Indexed Tree

What is the Indexed Tree?

- 순서를 갖는 정보들이 주었을 때, 구간의 대표 값이나 연산 결과를 빠르게 얻을 수 있는 자료구조
- 사용 예 : 구간합, 구간내 최대값, 구간내 카운트
- 갱신연산 복잡도 : O(logN)
- 쿼리연산 복잡도 : O(logN)

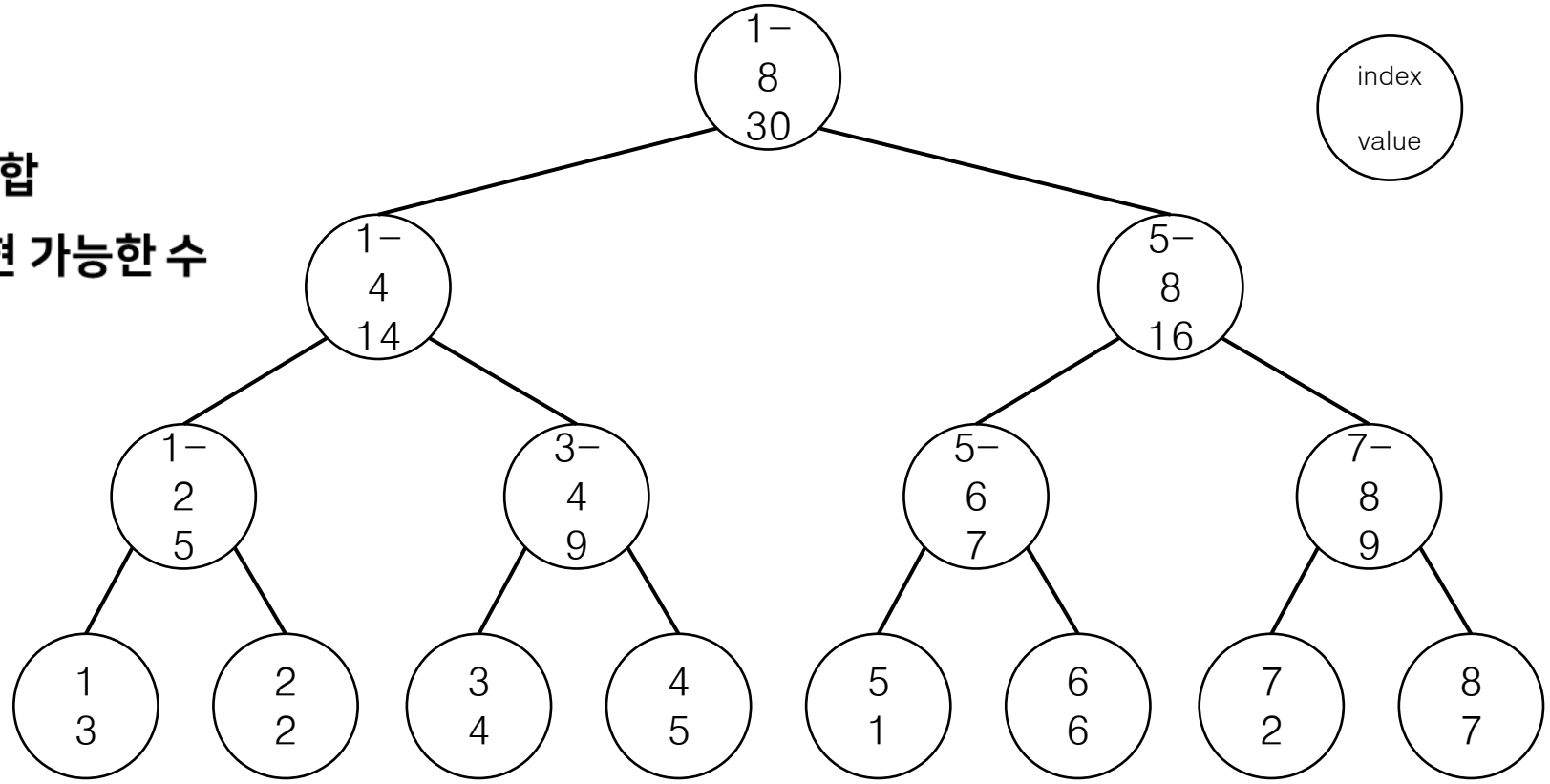
Contents

What is the indexed Tree?

- 포화 이진트리.
- 리프 노드 : 배열에 적혀 있는 수
- 내부 노드 : 왼쪽 자식과 오른쪽 자식의 합
- 리프 노드 개수(S) : N 이상의 2^n로 표현 가능한 수
- 깊이 : logS
- 총 노드 개수 : 2 * S – 1 개

구간합 예

	1	2	3	4	5	6	7	8
data	3	2	4	5	1	6	2	7

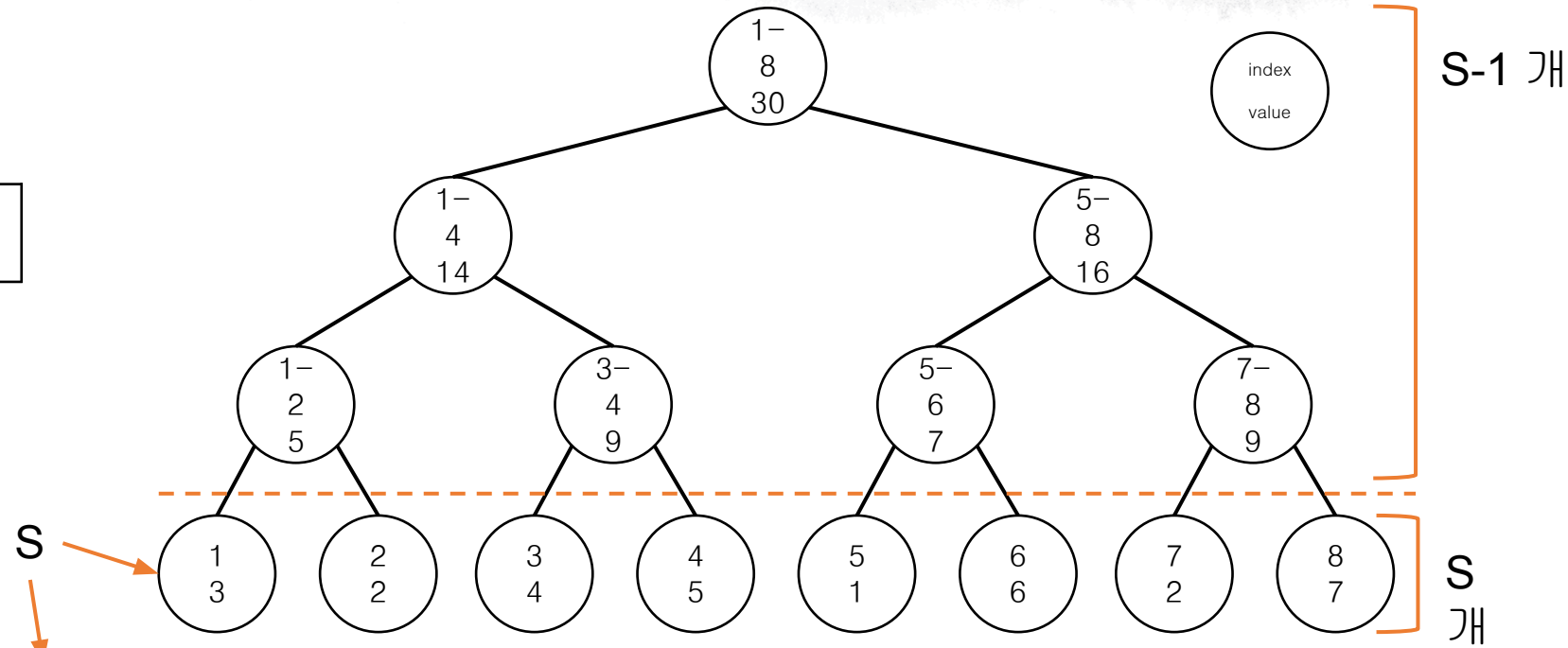


What is the indexed Tree?

- 배열로 나타낸 Indexed Tree

	1	2	3	4	5	6	7	8
data	3	2	4	5	1	6	2	7

N = 8
S = 2^3 = 8
총 노드 개수 = 15

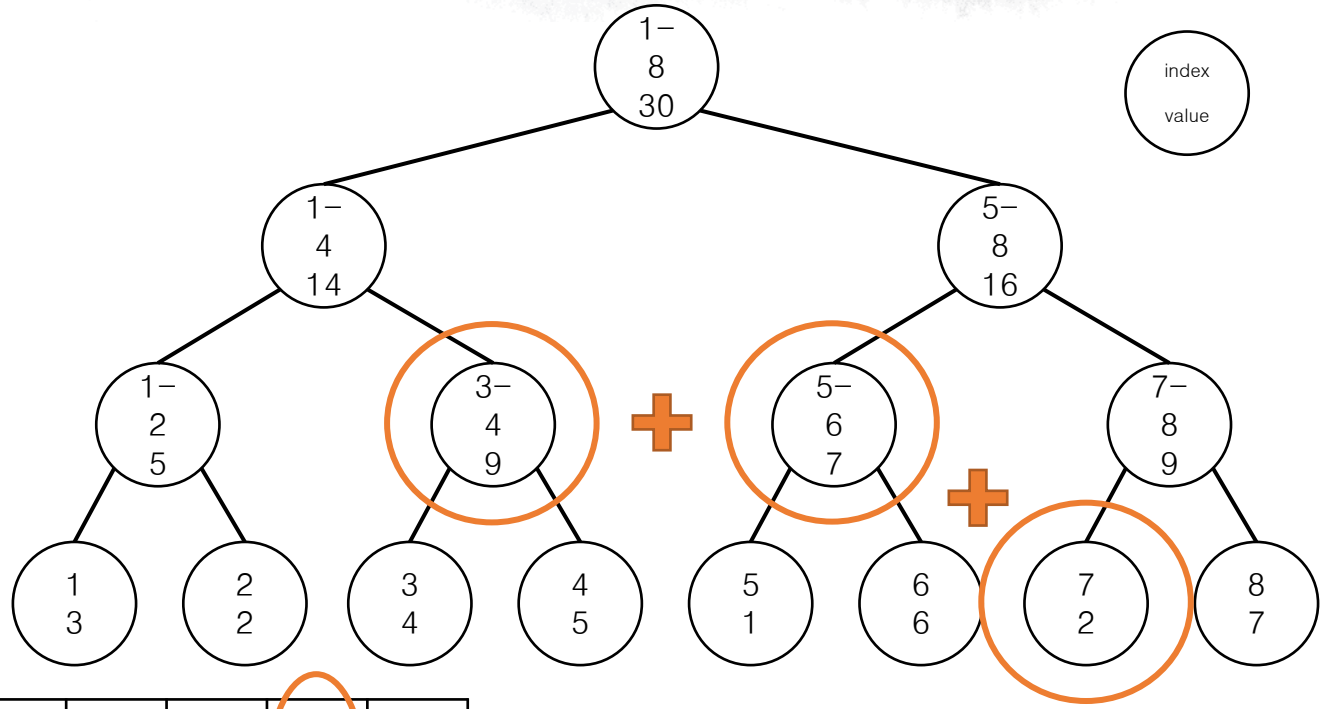


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
30	14	16	5	9	7	9	3	2	4	5	1	6	2	7

Query

- sum(3, 7)

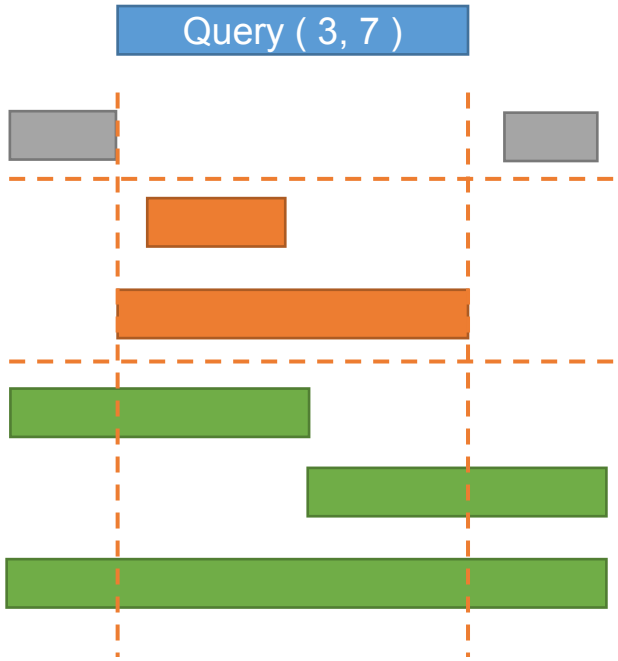
	1	2	3	4	5	6	7	8
data	3	2	4	5	1	6	2	7



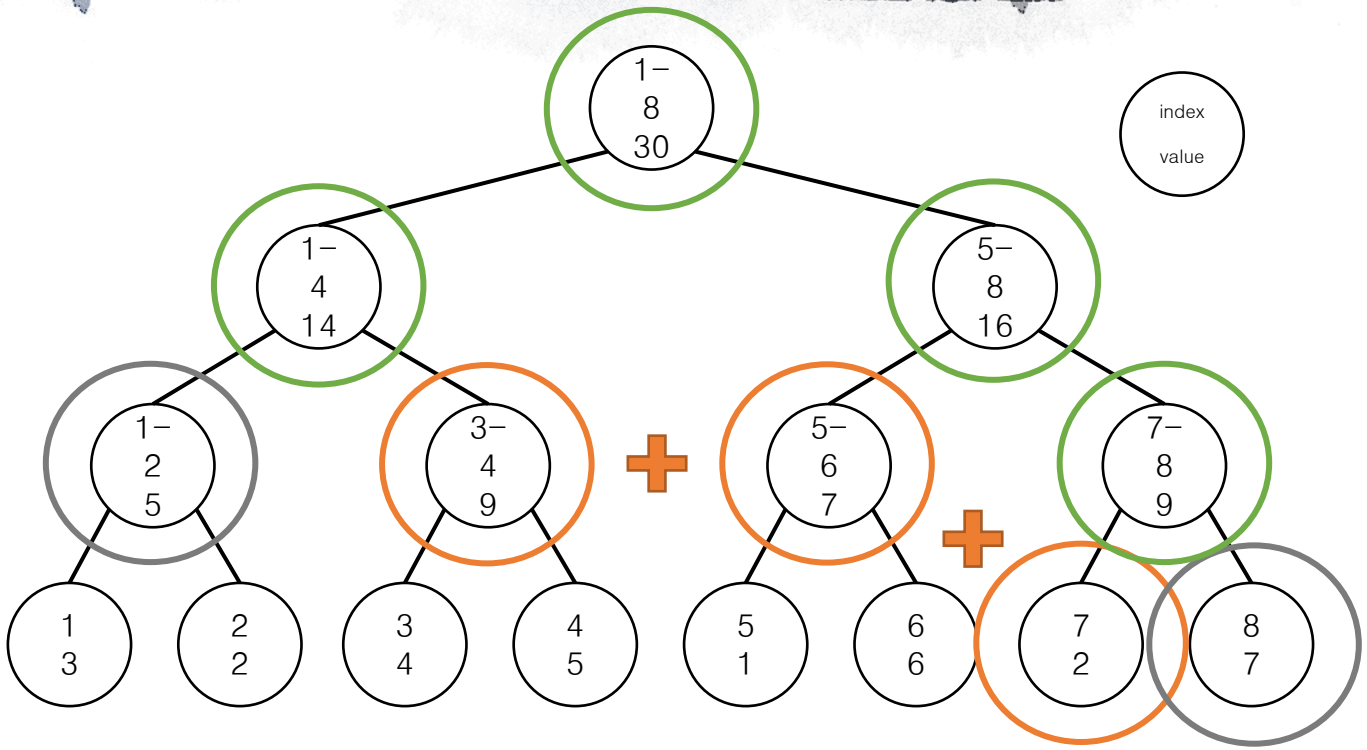
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
30	14	16	5	9	7	9	3	2	4	5	1	6	2	7

Query

- sum(3, 7)

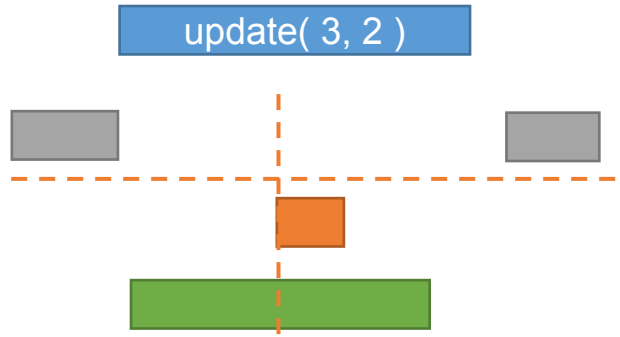


- 연관
- 값을
  가능
- 값 사용 불가
- 자식에게
  위임

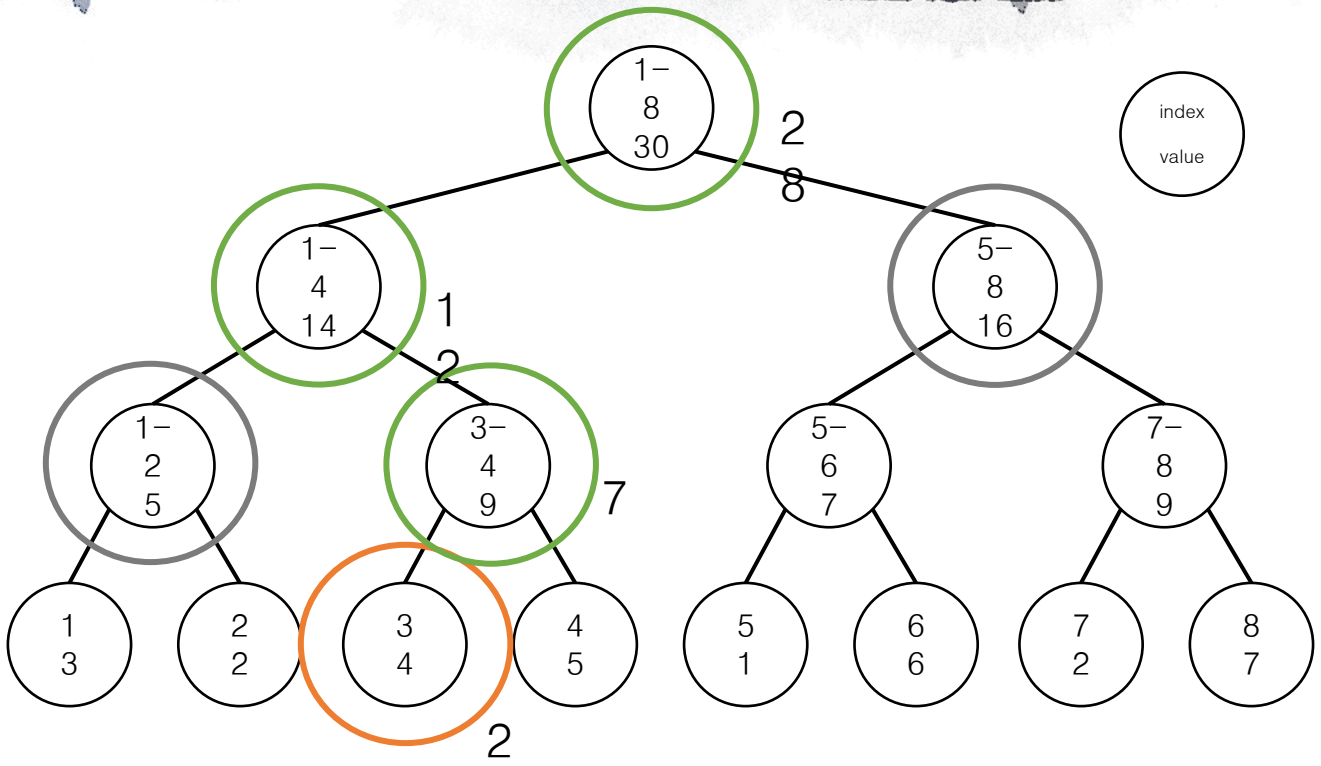


Update

- update(3, 2) : 3번째 값을 2로 갱신

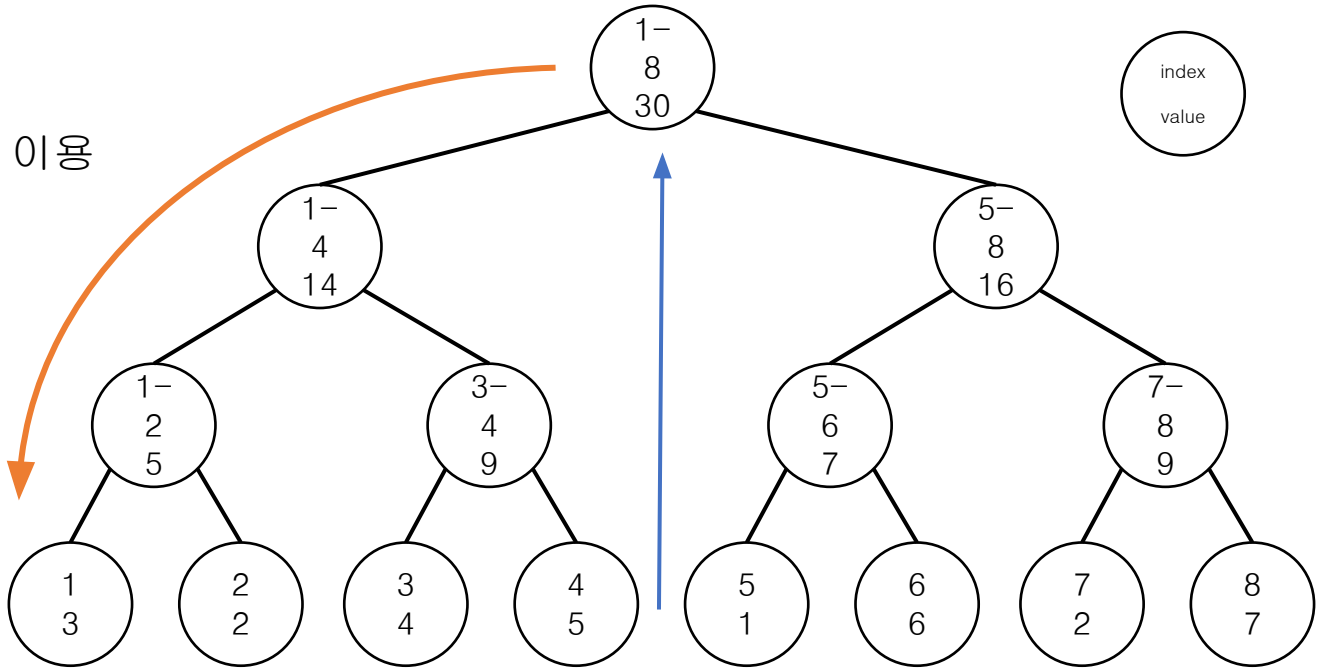


- 연관
- 없음
- 있음
- 갱신



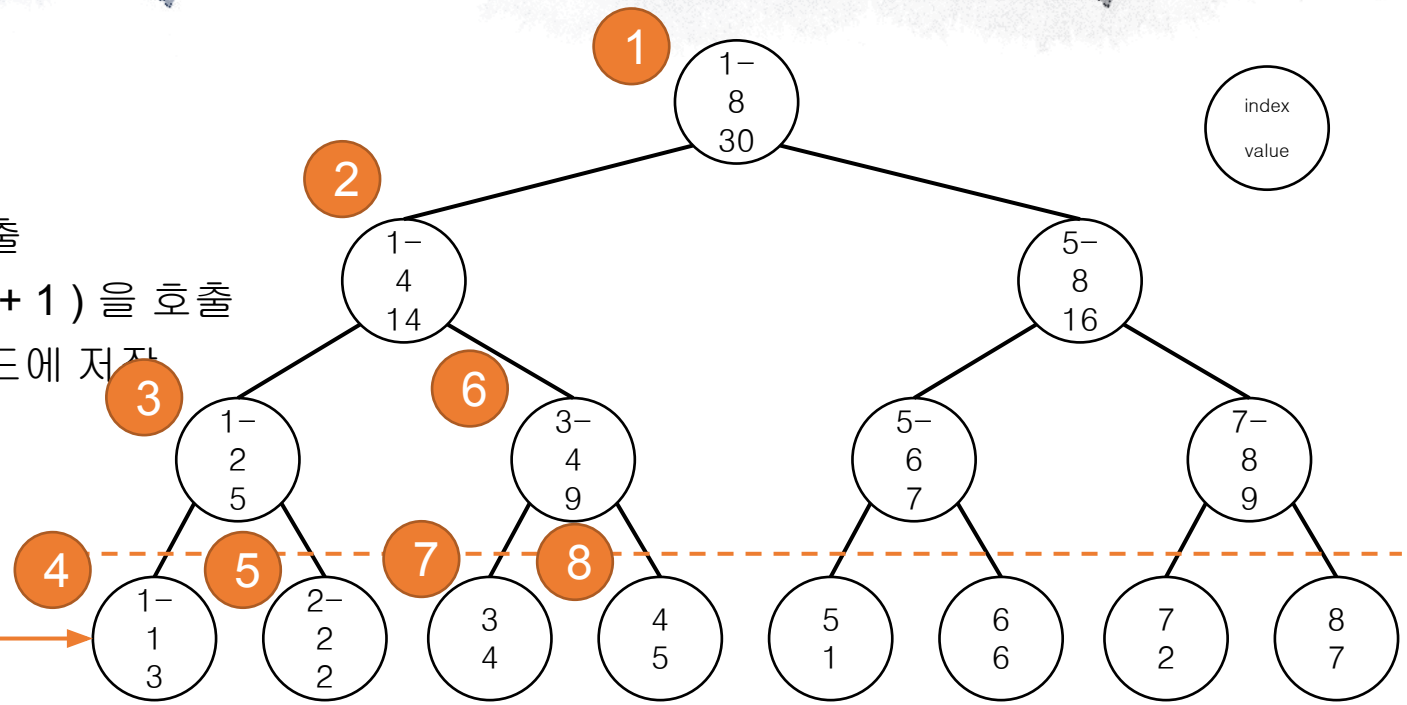
Top-Down Vs Bottom-Up

- Top-Down : DFS 기반 트리 탐색 (재귀 호출)
  - Indexed Tree 개념을 그대로 코드로 수행
  - 사람이 손으로 하는 방식과 유사
  - 왼쪽 자식 = 2 * node, 오른쪽 자식 = 2 * node + 1 이용
  - 가지치기 가능함
  - x번째로 빠른 숫자 등 카운팅 쿼리 가능
- Bottom-Up : 반복문 기반 이동
  - Index의 홀짝 특성을 이용
  - 부모 = node / 2 이용
  - 코드가 단순
  - 수행 속도가 미세하게 빠름



Top-Down
Init(left, right, node)

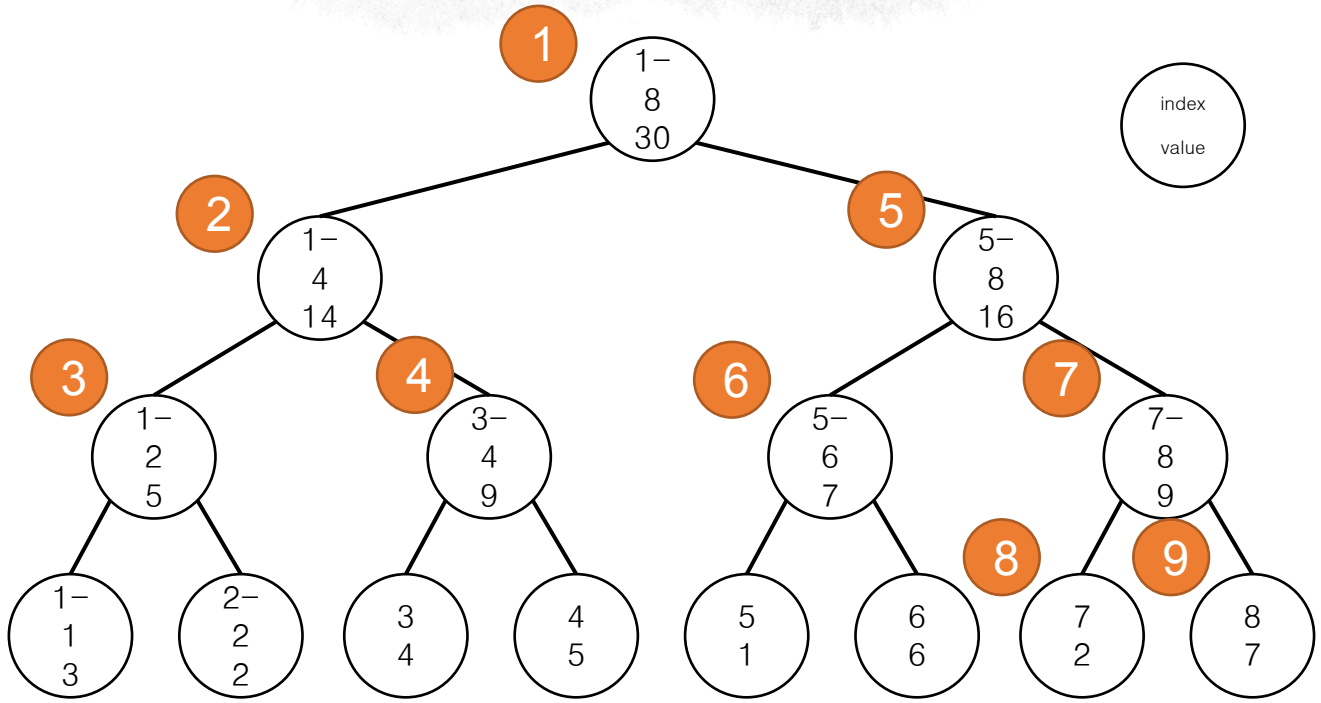
- Root 부터 시작 init(1, 8, 1)
- 내부노드 일 경우 (left != right)
  - 왼쪽 자식 init(left, mid, node * 2) 을 호출
  - 오른쪽 자식 init(mid + 1, right, node * 2 + 1) 을 호출
  - 왼쪽 자식 + 오른쪽 자식 값을 합쳐서 노드에 저장
  - 노드의 값을 리턴
- 리프노드 일 경우 (left == right)
  - 노드에 배열의 값 저장
  - 노드의 값을 리턴



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
30	14	16	5	9	7	9	3	2	4	5	1	6	2	7

Top-Down
Query(left, right, node, queryLeft, queryRight)

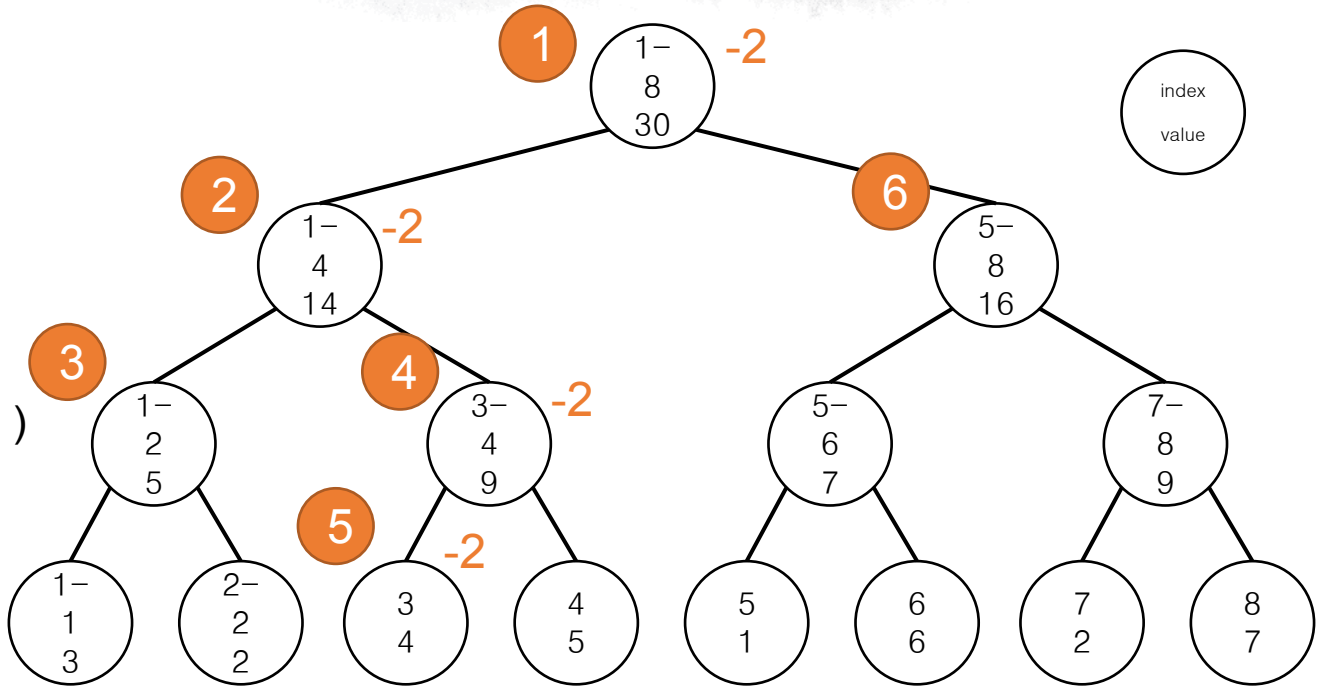
- Root 부터 시작 query(1, 8, 1, 3, 7)
- 노드가 Query 범위 밖 - 연관 없음
  - 무시
- 노드가 Query 범위 안에 들어옴 - 판단 가능
  - 현재 노드값 리턴
- 노드가 Query 범위 에 걸쳐있음 – 판단 불가
  - 왼쪽 query(l, mid, node * 2 , 3, 7) 을 호출
  - 오른쪽 query(mid + 1, r, node * 2 + 1. 3, 7) 을 호출
  - 왼쪽 query + 오른쪽 query 값을 합쳐서 리턴



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
30	14	16	5	9	7	9	3	2	4	5	1	6	2	7

Top-Down
Update(left, right, node, target, diff)

- Root 부터 시작 update(1, 8, 1, 3, -2)
- 노드가 Target 미포함 - 연관 없음
  - 무시
- 노드가 Target 포함
  - 현재 노드에 diff 반영
  - 자식이 있을 경우 왼쪽 update(l, mid, node * 2 , 3, -2)
  - 오른쪽 update(mid + 1, r, node * 2 + 1, 3, -2)

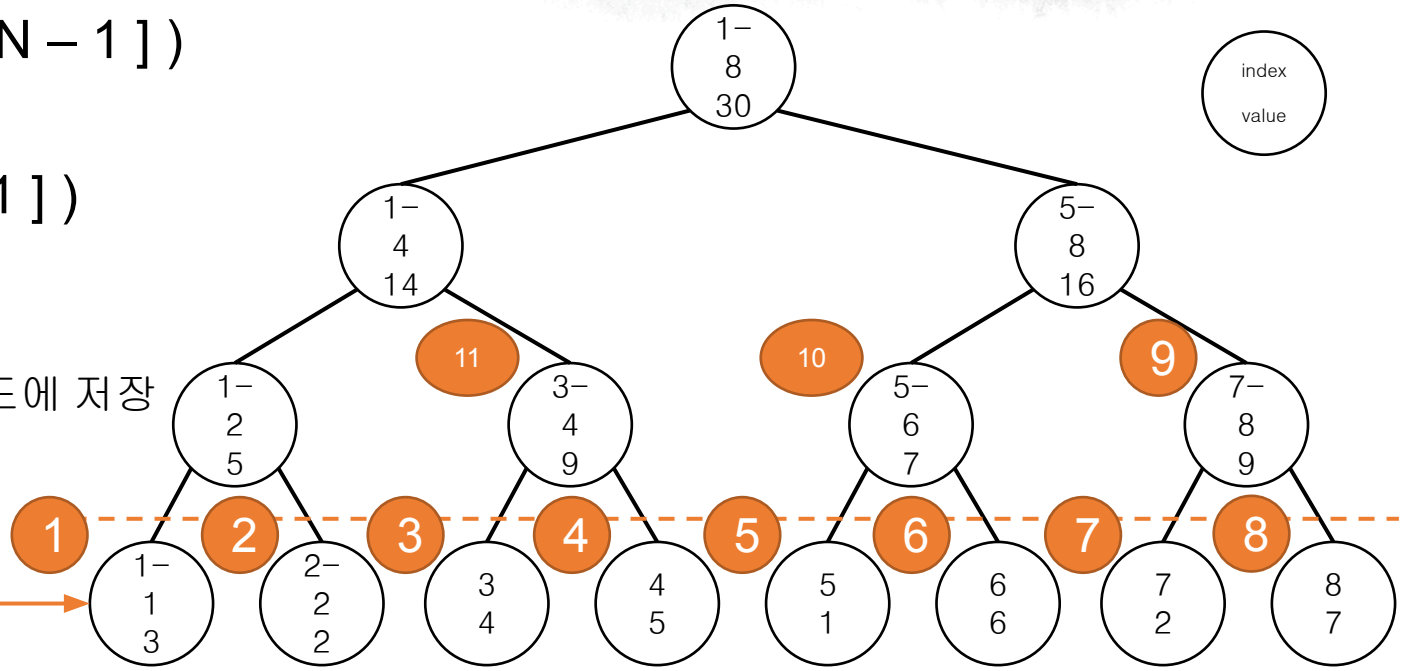


-2 -2 -2 -2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
30	14	16	5	9	7	9	3	2	4	5	1	6	2	7

Bottom-Up
Init

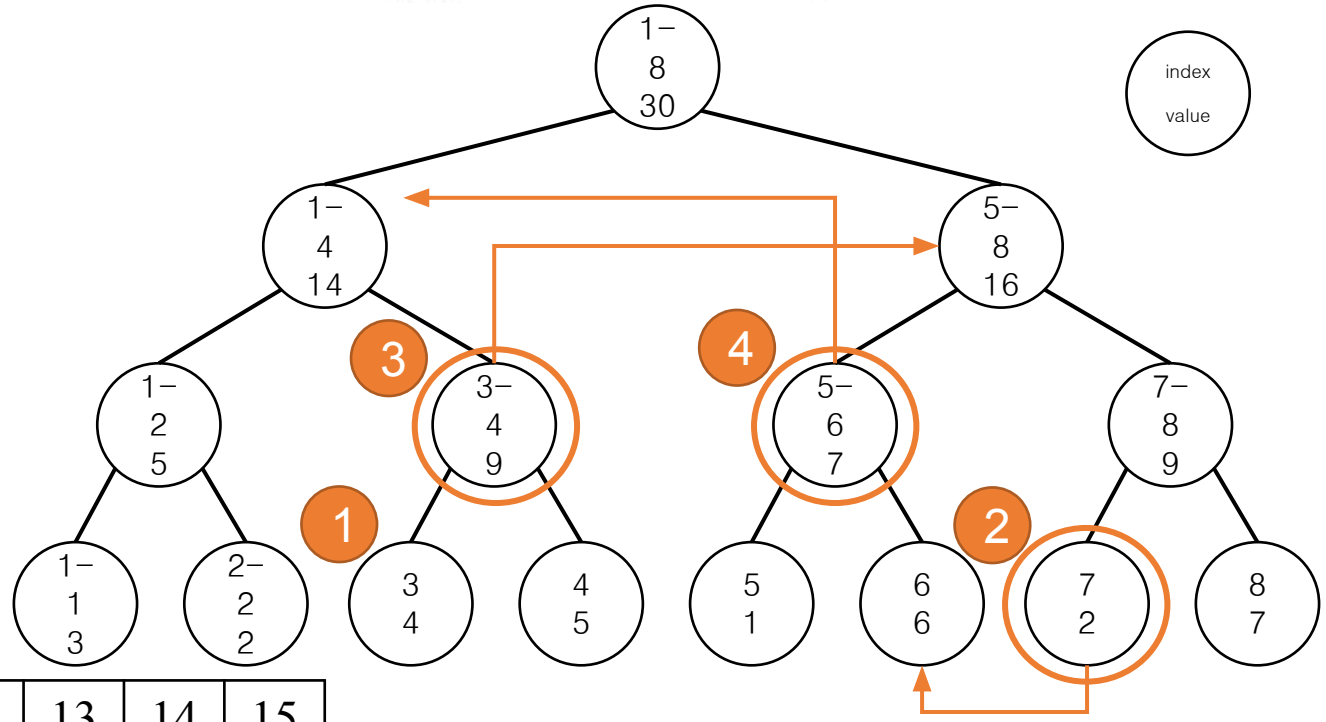
- 리프노드 순회 (index = [S] ~ [S + N – 1])
  - 노드에 배열의 값 저장
- 내부노드 순회 (index = [S – 1] ~ [1])
  - 왼쪽 자식 index * 2
  - 오른쪽 자식 index * 2 + 1
  - 왼쪽 자식 + 오른쪽 자식 값을 합쳐서 노드에 저장



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
30	14	16	5	9	7	9	3	2	4	5	1	6	2	7

Bottom-Up
Query(queryLeft, queryRight)

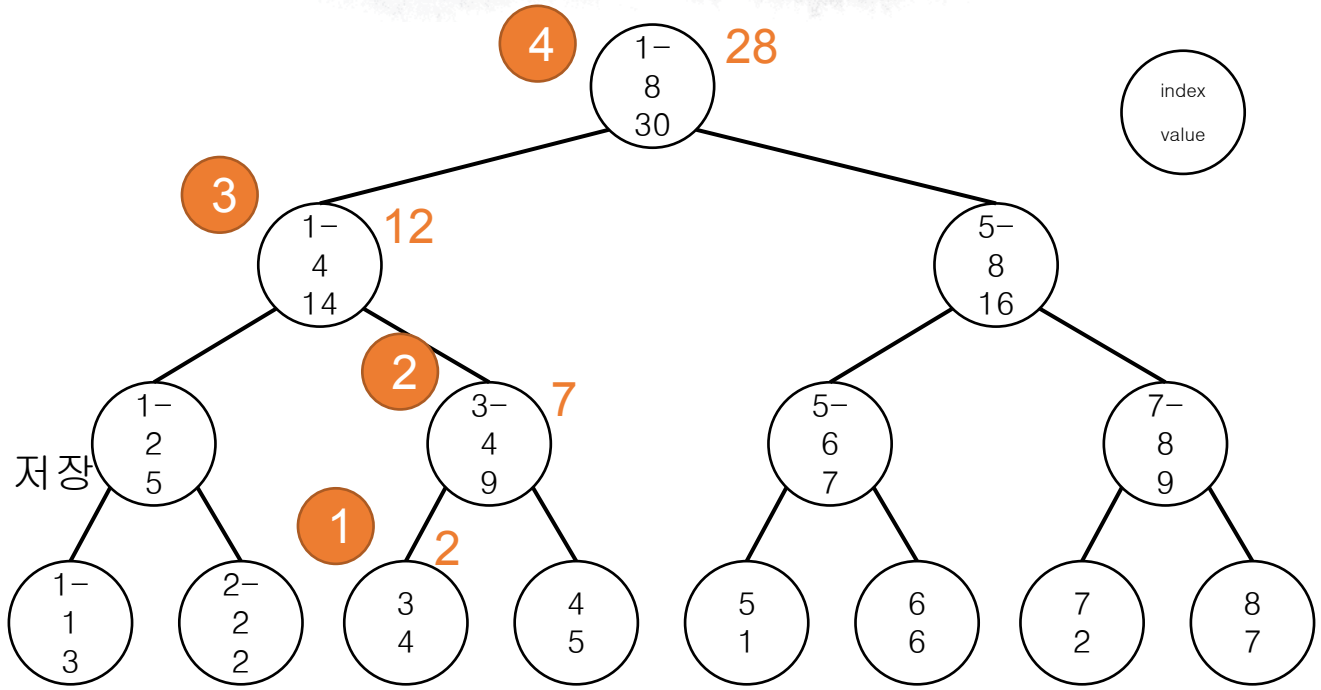
- 리프 노드부터 시작 query(3, 7)
  - nodeLeft = S + queryLeft – 1
  - nodeRight = S + queryRight – 1
- while(nodeLeft <= nodeRight)
- leftNode 분기 조건
  - 짝수 : 부모 값 사용 가능 => leftNode = leftNode / 2
  - 홀수 : 현재 노드 값 추가 => leftNode = (leftNode + 1) / 2
- rightNode 분기 조건
  - 짝수 : 현재 노드 값 추가 => rightNode = (rightNode – 1) / 2
  - 홀수 : 부모 값 사용 가능 => rightNode = rightNode / 2



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
30	14	16	5	9	7	9	3	2	4	5	1	6	2	7

Bottom-Up
Update(target, value)

- 리프 부터 시작 update(3, 2)
- node = S + target – 1
- 노드를 해당 값으로 갱신 1
- 부모로 이동 node /= 2
- while (node >= 1) 2 ~ 4
  - 좌측 (node * 2) 과 우측 (node * 2 + 1) 합을 노드에 저장
  - 부모로 이동 node /= 2



28 12 7 2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
30	14	16	5	9	7	9	3	2	4	5	1	6	2	7