고급 자료 구조

CS202 Lecture Notes

School of Computing
KAIST

쉬운 최솟값 문제

- 오늘은 배열에서 구간의 합과 최솟값 등을 구하는 방법에 대해서 공부합니다.
- 배열에서 특정 구간에 대한 최솟값을 어떻게 찾을 수 있을까요?
- 간단한 루프 문으로 찾을 수 있습니다.

Code

```
int min_value = 1000000000;
for (int i = start; i <= end; i++){
    min_value = min(min_value, arr[i]);
}
```

쉬운 최솟값 문제

- 끝일까요? 그럴 리가..
- 여러 개의 구간이 주어지는 상황을 생각해 봅시다.
  구간마다 $O(N)$ 의 시간이 걸리기 때문에 느립니다.
- 어떻게 해결할 수 있을까요?

쉬운 합 문제

- 보기보다 쉽지 않아 보이니, 일단 다른 문제로 넘어갑시다.
- 배열에서 특정 구간에 대한 합을 어떻게 찾을 수 있을까요?

쉬운 합 문제

- 보기보다 쉽지 않아 보이니, 일단 다른 문제로 넘어갑시다.
- 배열에서 특정 구간에 대한 합을 어떻게 찾을 수 있을까요?
- 제가 또 여러 개의 구간을 주면서 괴롭히겠죠? 하지만 우리는 부분 합이라는 개념을 알고 있습니다.

Code

```
for (int i = 1; i <= n; i++){
    sum[i] = arr[i] + sum[i - 1];
}
for (int i = 1; i <= num_of_query; i++){
    int start, end;
    scanf("%d %d",&start,&end);
    printf("%lld\n", sum[end] - sum[start - 1]);
}
```

쉬운 합 문제

- 이번에는 완벽했을까요?

쉬운 합 문제

- 이번에는 완벽했을까요?
- 그럴 리가...
- 쿼리 중간에 제가 **배열의 원소를 변경**했다고 생각해 봅시다. 다음 쿼리가 들어오기 전에 부분합을 다시 만들어야 할까요?
- 그러면 쿼리 하나에 또 $O(N)$ 이 들겠네요.
- 어떻게 해결할 수 있을까요?

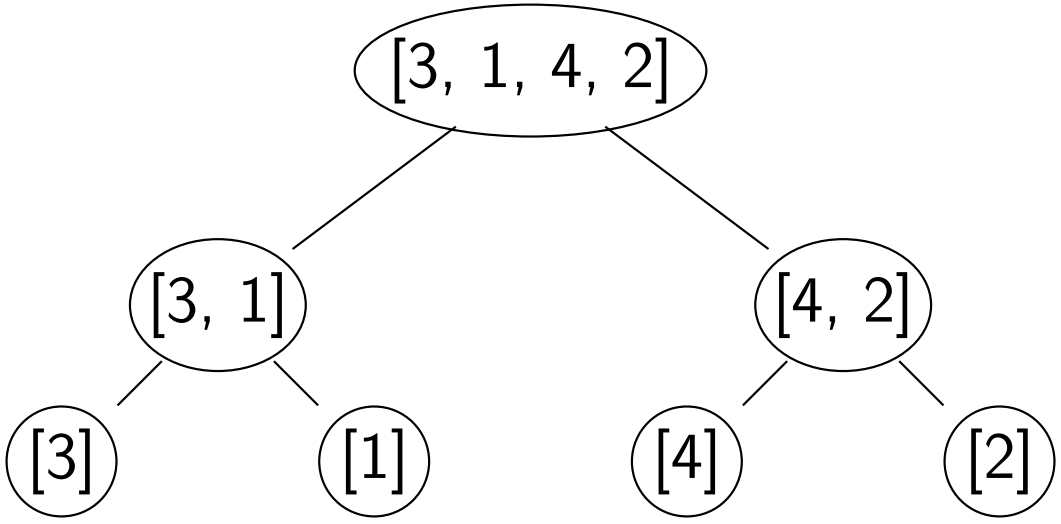
어려운 문제

- 구간의 최솟값, 합 등의 연산을 빠르게 하는 방법이 필요합니다.
- 도전해야 할 문제들이 크게 두 가지 있습니다.
- **1. 많은 쿼리:** 단순히 순서대로 배열을 순회하는 것은 너무 느립니다. 적절한 전처리로 쿼리를 빠르게 처리해야 합니다.
- **2. 변화하는 내용:** 배열의 원소가 변화할 수 있습니다. 부분합은 쿼리를 빠르게 처리하는 전처리였으나, 내용이 변화할 때 비효율적입니다. 이 전처리는 변화에 빠르게 응답해야 합니다.

세그먼트 트리

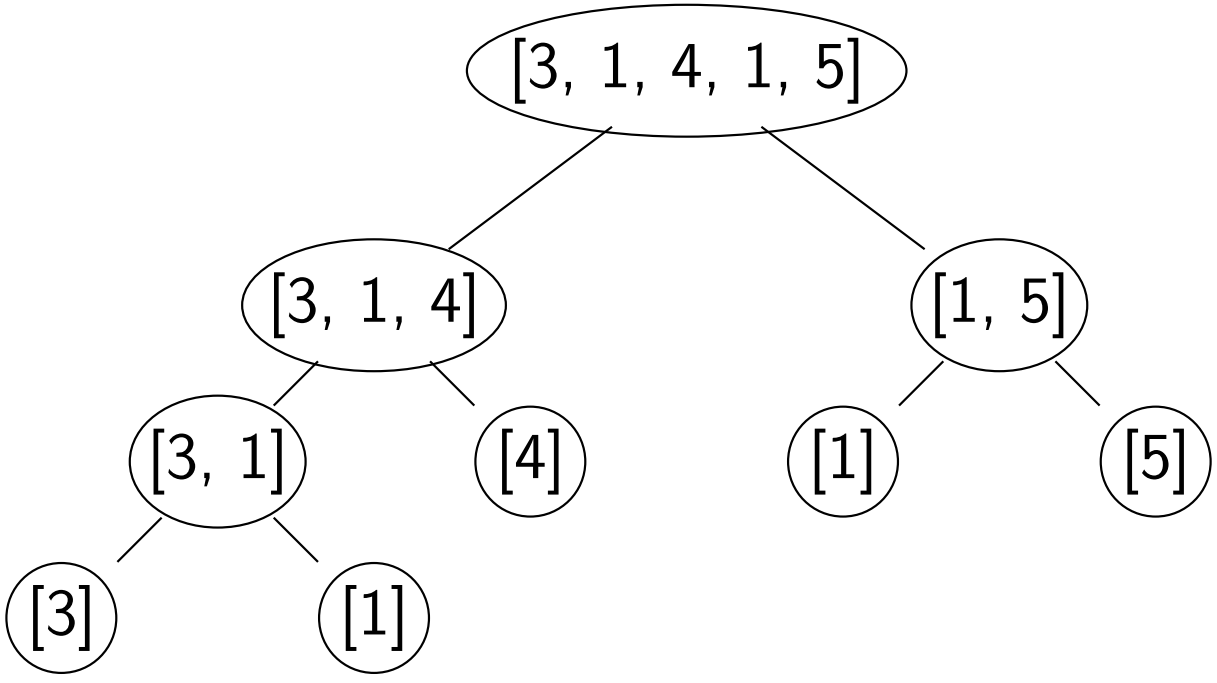
- 세그먼트 트리는 전체 배열을 반복적으로 절반으로 나눠가면서 구성되는 형태의 자료구조입니다.

- 예를 들어, 배열 [3, 1, 4, 2] 는 다음과 같이 분할됩니다.

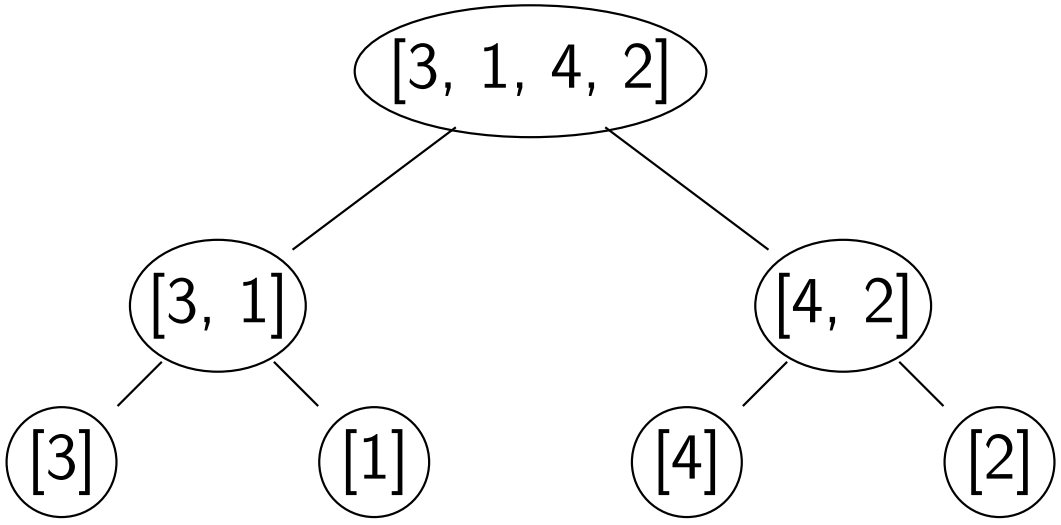


세그먼트 트리

- 크기가 홀수일 때는:



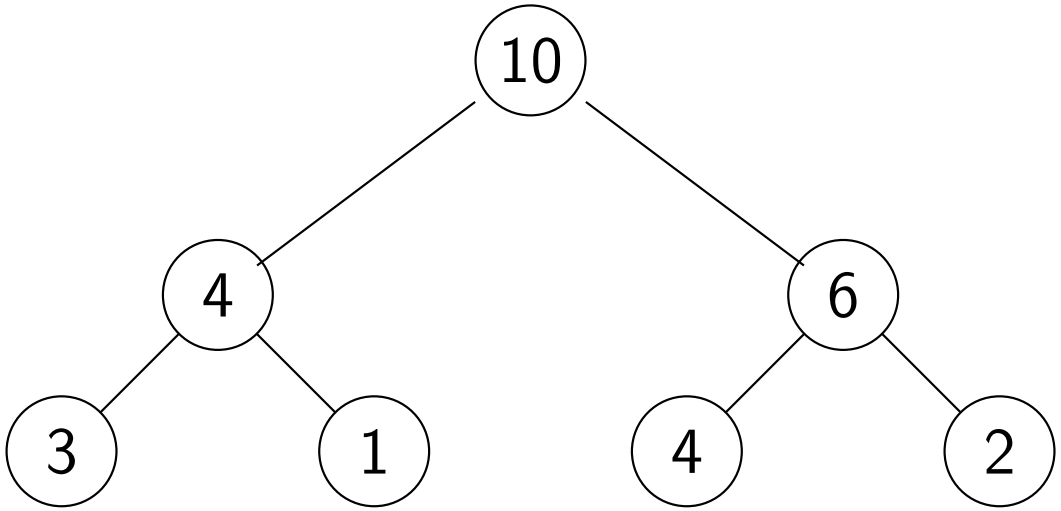
세그먼트 트리



- 트리의 높이가 높지 않고, 각 노드가 원소 하나 뿐이거나, 왼쪽 자식과 오른쪽 자식의 연결(concatenation) 임을 알 수 있습니다.

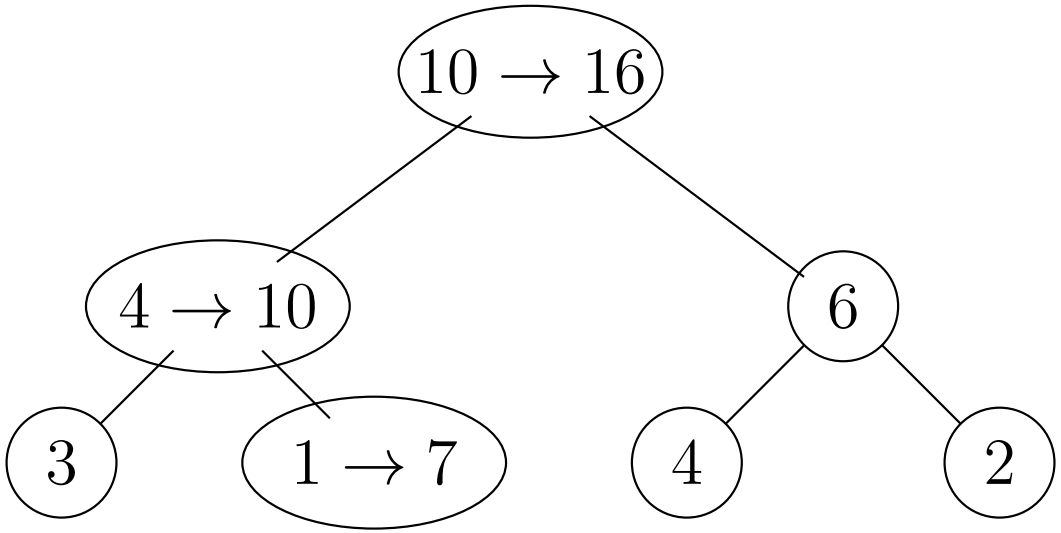
세그먼트 트리

- 실제로는 배열을 저장하는 것이 아니라, 우리가 알고 싶은 정보만 저장합니다. 예를 들어 합을 저장하면 이렇습니다.



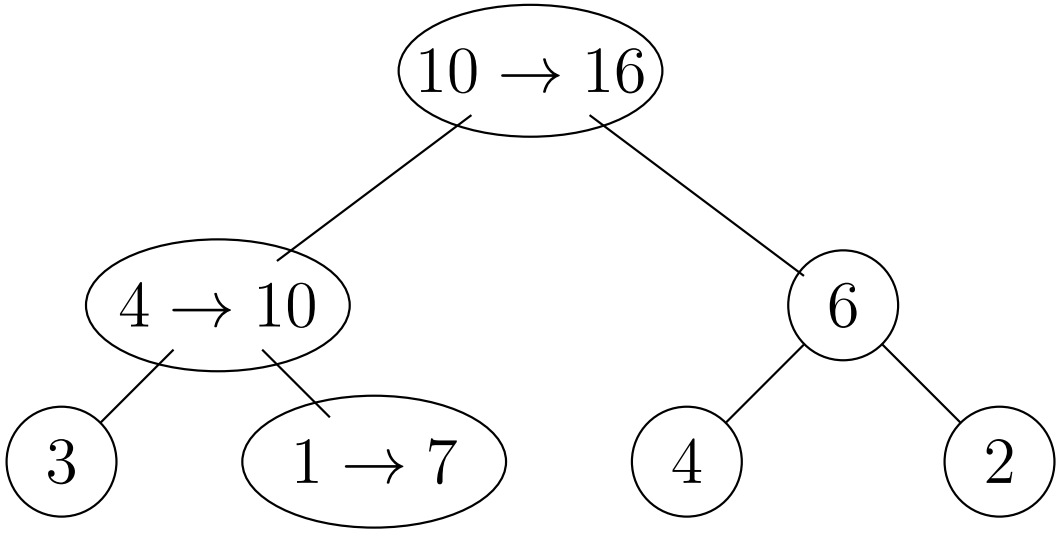
세그먼트 트리

- 이제 원소를 변화시켜 봅시다.



세그먼트 트리

- 이제 원소를 변화시켜 봅시다.

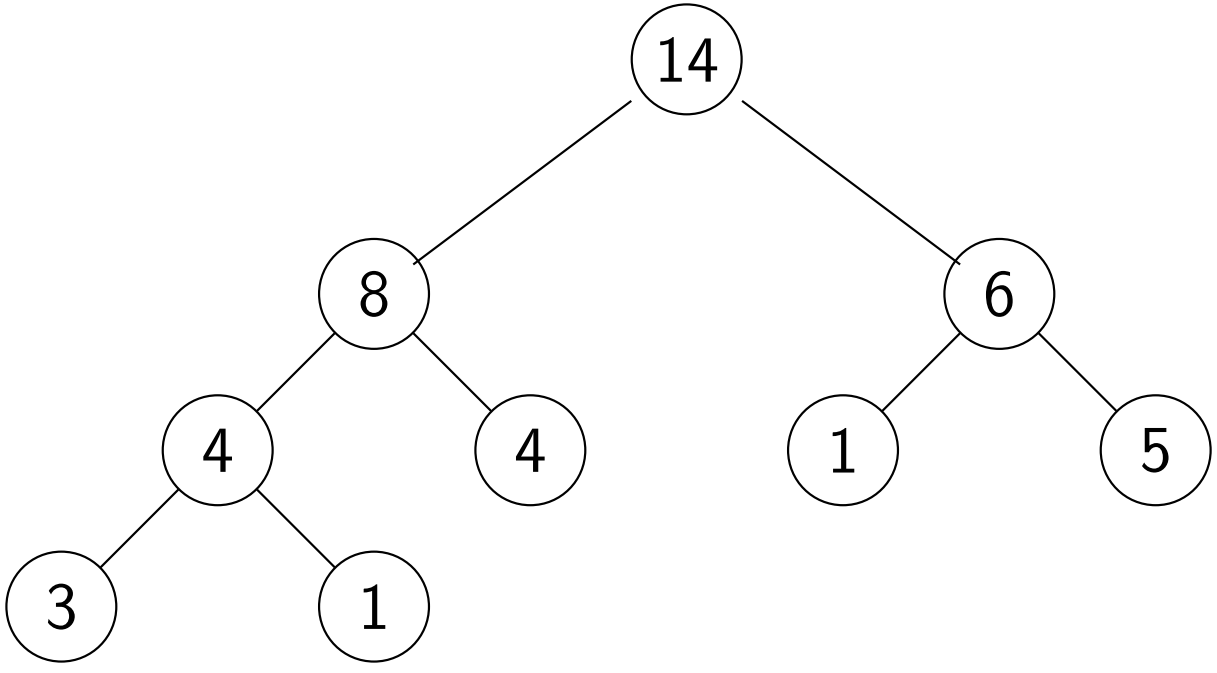


- 하나의 원소를 가진 **리프 노드** 에 변화가 가해집니다. 이 변화는 리프 노드에서 **부모** 를 타고 전해집니다.

- 그 외 다른 노드들은 아무 변화가 없습니다.

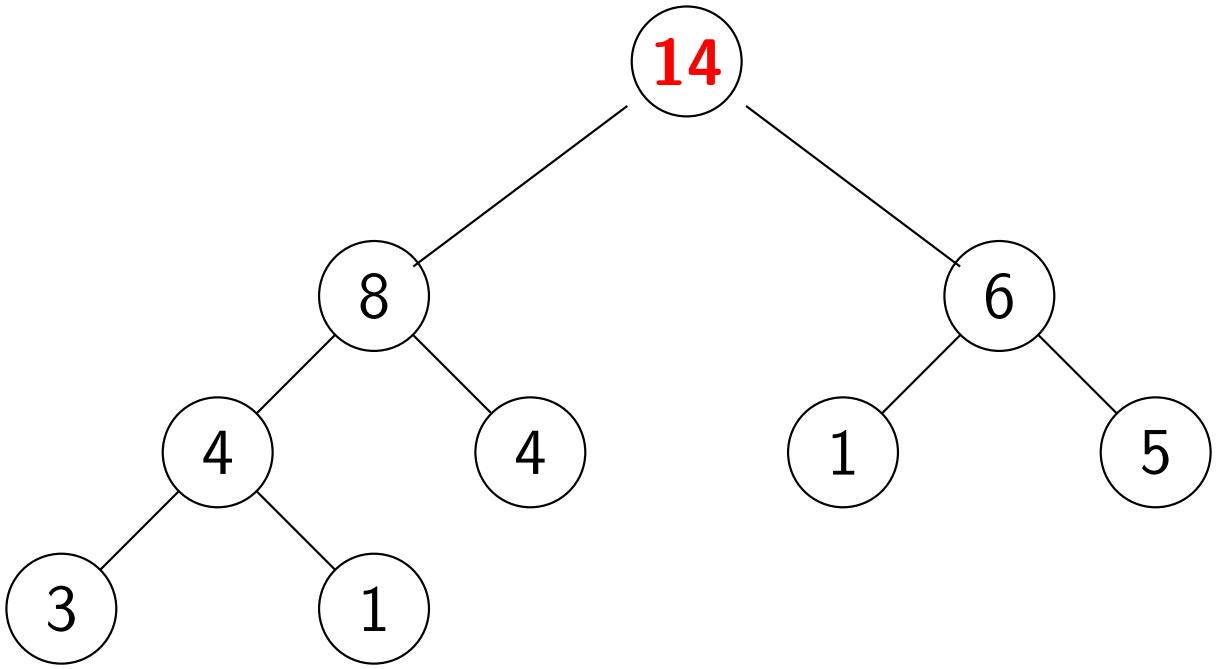
세그먼트 트리

- 이제 [3, 1, 4, 1] 구간에 합 쿼리를 날려 봅시다.



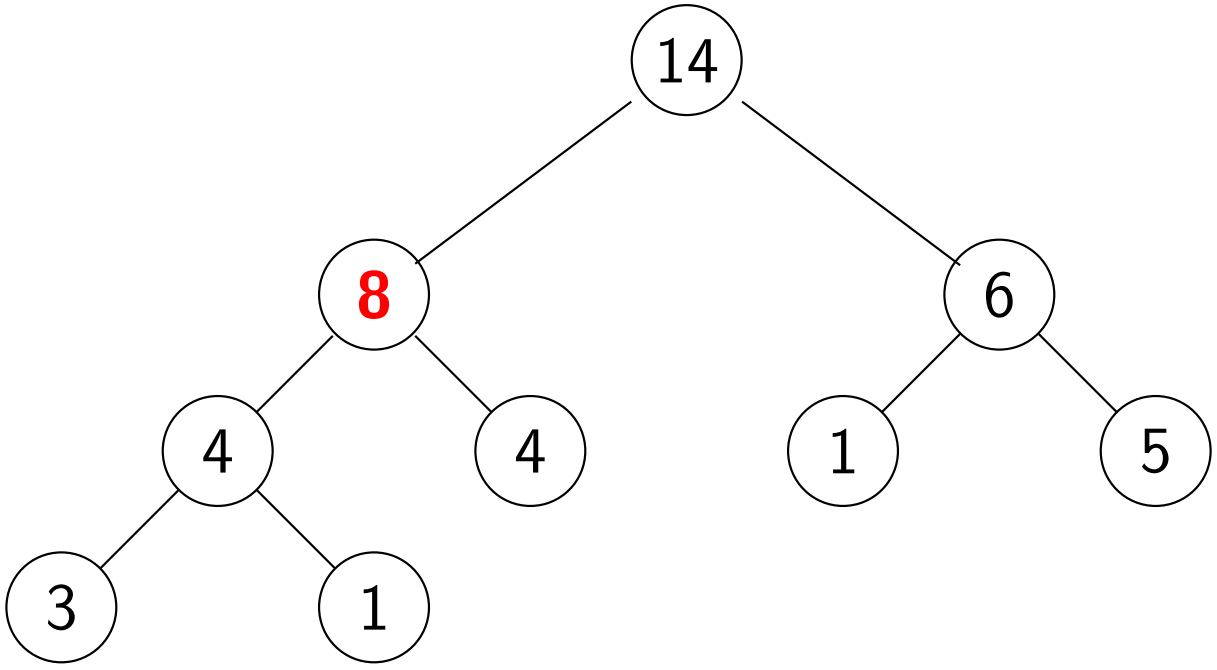
세그먼트 트리

- 이제 [3, 1, 4, 1] 구간에 합 쿼리를 날려 봅시다.
- 루트에서 시작해서 재귀적으로 움직입니다.



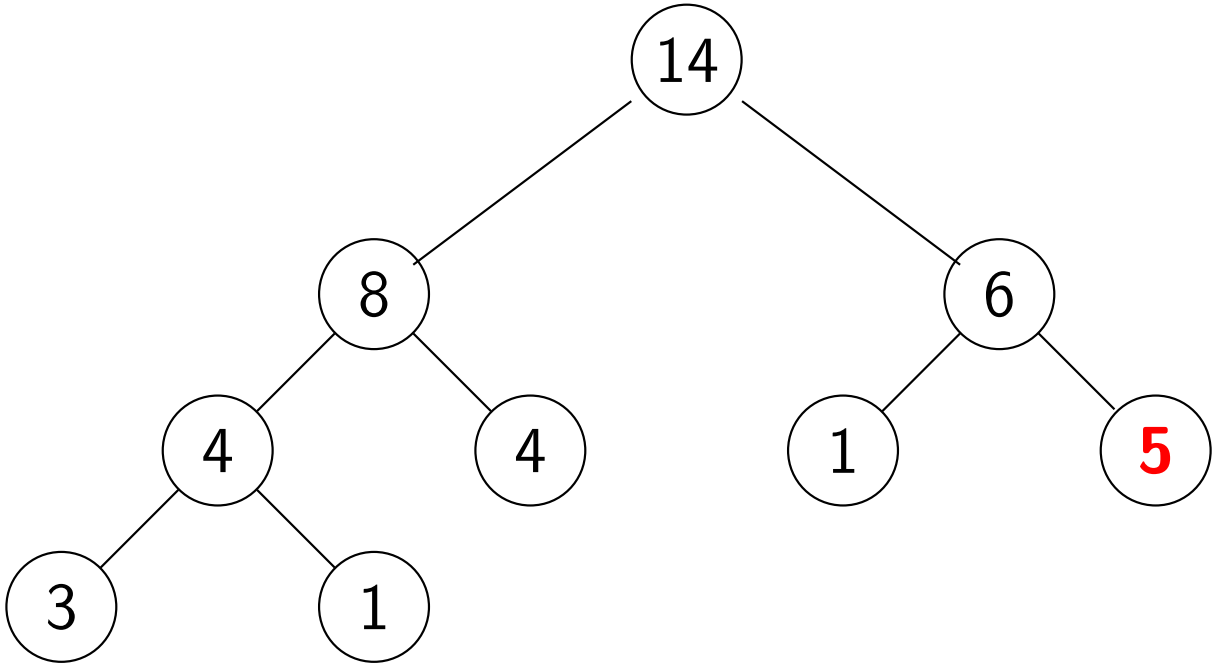
세그먼트 트리

- 현재 보고 있는 노드가 원하는 구간에 포함되면, 노드에 적힌 합은 답의 일부가 되니 전부 더해줘도 됩니다.
  그리고 아래로 더 내려갈 필요도 없습니다.



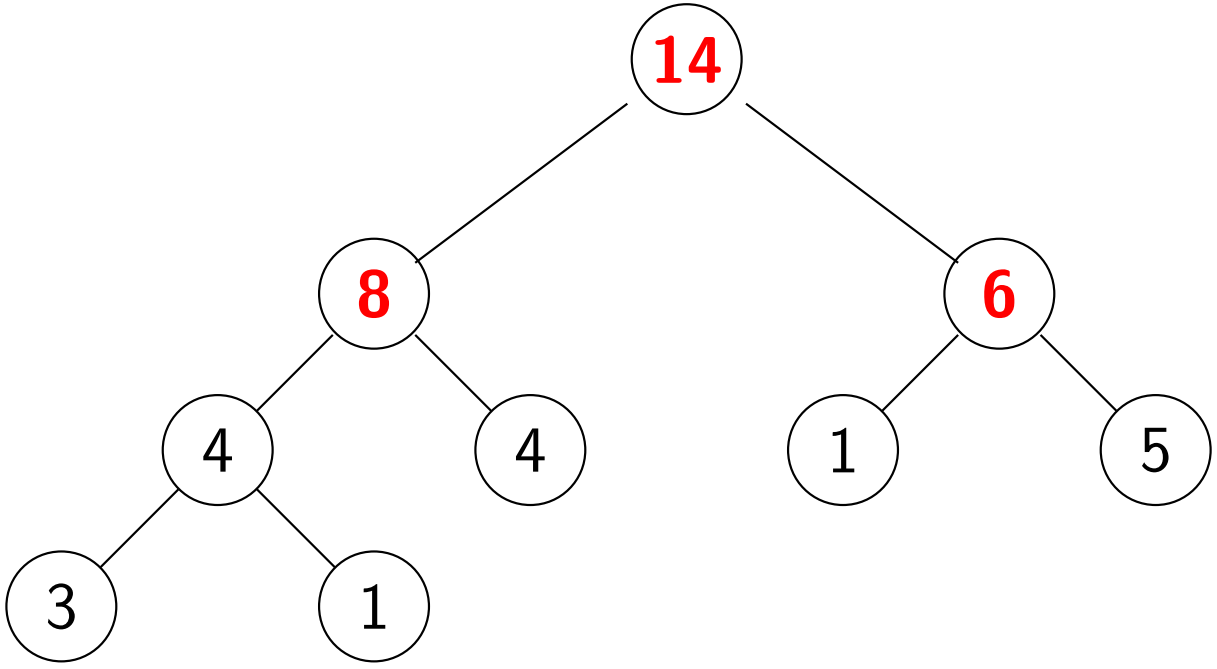
세그먼트 트리

- 현재 보고 있는 노드가 원하는 구간과 겹치지 않으면, 더 내려갈 필요가 없습니다. 아래를 무시하면 되기 때문입니다.



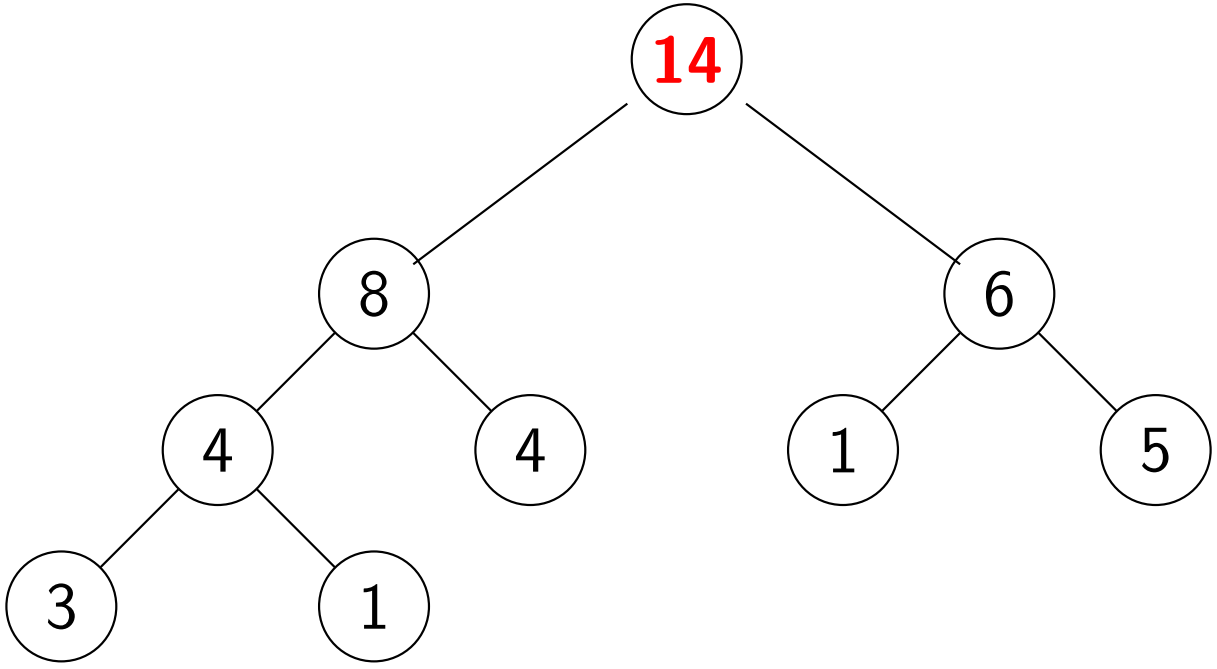
세그먼트 트리

- 아니면 두 노드를 전부 탐색합니다.



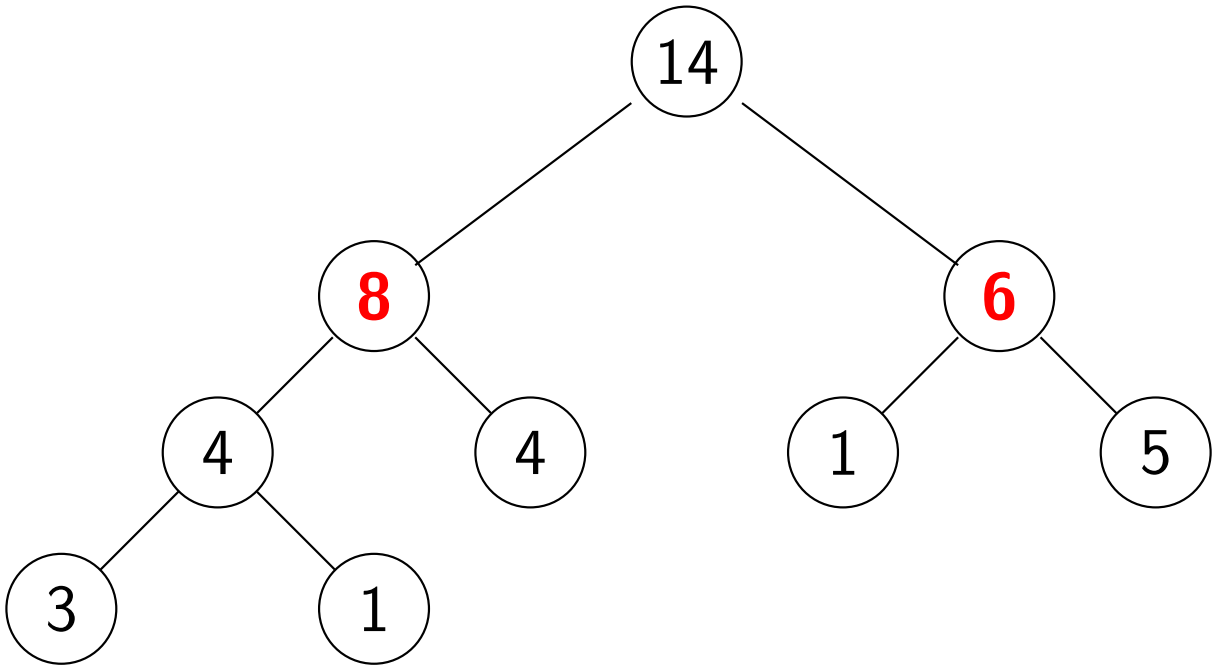
세그먼트 트리

- 이런 식으로 탐색이 진행됩니다.



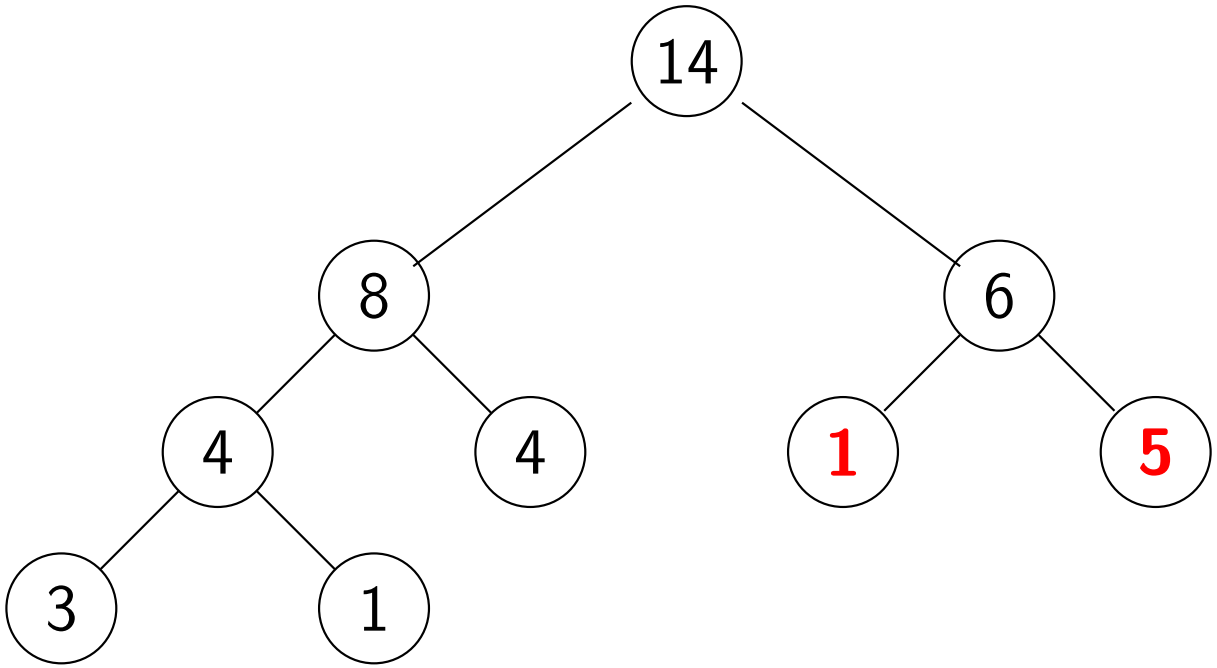
세그먼트 트리

- 이런 식으로 탐색이 진행됩니다.
- 8에서 답을 더하고 탐색이 종료됩니다.



세그먼트 트리

- 이런 식으로 탐색이 진행됩니다.
- 1에서 답을 더하고 탐색이 종료됩니다.
- 5에서는 답을 더하지 않고 탐색이 종료됩니다.



세그먼트 트리

- 시간 복잡도를 분석해 봅시다.
- 배열이 계속 절반이 되니 트리의 깊이는 $O(\log N)$ 입니다.

세그먼트 트리

- 시간 복잡도를 분석해 봅시다.
- 배열이 계속 절반이 되니 트리의 깊이는 $O(\log N)$ 입니다.
- 원소를 갱신할 때는, 가장 깊은 곳에서 위로 올라옵니다. 트리의 깊이만큼의 시간이 소요됩니다. 고로 갱신 연산은 $O(\log N)$ 의 시간 복잡도를 가집니다.

세그먼트 트리

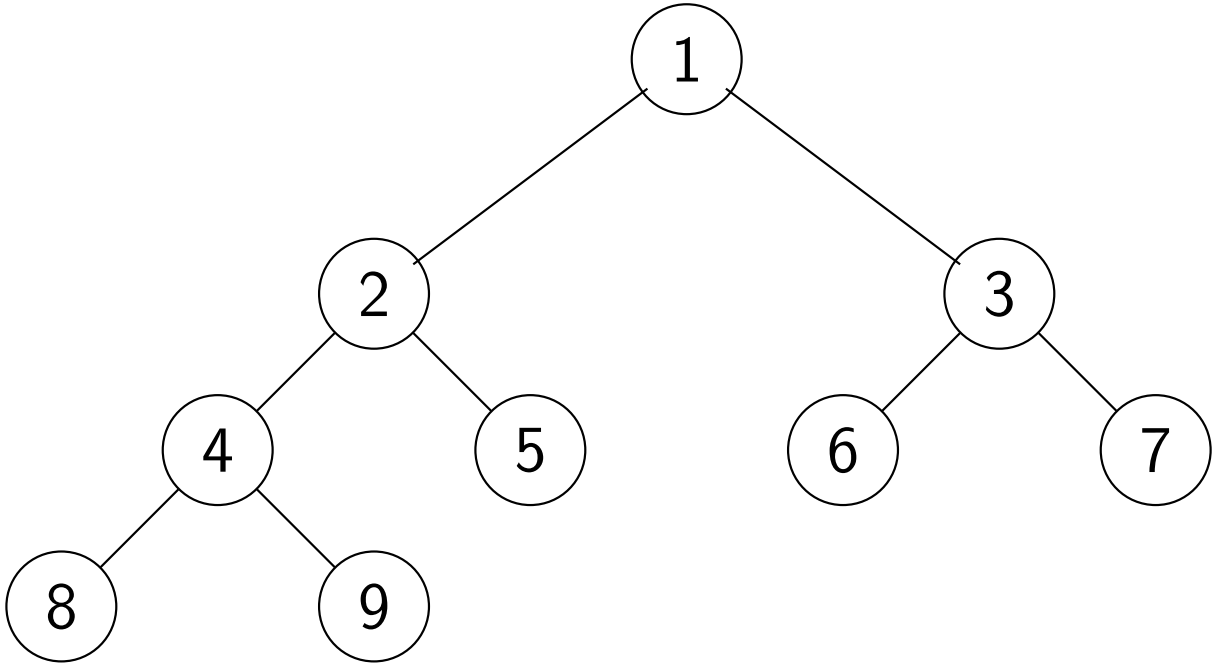
- 시간 복잡도를 분석해 봅시다.
- 배열이 계속 절반이 되니 트리의 깊이는 $O(\log N)$ 입니다.
- 원소를 갱신할 때는, 가장 깊은 곳에서 위로 올라옵니다. 트리의 깊이만큼의 시간이 소요됩니다. 고로 갱신 연산은 $O(\log N)$ 의 시간 복잡도를 가집니다.
- 구간 쿼리를 날릴 때는, 각 깊이에서 탐색하는 노드가 최대 4개입니다. (Why?) 탐색하는 노드는 $4 \times O(\log N)$ 개 이고, 역시 $O(\log N)$ 의 시간 복잡도를 가집니다.
- 모든 연산이 충분히 빠릅니다.

구현하기

- 세그먼트 트리는 배열을 사용해서 구현합니다.
- $N = 2^k$ 인 경우, 세그먼트 트리를 만드는 데 $2N$ 개의 노드가 필요합니다.
- 트리가 작아질 수록 필요한 노드는 줄어듭니다.
- 고로, N 이상의 최소 2의 지수승을 2^k 라고 했을 때, 2^{k+1} 크기의 노드를 할당하면 됩니다.
- 예를 들어, $N = 250000$ 이면, $N \leq 2^{18}$. 트리에 2^{19} 개의 노드가 필요합니다. 2^{19} 크기의 배열을 할당하면 됩니다.

구현하기

- 루트는 1번입니다.
- i번 노드의 자식은 $2i$, $2i+1$ 번 노드입니다.



구현하기

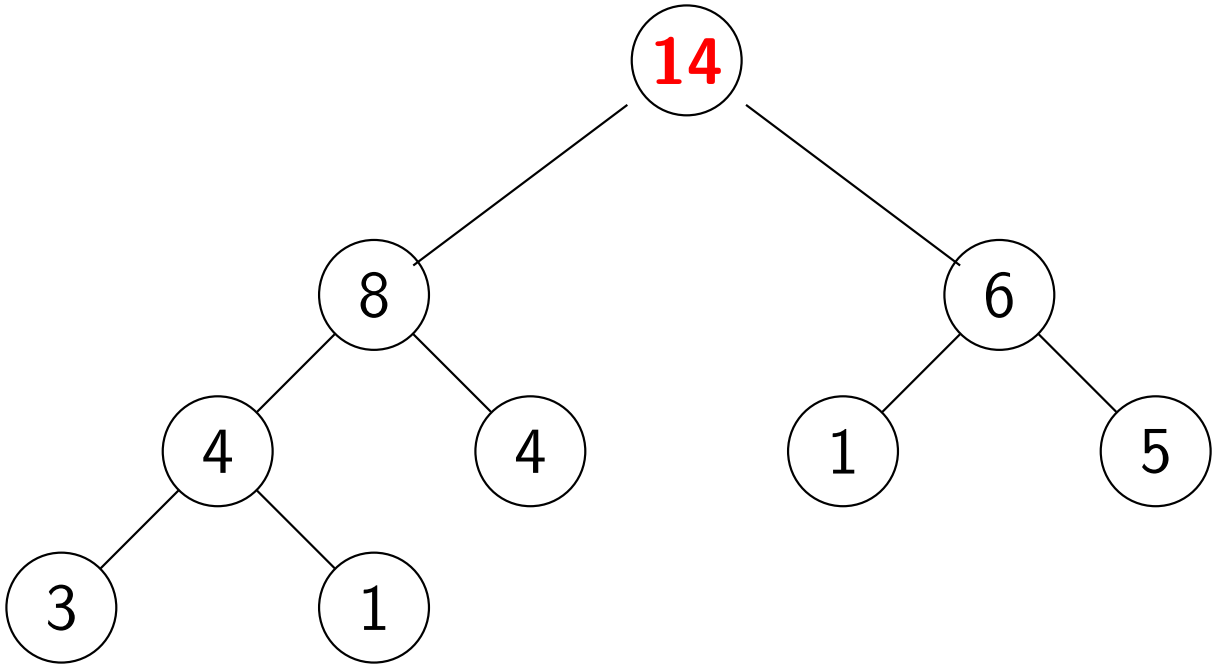
- 원소 갱신을 할 때, 루트에서 내려가면서 리프를 찾고, 다시 올라오면서 값을 갱신해 줍니다.
- 이진 탐색의 요령으로 리프를 재귀 함수로 찾아줍시다.
- 재귀 함수를 종료할 때, 자식의 변경 사항을 토대로 부모를 변경해 줍니다.

Code

```
void update(int pos, int s, int e, int p, int v){
    // A[pos] := v
    // We are at node p which represents interval [s, e].
    if(s == e){
        // update the leaf node.
        return;
    }
    int m = (s + e) / 2; // binary search
    if(pos <= m) update(pos, s, m, 2*p, v);
    else update(pos, m+1, e, 2*p+1, v);
    // update the node from the child node value.
}
```

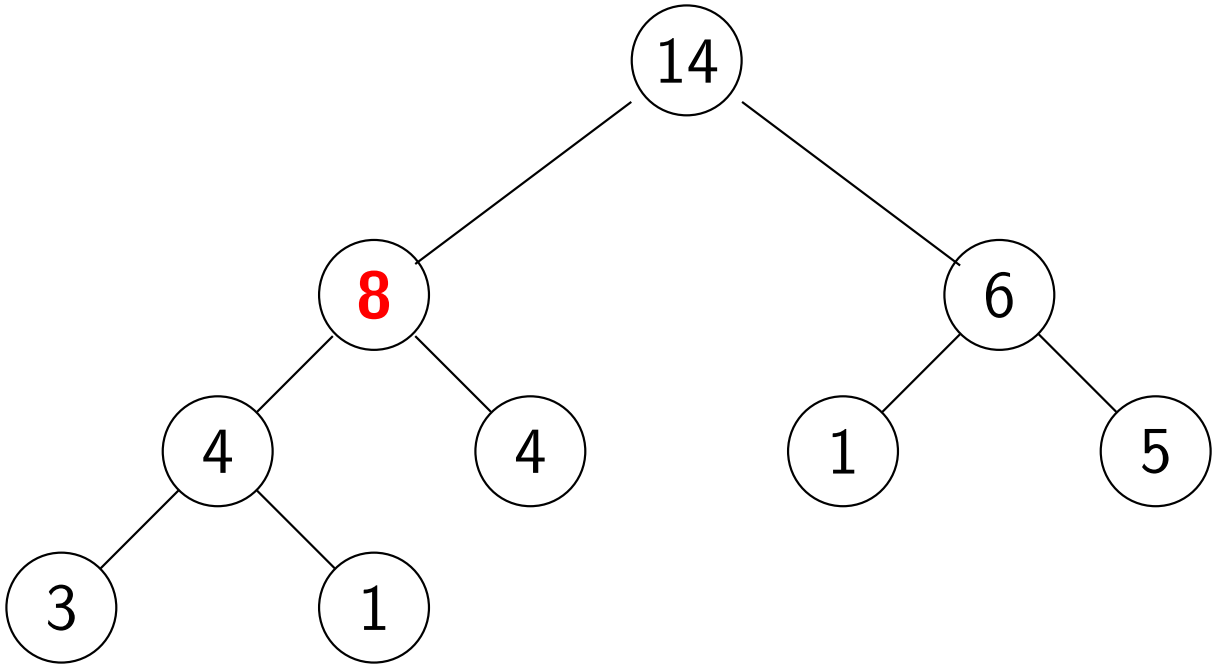
세그먼트 트리

- 2번 원소를 9로 바꿔보겠습니다.
- f([1, 5], 1): $[1, 5]$ 구간을 탐색하고 있고, 현재 노드 번호는 1번.



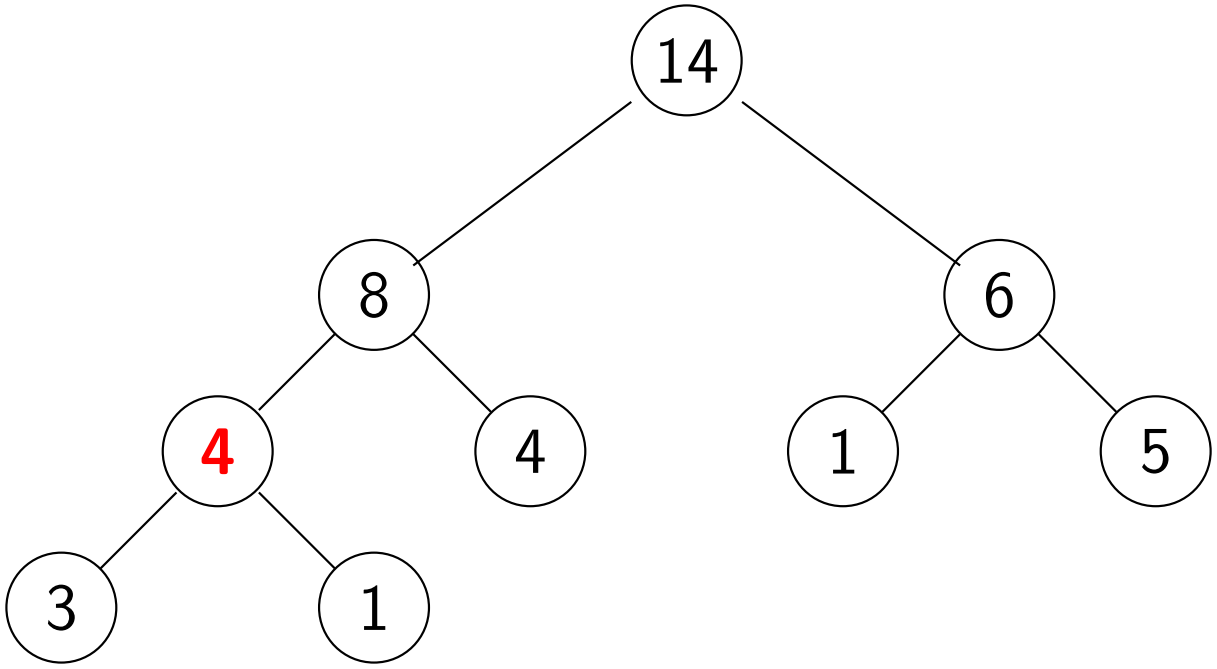
세그먼트 트리

- 2번 원소를 9로 바꿔보겠습니다.
- f([1, 3], 2): $[1, 3]$ 구간을 탐색하고 있고, 현재 노드 번호는 2번.



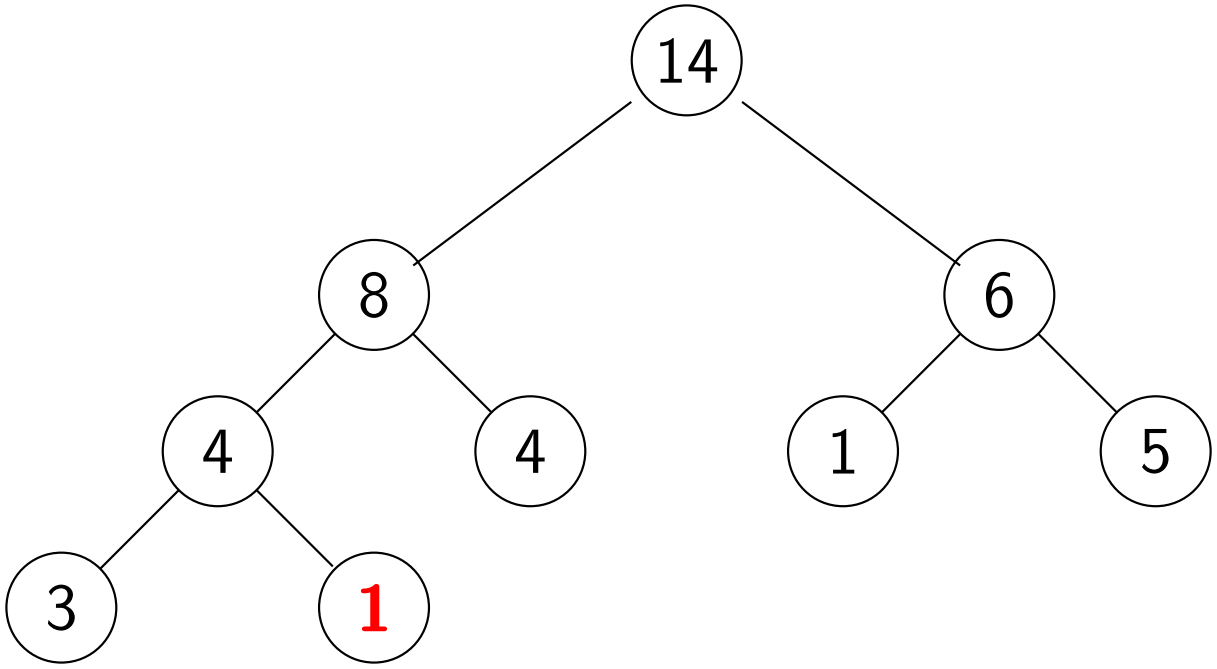
세그먼트 트리

- 2번 원소를 9로 바꿔보겠습니다.
- f([1, 2], 4): $[1, 2]$ 구간을 탐색하고 있고, 현재 노드 번호는 4번.



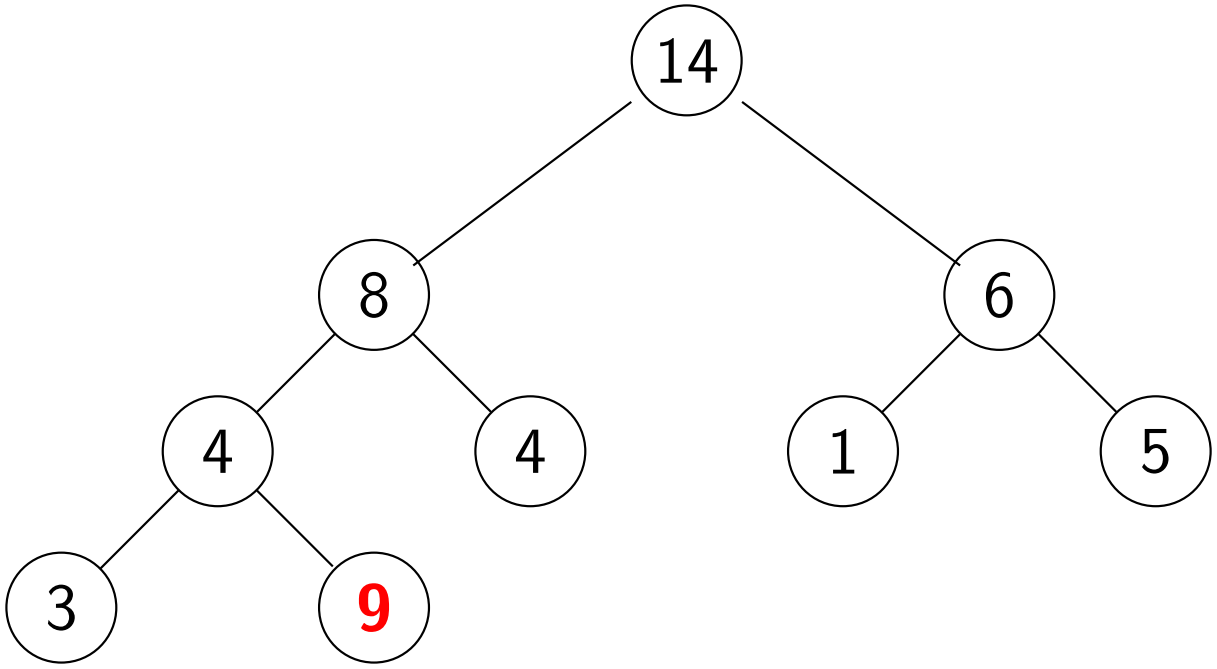
세그먼트 트리

- 2번 원소를 9로 바꿔보겠습니다.
- f([2, 2], 9): $[2, 2]$ 구간을 탐색하고 있고, 현재 노드 번호는 9번.



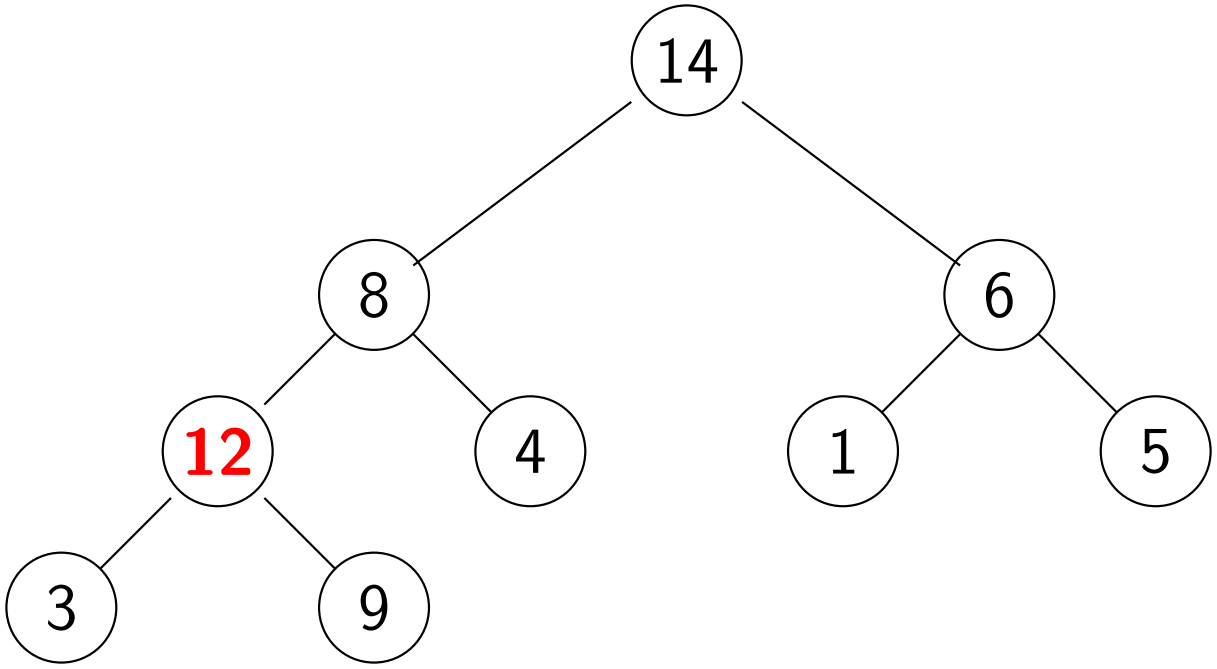
세그먼트 트리

- 2번 원소를 9로 바꿔보겠습니다.
- f([2, 2], 9): $[2, 2]$ 구간을 탐색하고 있고, 현재 노드 번호는 9번.



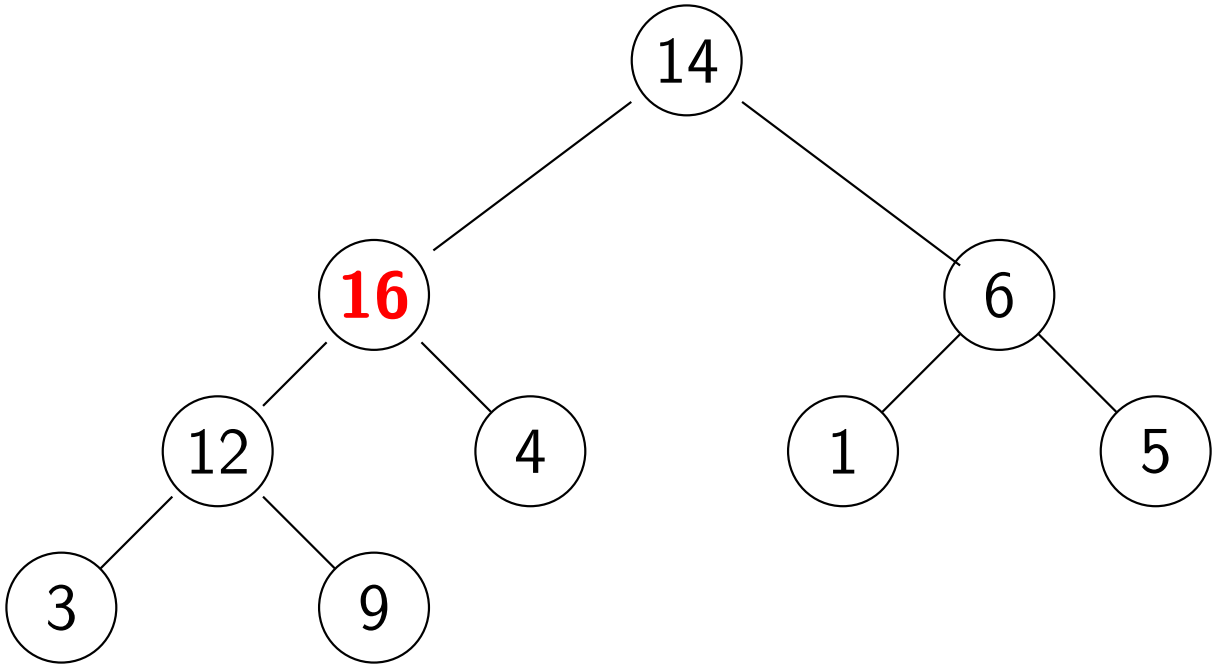
세그먼트 트리

- 2번 원소를 9로 바꿔보겠습니다.
- f([1, 2], 4): $[1, 2]$ 구간을 탐색하고 있고, 현재 노드 번호는 4번.



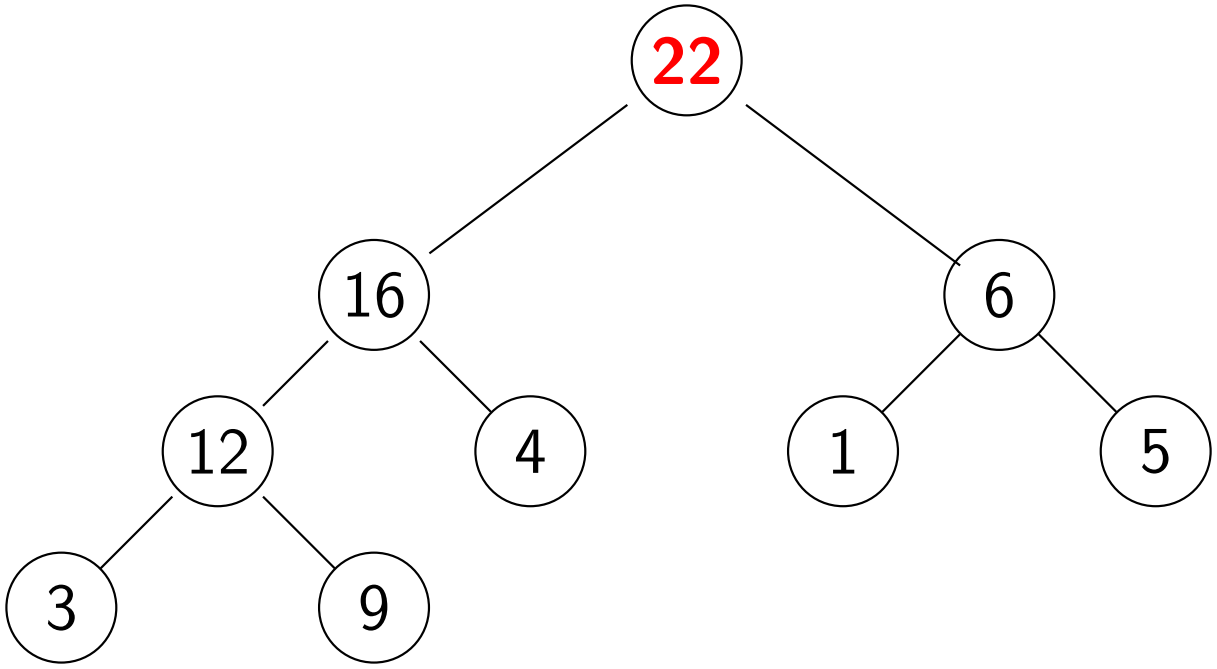
세그먼트 트리

- 2번 원소를 9로 바꿔보겠습니다.
- f([1, 3], 2): $[1, 3]$ 구간을 탐색하고 있고, 현재 노드 번호는 2번.



세그먼트 트리

- 2번 원소를 9로 바꿔보겠습니다.
- f([1, 5], 1): $[1, 5]$ 구간을 탐색하고 있고, 현재 노드 번호는 1번.



구현하기

- 구간 쿼리 역시 기본 원칙은 비슷합니다.
- 현재 보는 노드의 구간과 원하는 구간이 겹치지 않으면, 끊습니다.
- 현재 보는 노드의 구간이 원하는 구간 안에 속하면, 값을 전부 가져옵니다.
- 그렇지 않다면, 재귀적으로 내려갑니다.

Code

```
long long query(int s, int e, int ps, int pe, int p){
    // We want the result of interval [s, e]
    // We are at node p, which represents interval [ps, pe]
    if(e < ps || pe < s) return 0; // does not intersect
    if(s <= ps && pe <= e) return tree[p]; // fully
        contained
    int pm = (ps + pe) / 2;
    auto l = query(s, e, ps, pm, 2*p);
    auto r = query(s, e, pm+1, pe, 2*p+1);
    return l + r; // return the sum of both.
}
```