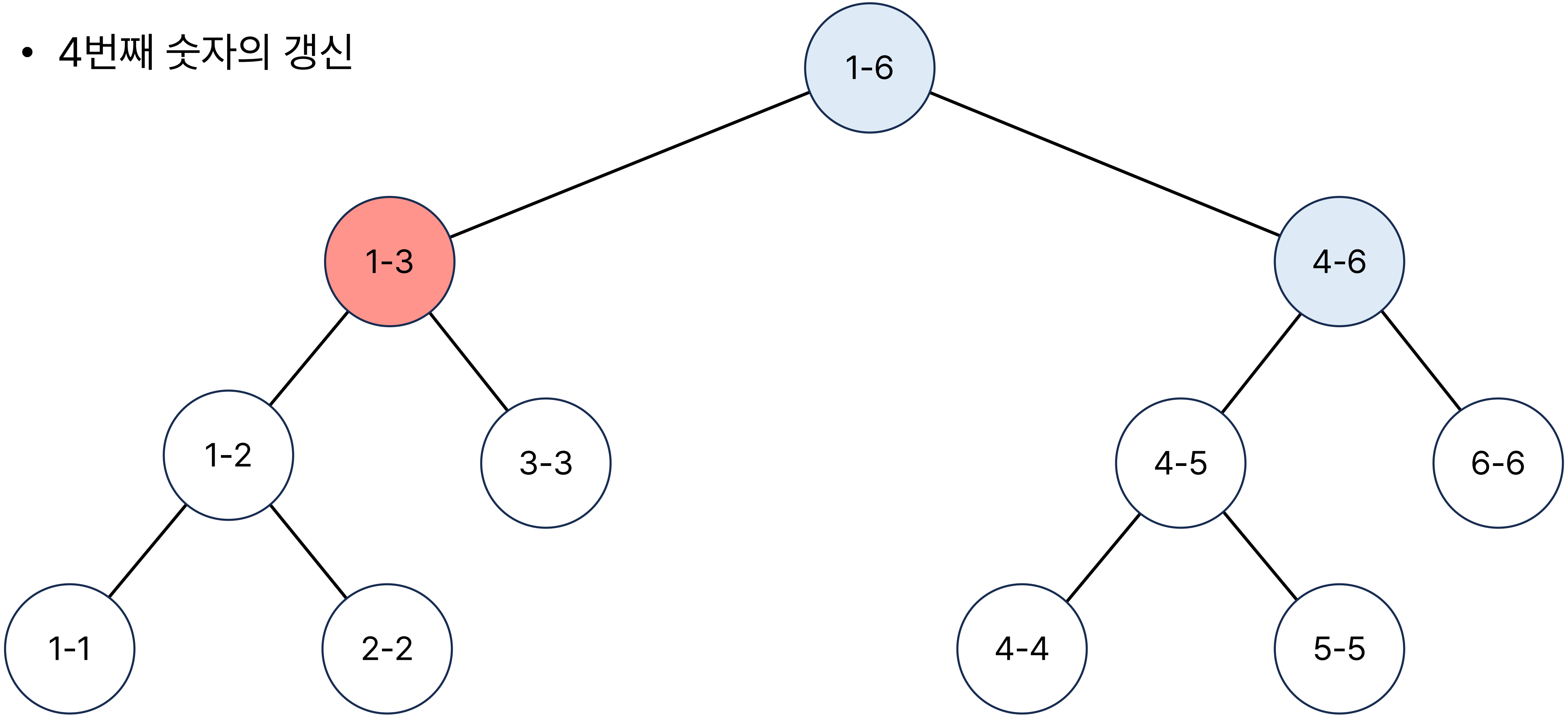
Segment Tree Update

- 4번째 숫자의 갱신



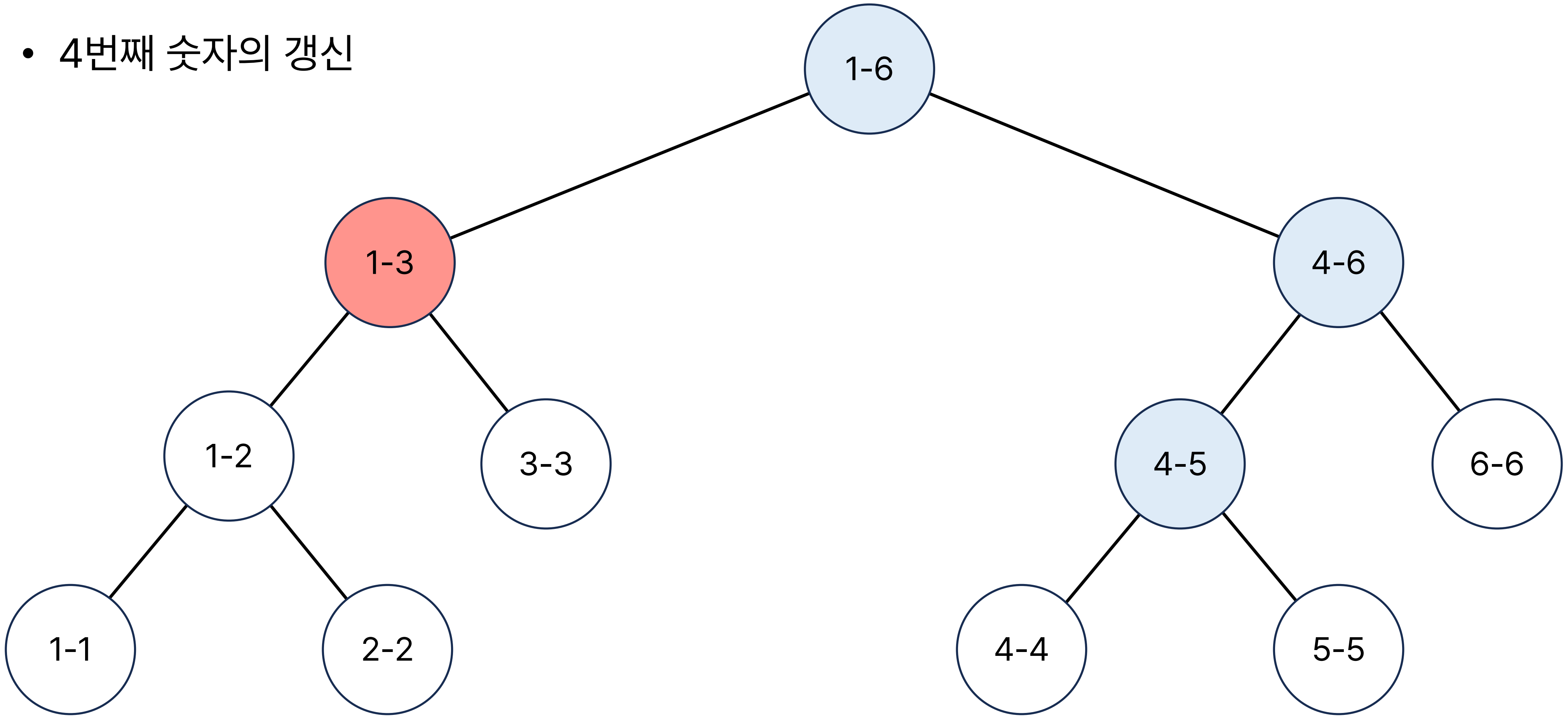
Segment Tree Update

- 4번째 숫자의 갱신



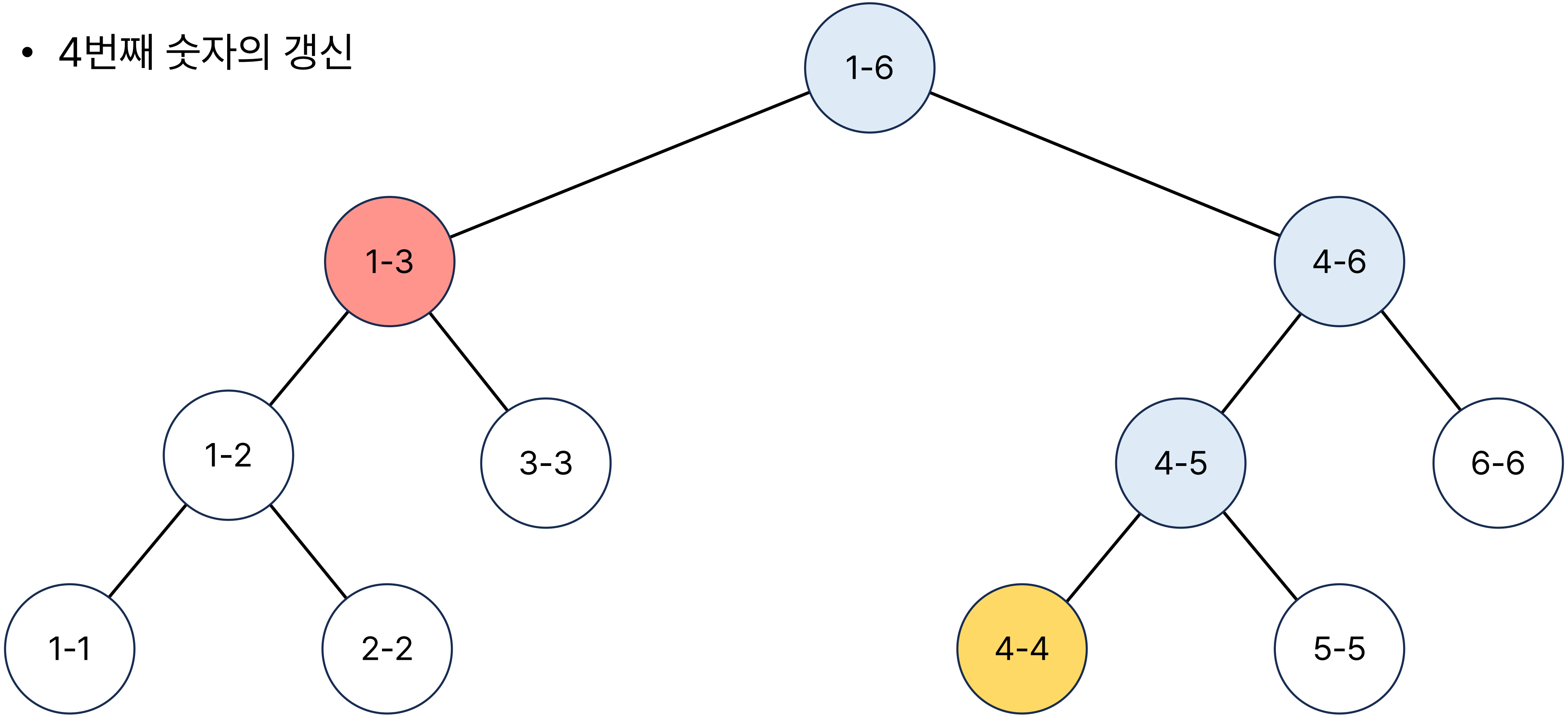
Segment Tree Update

- 4번째 숫자의 갱신



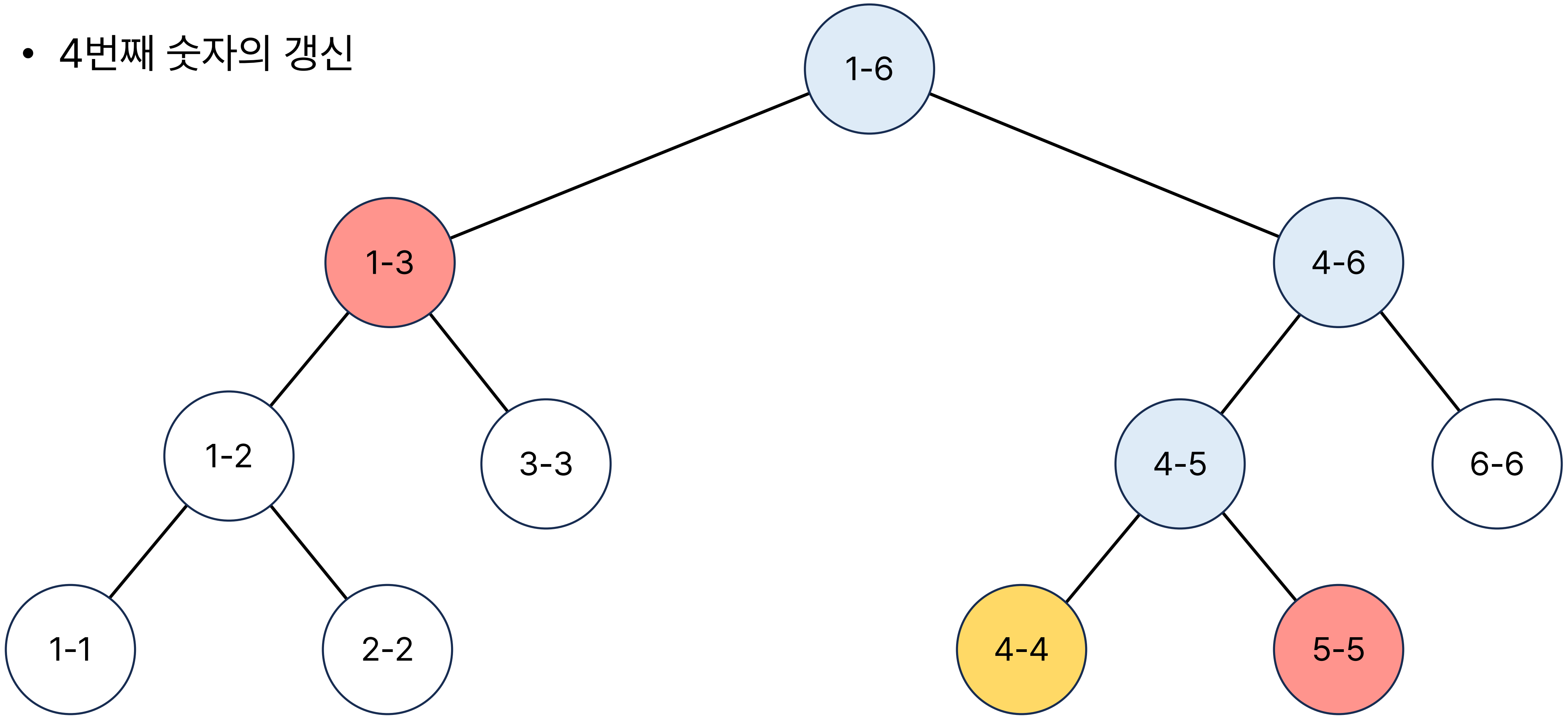
Segment Tree Update

- 4번째 숫자의 갱신



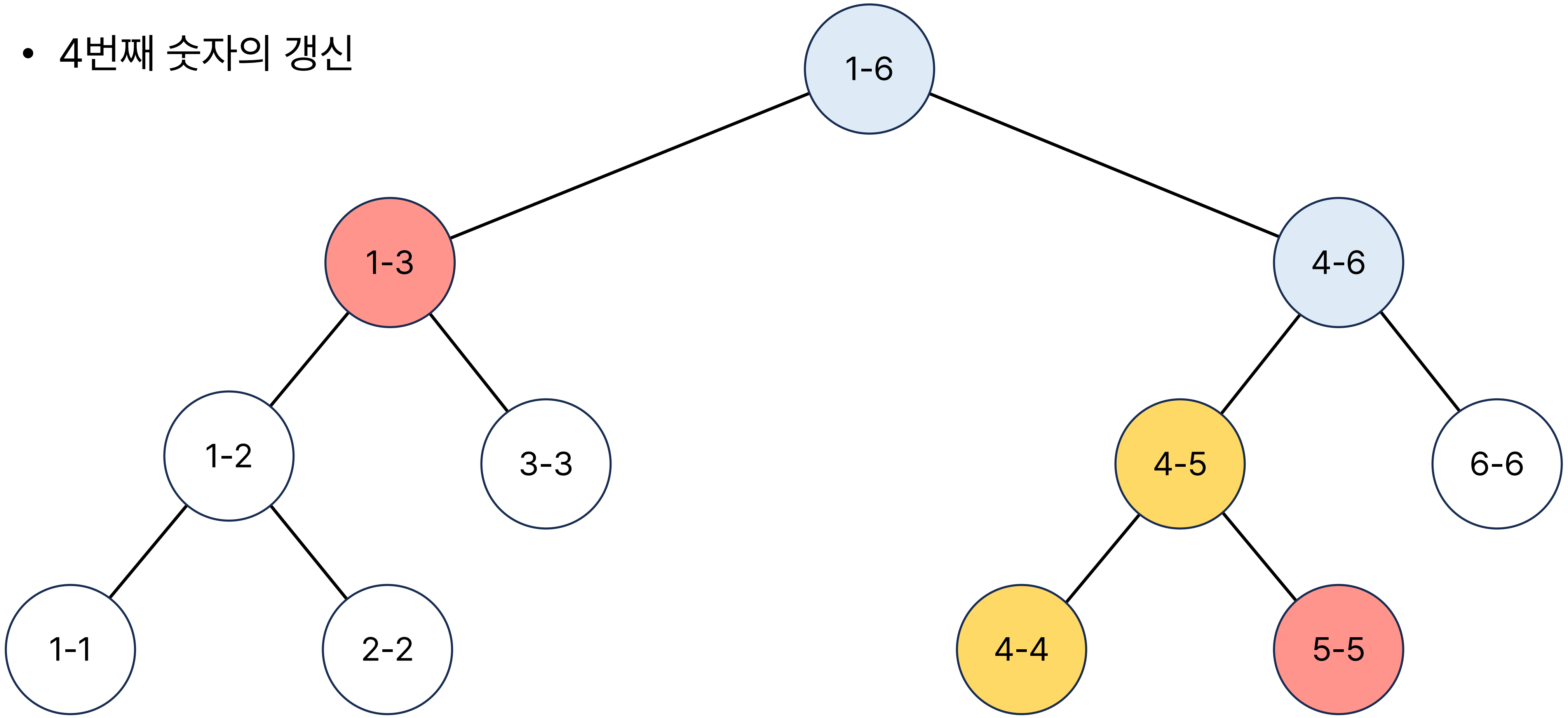
Segment Tree Update

- 4번째 숫자의 갱신



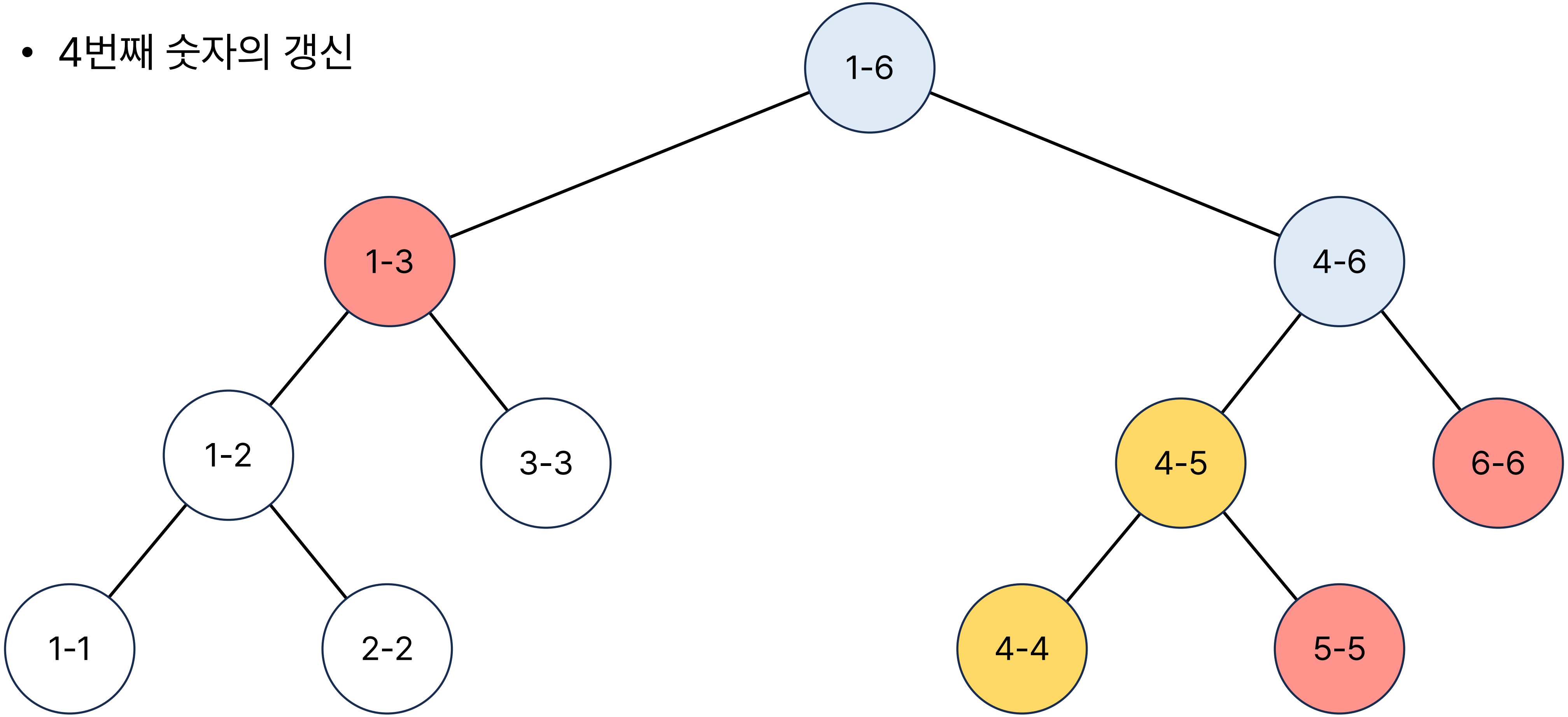
Segment Tree Update

- 4번째 숫자의 갱신



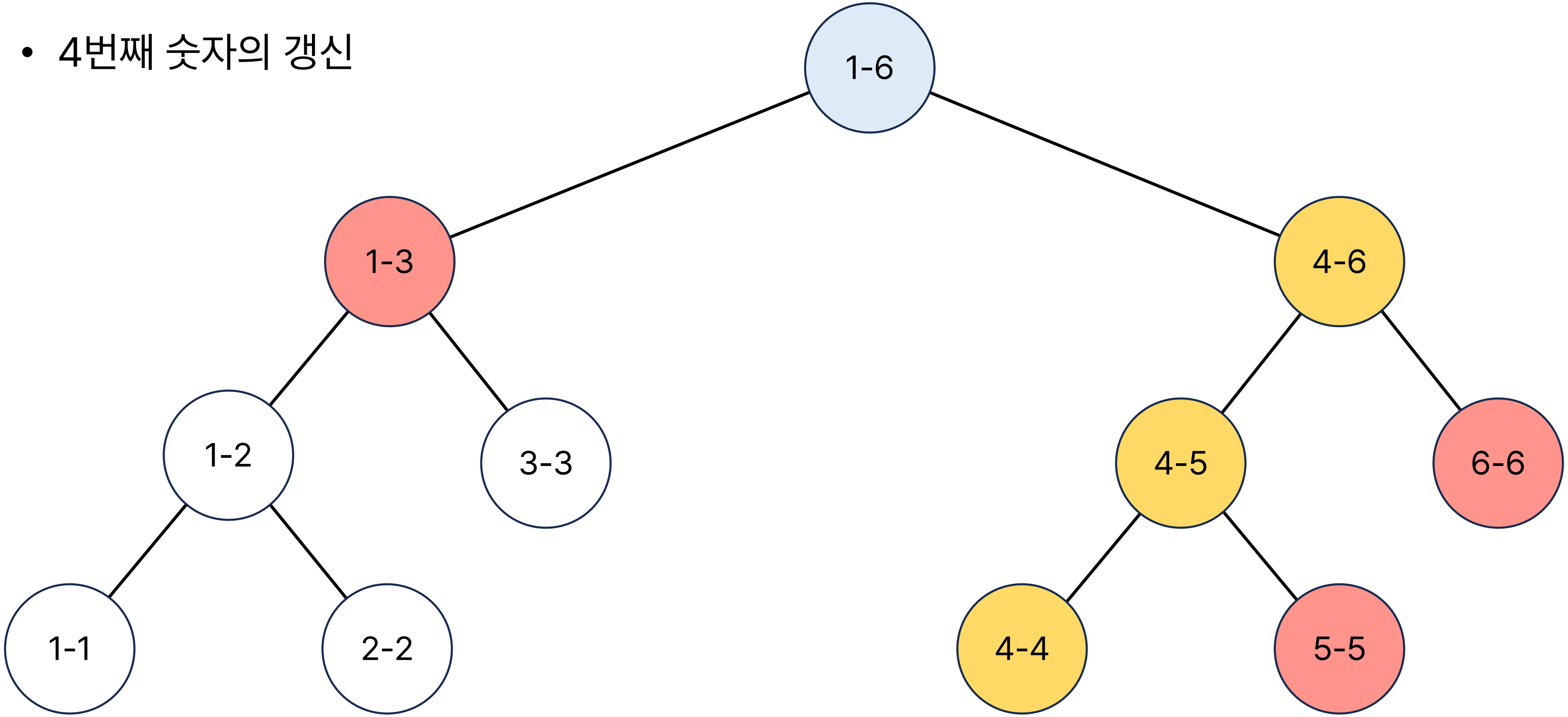
Segment Tree Update

- 4번째 숫자의 갱신



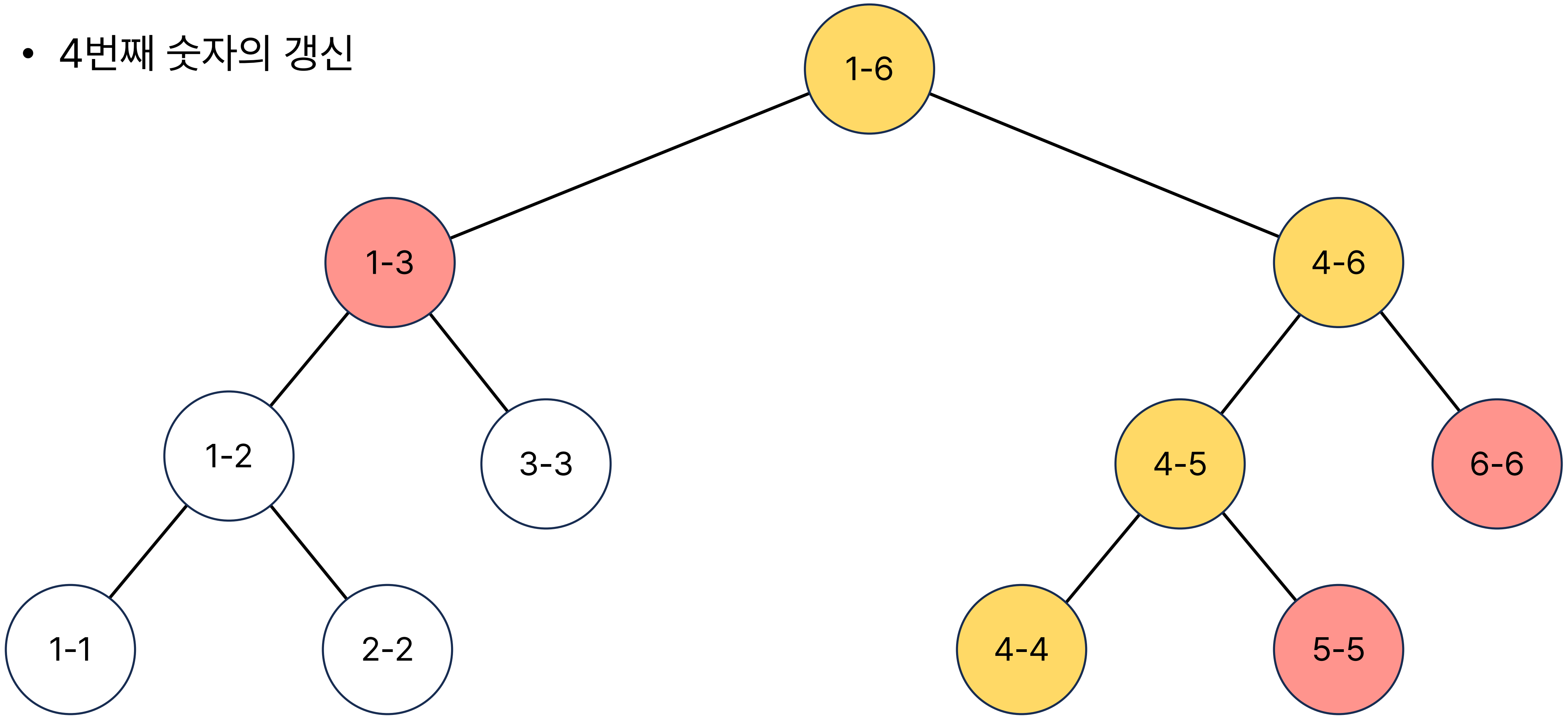
Segment Tree Update

- 4번째 숫자의 갱신



Segment Tree Update

- 4번째 숫자의 갱신



Segment Tree Update

```
void update(int key, int value, int start, int end, int index) {
    if (key < start or end < key)
        return;
    if (start == end) {
        tree[index] = value;
        return;
    }
    update(key, value, start, (start + end) / 2, index * 2);
    update(key, value, (start + end) / 2 + 1, end, index * 2 + 1);
    tree[index] = tree[index * 2] + tree[index * 2 + 1];
}
```

Segment Tree Update

- 또는 이분 탐색으로 [K, K]구간을 관리하는 노드를 찾아갈 수 있다
- 왼쪽 자식은 [start, (start + end) / 2]의 구간을 관리하므로 K가 (start + end) / 2 보다 크다면 오른쪽 자식 노드로 내려가고 그렇지 않다면 왼쪽 자식 노드로 내려가면 K, K]구간을 관리하는 노드를 찾아갈 수 있다

Segment Tree Update

```
void update(int key, int value, int start, int end, int index) {
    if (start == end) {
        tree[index] = value;
        return;
    }
    if (key <= (start + end) / 2)
        update(key, value, start, (start + end) / 2, index * 2);
    else
        update(key, value, (start + end) / 2 + 1, end, index * 2 + 1);
    tree[index] = tree[index * 2] + tree[index * 2 + 1];
}
```

Segment Tree

- 세그먼트 트리는 구간 합 외에도 결합법칙을 성립하는 연산을 모두 적용할 수 있다
- 곱셈, 최댓값, 최솟값, 비트 연산 등 여러 결합법칙이 성립하는 연산을 세그먼트 트리로 만들 수 있다
- 누적 합의 경우 최댓값, 최솟값 등의 연산을 구현하지 못했지만 세그먼트 트리는 가능하다

문제

- 구간 합 구하기 BOJ 2042
- 최솟값과 최댓값 BOJ 2357
- 구간 곱 구하기 BOJ 11505
- 공장 BOJ 7578