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

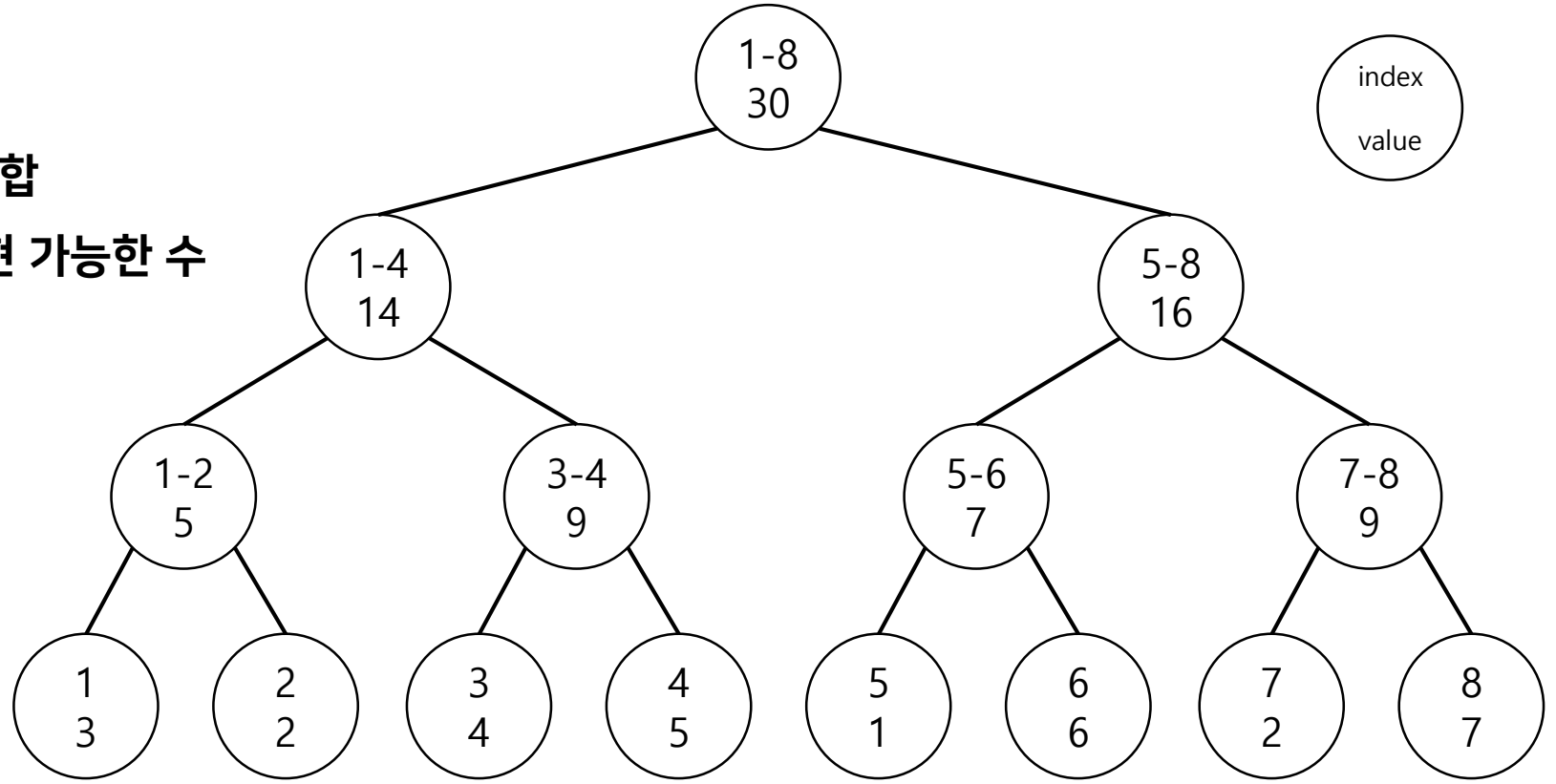
1. Indexed Tree?
2. Query
3. Update
4. Top-Down vs Bottom-Up
5. Practices

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**
- **높이 : 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

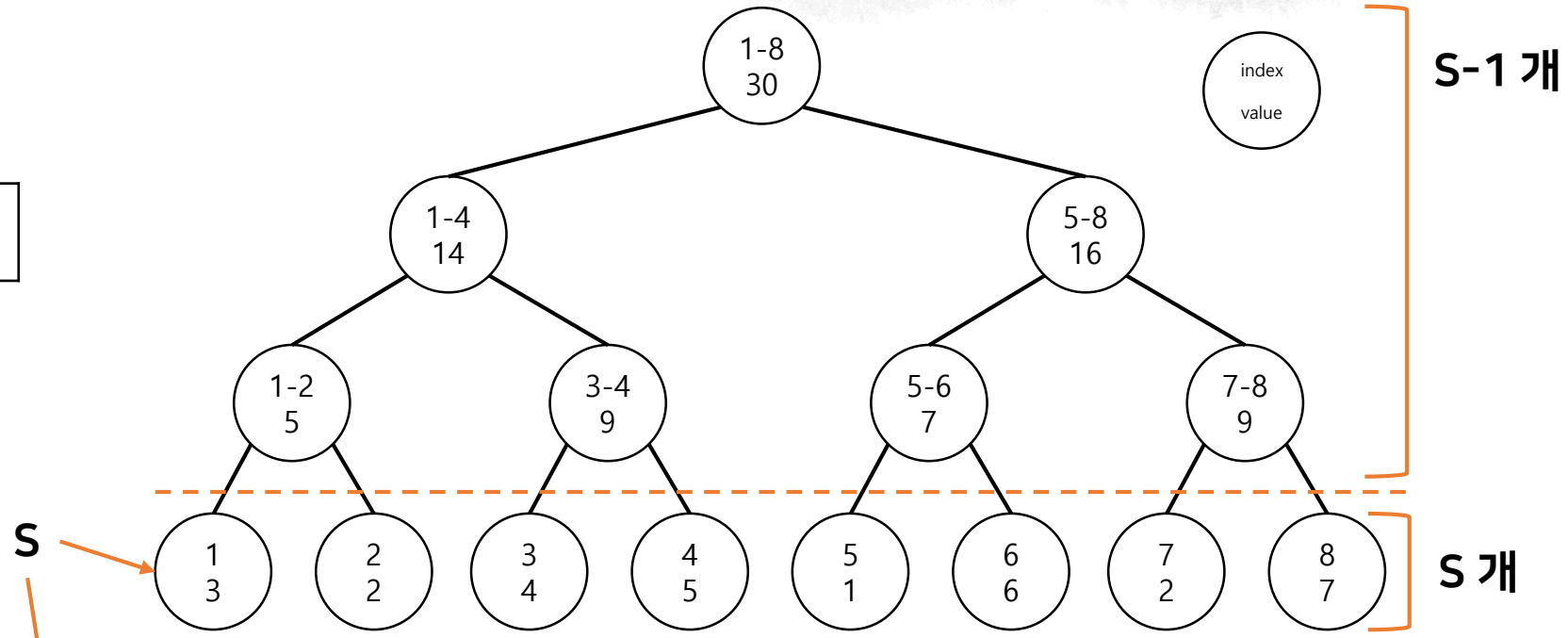


What is the indexed Tree?

- **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

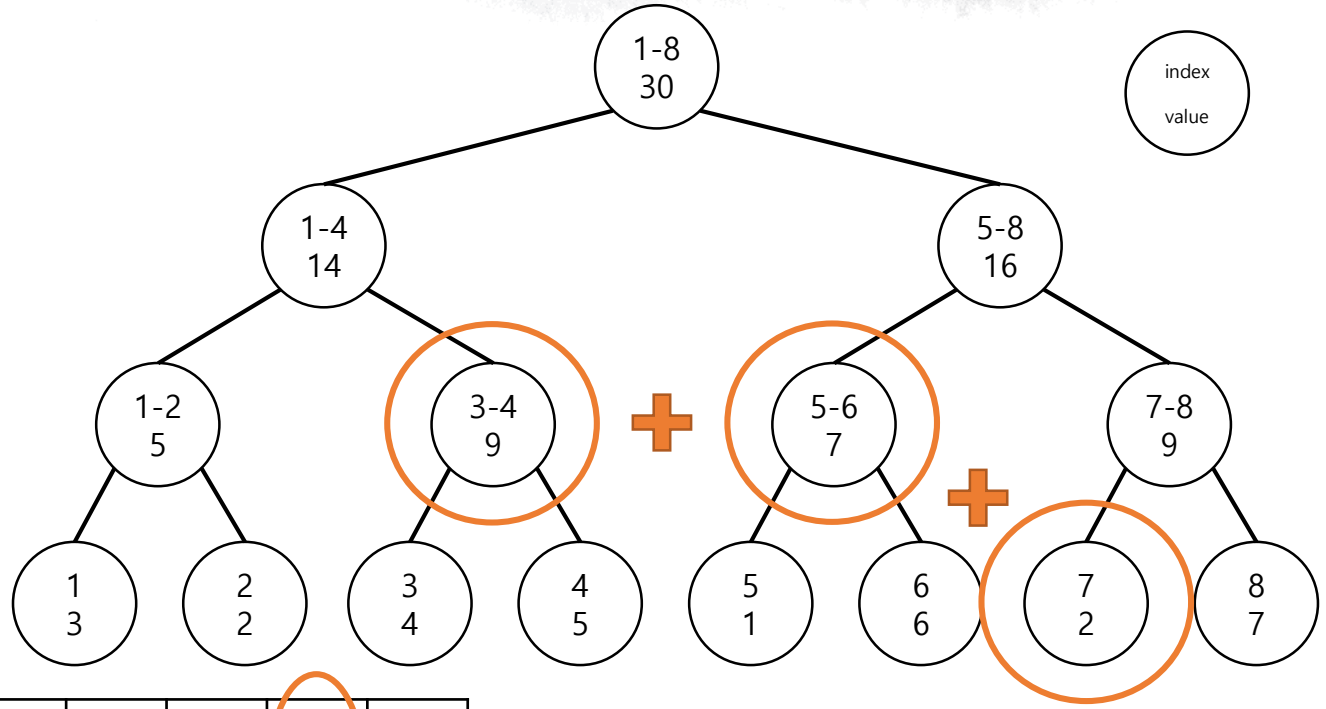


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
30	14	16	5	9	7	9	3	2	4	5	1	6	2	7

Query

- **sum(3, 7)**

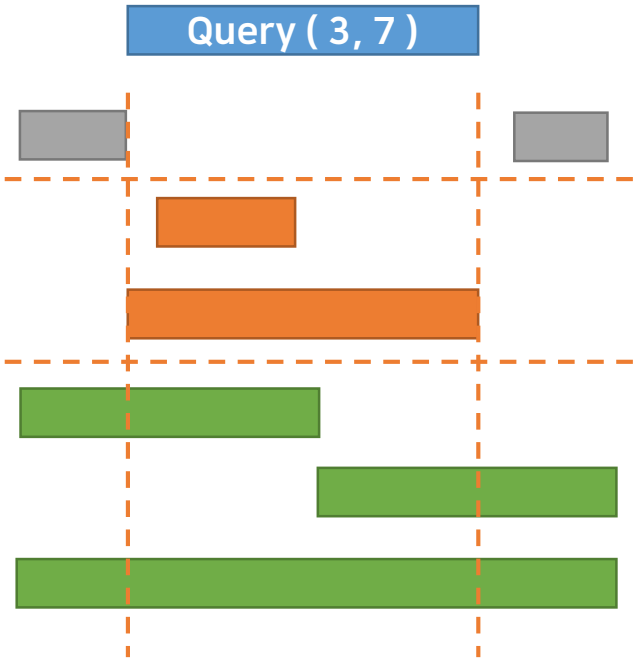
	1	2	3	4	5	6	7	8
data	3	2	4	5	1	6	2	7



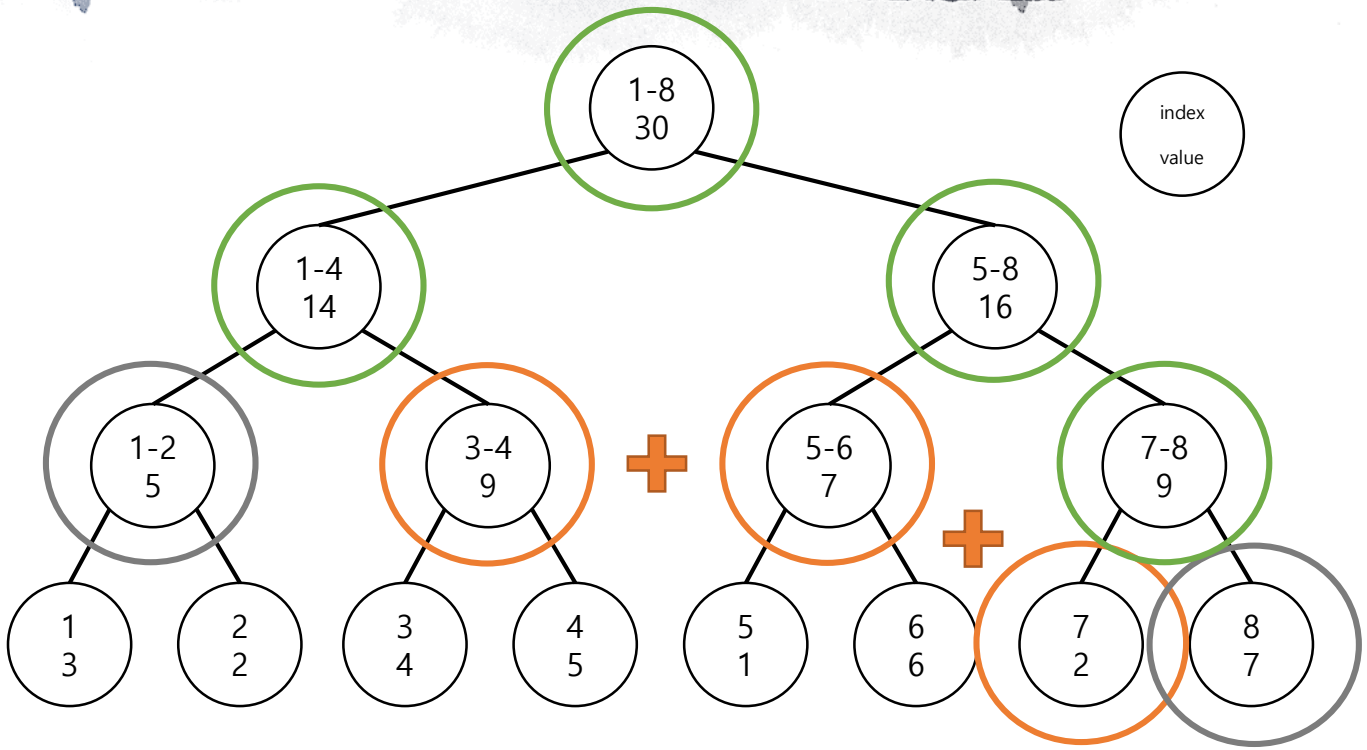
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
30	14	16	5	9	7	9	3	2	4	5	1	6	2	7

Query

- sum(3, 7)

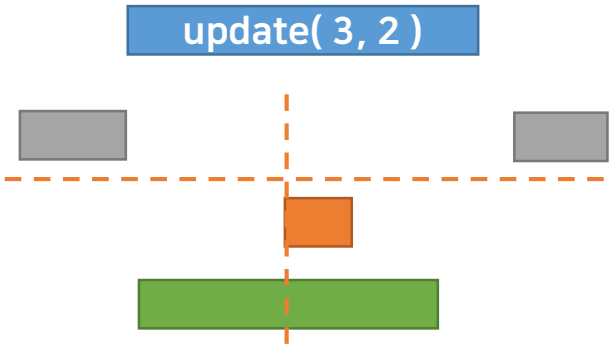


- 연관 없음
- 판단 가능
- 값 사용
- 판단 불가
- 값 사용 불가
- 자식에게 위임

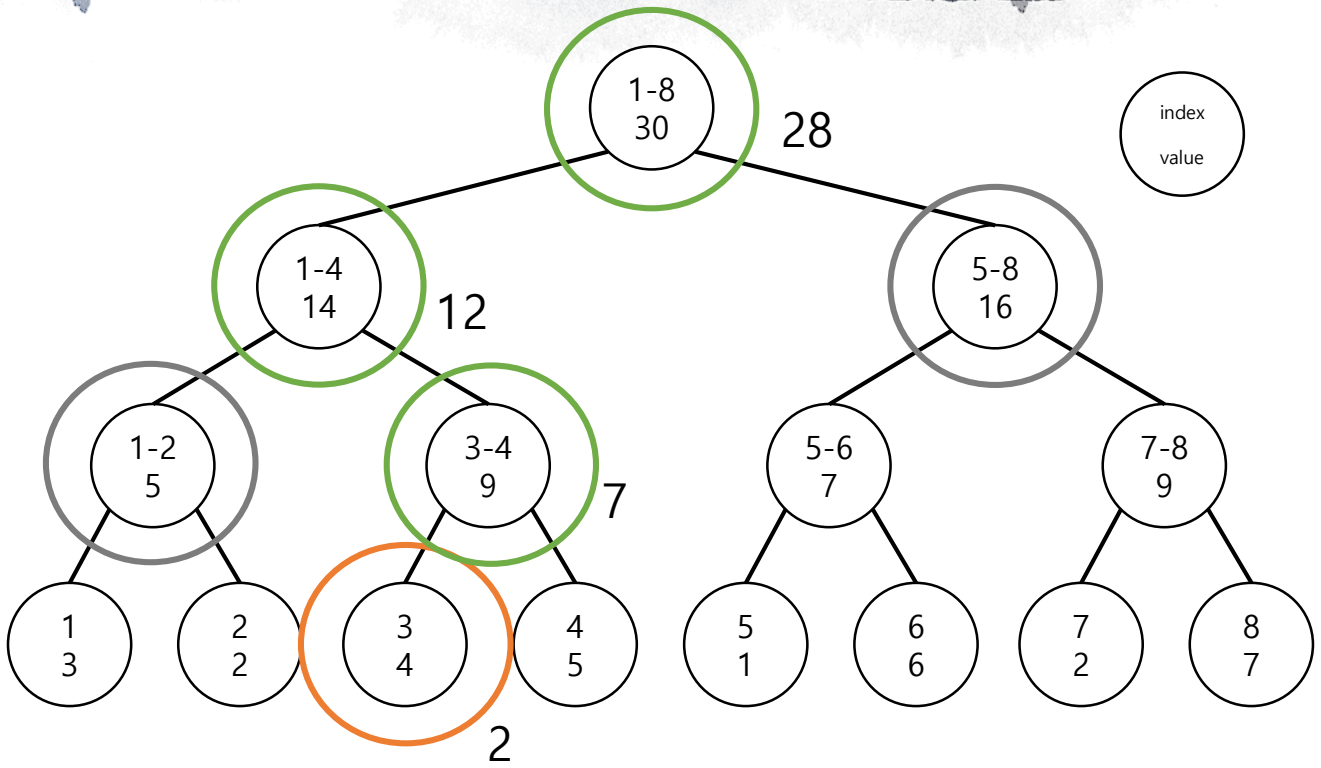


Update

- update(3, 2) : 3번째 값을 2로 갱신

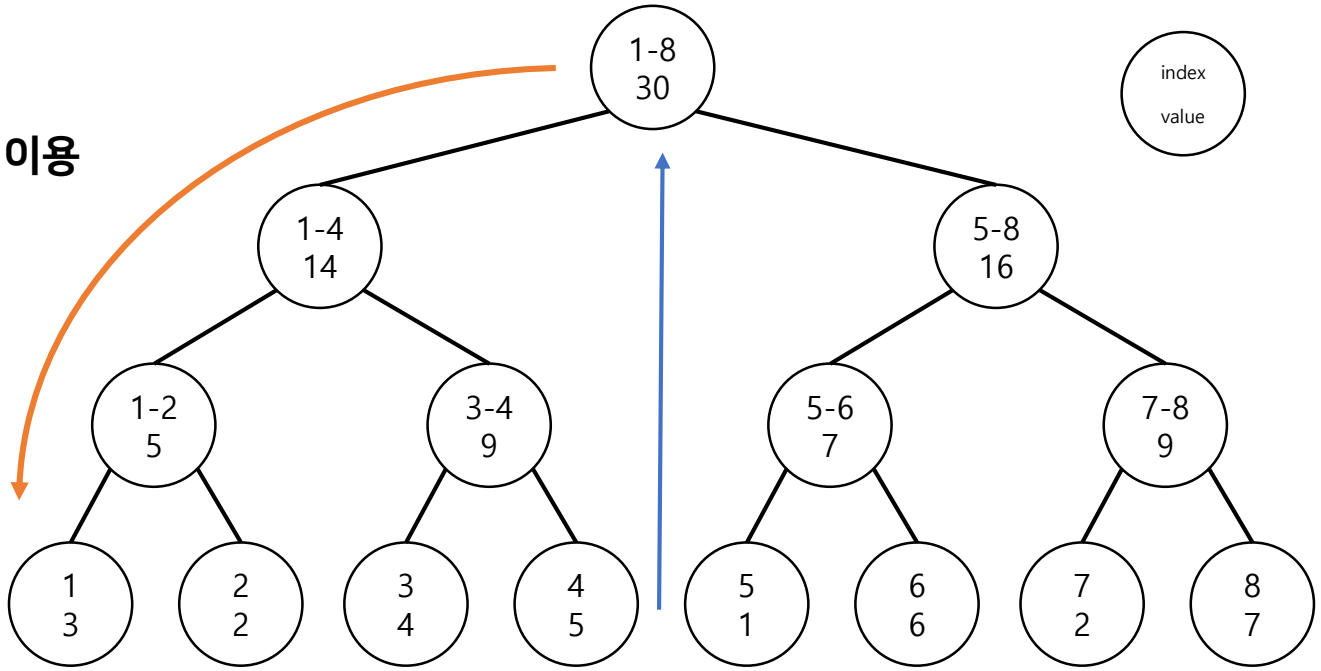


- 연관 없음
- 연관 있음
- 갱신



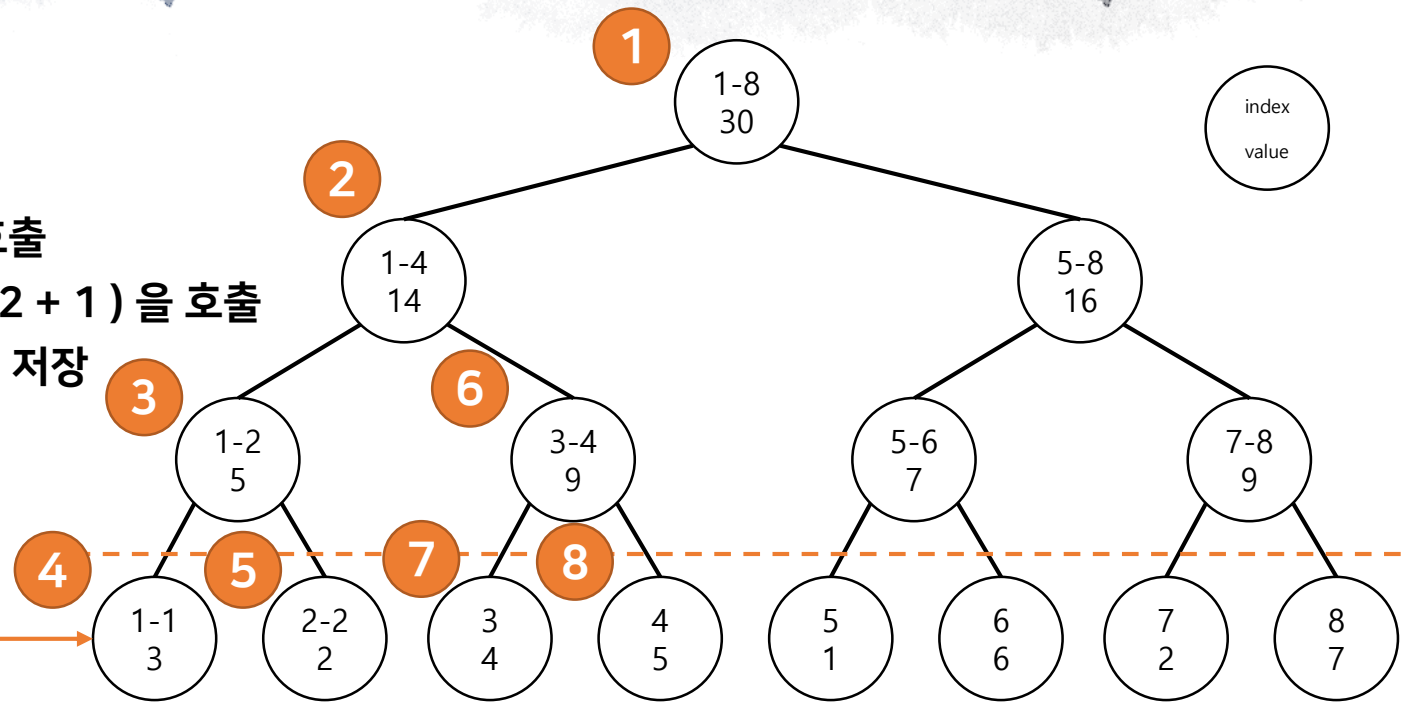
Top-Down Vs Bottom-Up

- **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

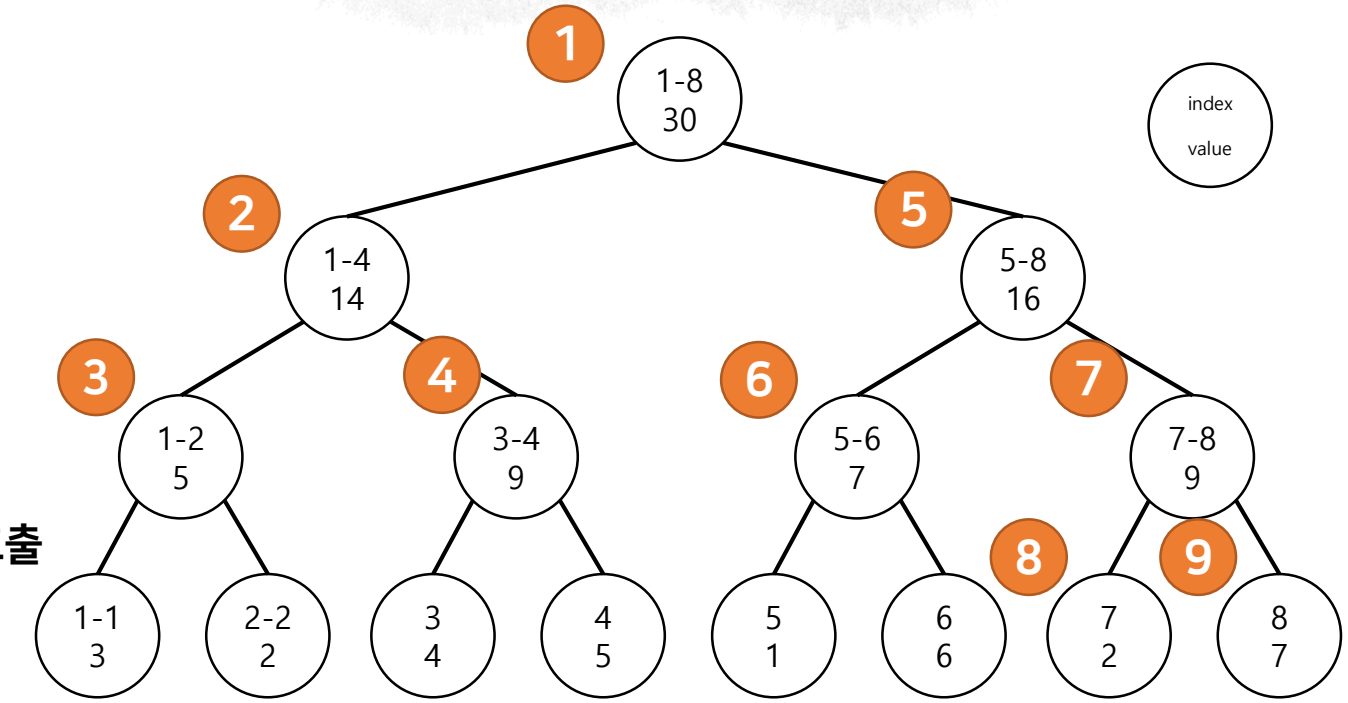


S

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)

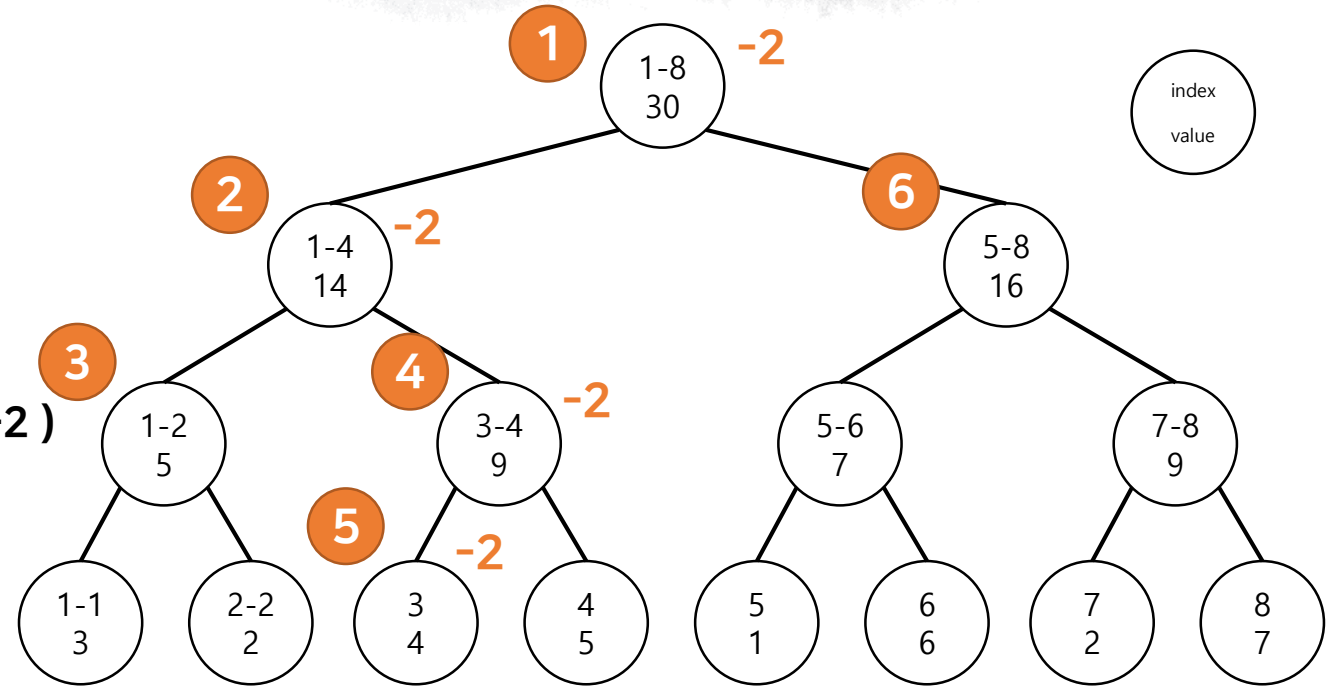
- Root 부터 시작 query(1, 8, 1, 3, 7)
- 노드가 Query 범위 밖 - 연관 없음
  - 무시
- 노드가 Query 범위 안에 들어옴 - 판단 가능
  - 현재 노드값 리턴
- 노드가 Query 범위 에 걸쳐있음 – 판단 불가
  - 왼쪽 query(l, mid, node * 2 , 3, 7) 을 호출
  - 오른쪽 query(mid + 1, r, node * 2 + 1. 3, 7) 을 호출
  - 왼쪽 query + 오른쪽 query 값을 합쳐서 리턴



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)

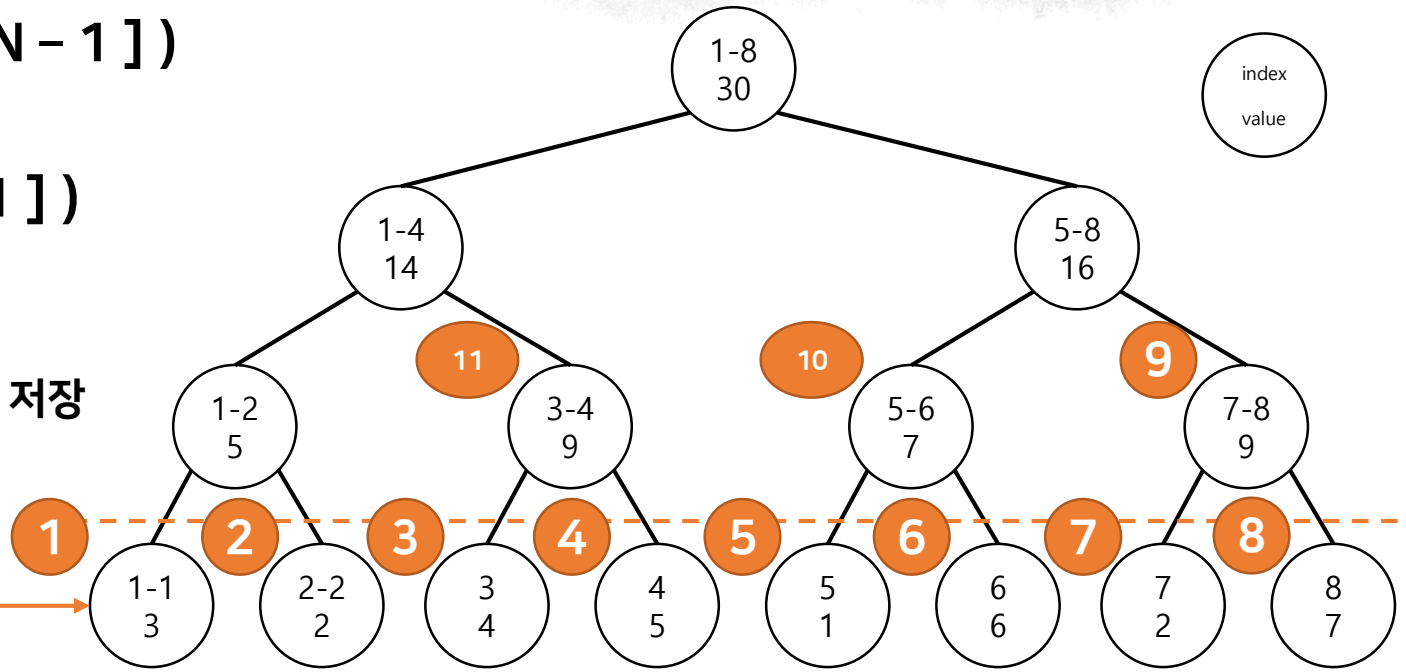


-2 -2 -2 -2

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

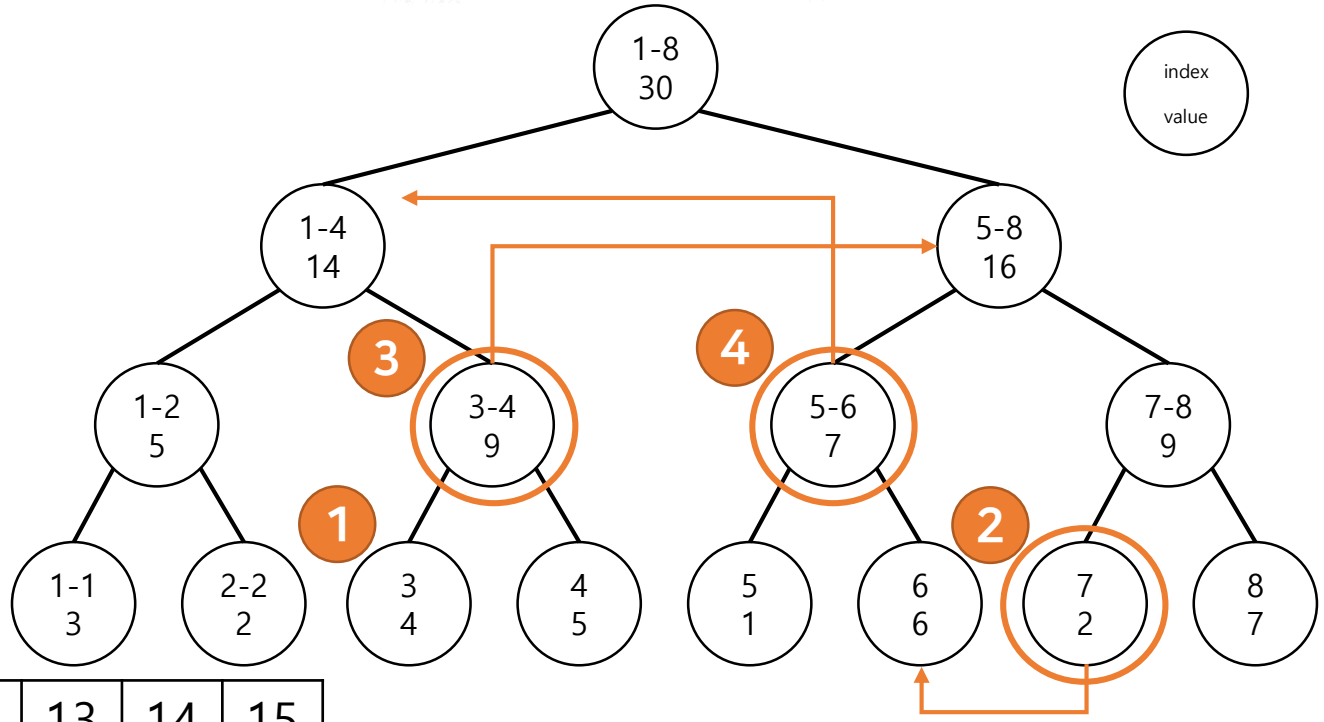
- 리프노드 순회 (index = [S] ~ [S + N – 1])
  - 노드에 배열의 값 저장
- 내부노드 순회 (index = [S – 1] ~ [1])
  - 왼쪽 자식 index * 2
  - 오른쪽 자식 index * 2 + 1
  - 왼쪽 자식 + 오른쪽 자식 값을 합쳐서 노드에 저장



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)

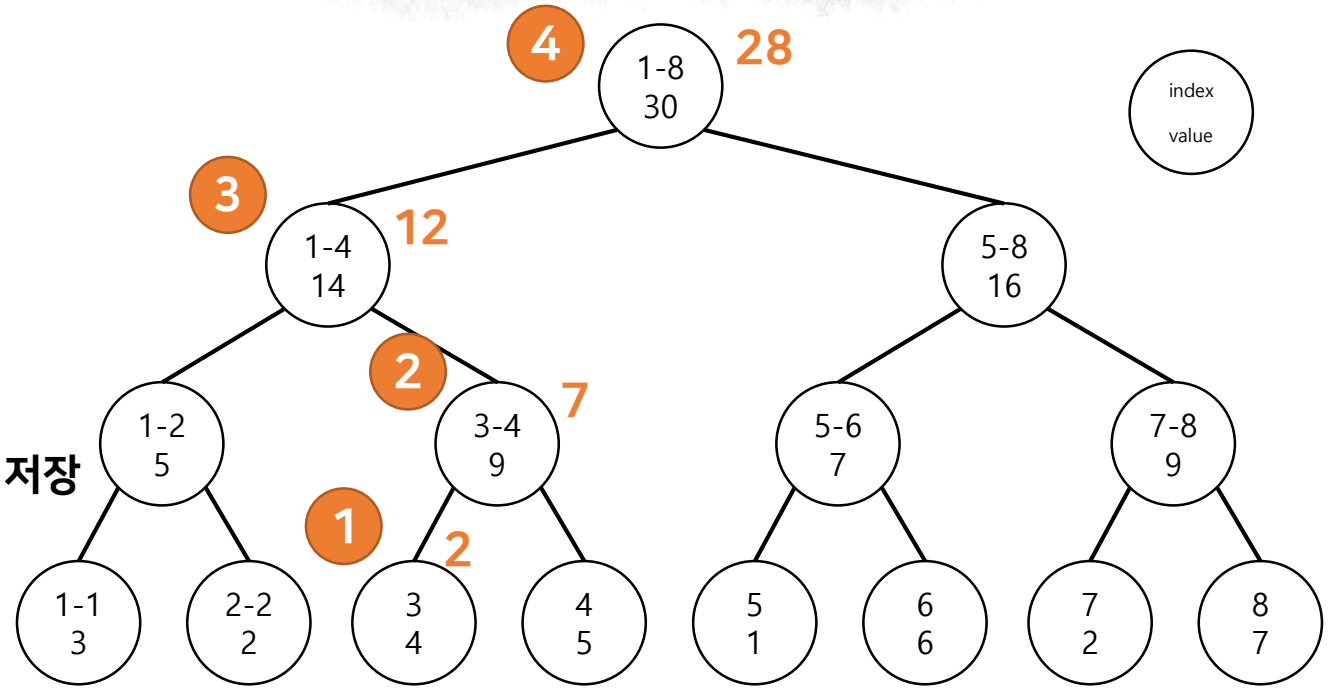
- 리프 노드부터 시작 query(3, 7)
  - nodeLeft = S + queryLeft – 1
  - nodeRight = S + queryRight – 1
- while(nodeLeft <= nodeRight)
- leftNode 분기 조건
  - 짝수 : 부모 값 사용 가능 => leftNode = leftNode / 2
  - 홀수 : 현재 노드 값 추가 => leftNode = (leftNode + 1) / 2
- rightNode 분기 조건
  - 짝수 : 현재 노드 값 추가 => rightNode = (rightNode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

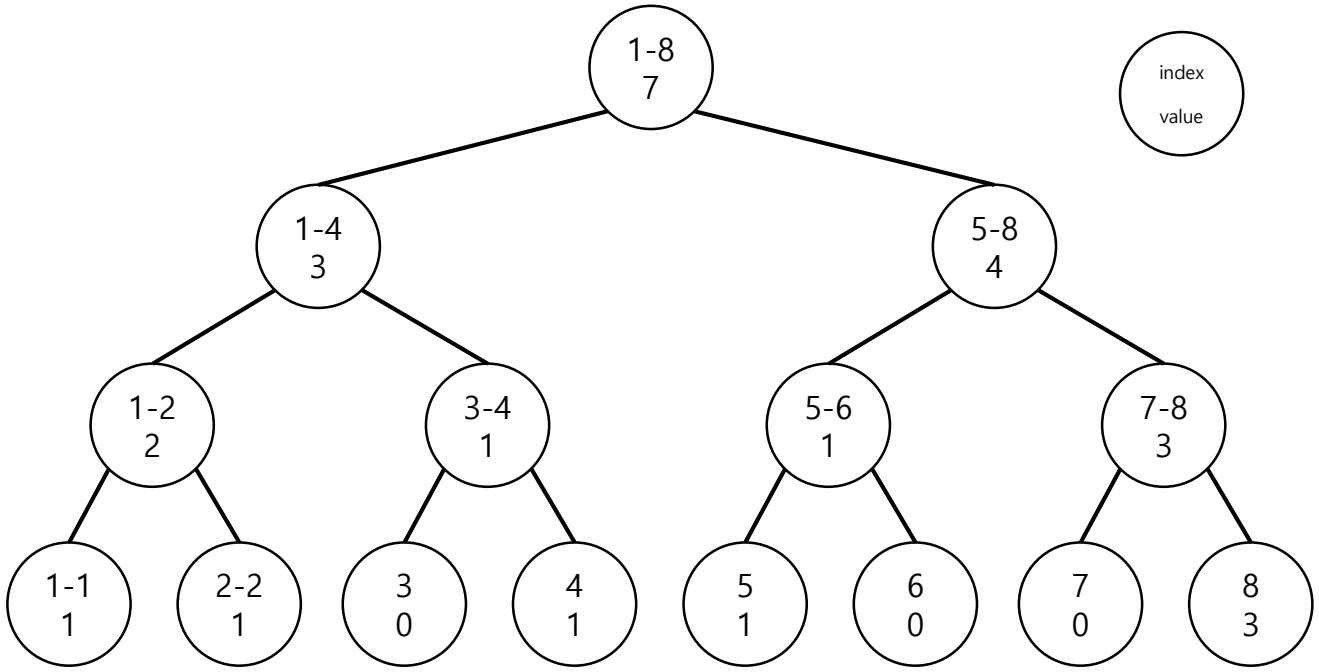


28 12 7 2

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

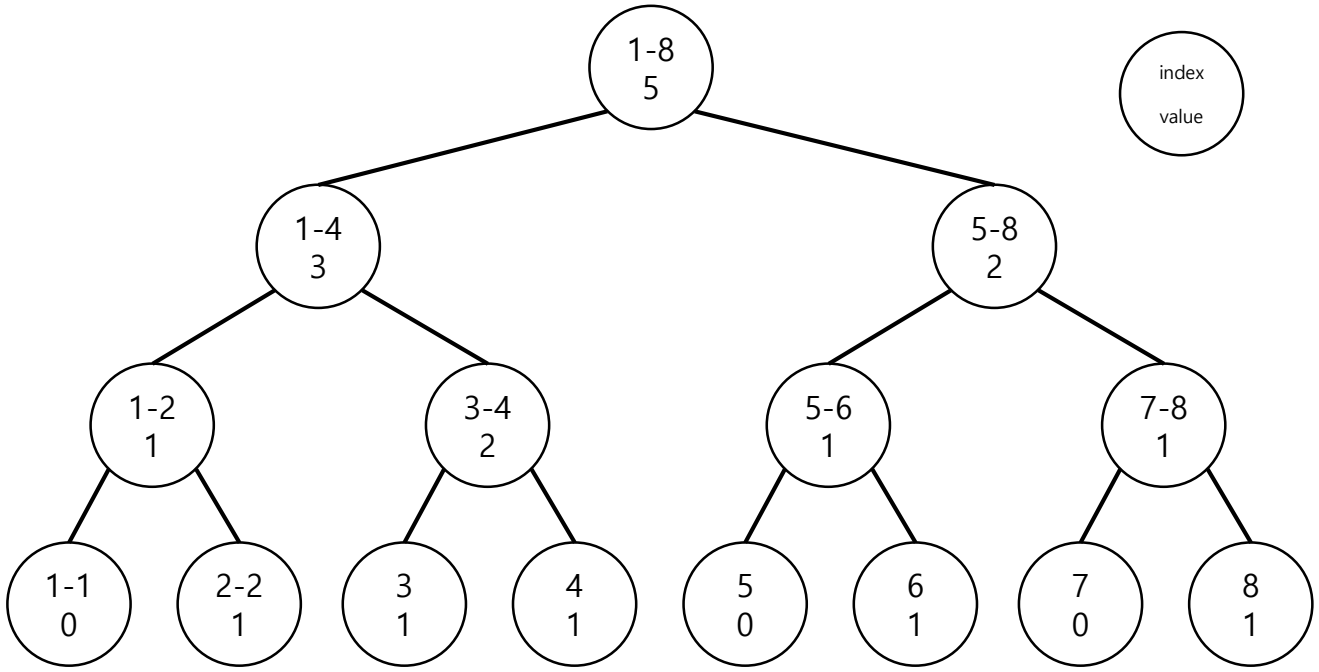
N번째 카드를 가진 사람

- [2, 4, 5, 8, 8, 8, 1], Q (6, 5, 1)
- index = 카드에 적힌 수
- value = 해당 카드를 가진 참가자 수
- 풀이 원리
  - 카드를 가진 사람을 tree에 넣음
  - Root부터 Query시작
  - 좌측 자식 값 >= Query Count 면 좌측으로 (Query Count 변화 X)
  - 좌측 자식 값 < Query Count 면 우측으로 (Query Count -= 좌측 자식 값)
  - Leaf에 도달하면 해당 노드의 카드에 적힌 수가 원하는 참가자 번호
  - Query 수행 후 해당 숫자가 적힌 카드 수 -1



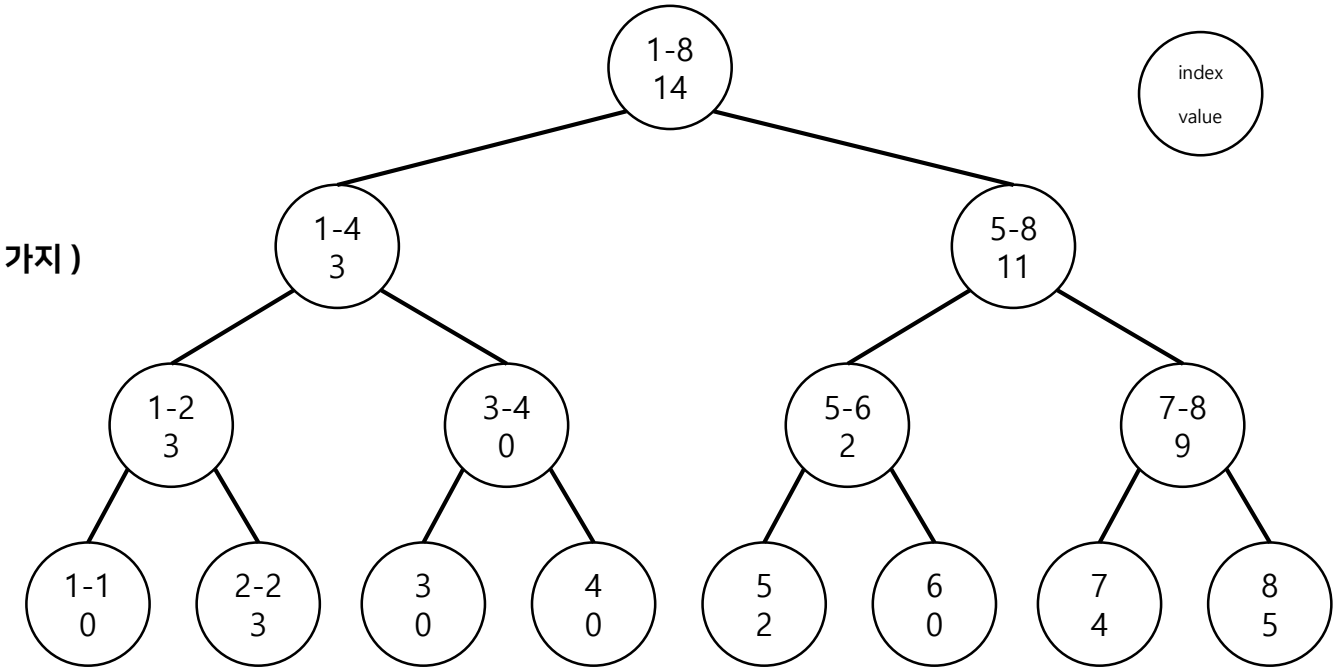
두 가지 비교조건

- 키(순번) [175(1), 182(2), 178(3), 179(4), 170(5), 179(6), 171(7), 185(8)], Q (시작,끝,키) (3, 7, 175), Q (1, 8 180)
- index = 줄에 서 있는 사람 순번
- value = 해당 순번 사람 수
- 풀이 원리
  - 질문을 선택
  - 질문의 키보다 큰 사람을 tree에 넣음
  - 현재 tree에는 질문보다 큰 사람만 들어가 있음
  - 질문의 구간의 합을 구함 => 해당 구간에 질문 보다 큰 사람의 수
- 정렬을 적용하여 계산 효율화
  1. 질문, 사람을 키가 큰 순으로 정렬
     - [185(8), 182(2), 179(4), 179(6), 178(3), 175(1), 171(7), 170(5)]
     - [Q(1, 8, 180), Q(3, 7, 175)]
  2. 질문 선택
  3. 질문 보다 키 큰 사람을 tree에 넣음 (사람 순회 index를 유지!!)
  4. 질문 구간 합 구하기
  5. 2 부터 다시 반복



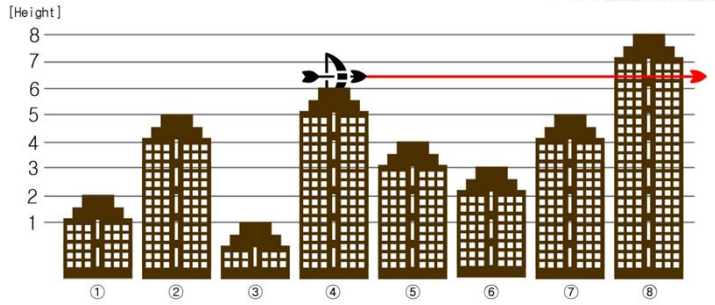
두 가지 비교 조건

- (x,y,점수) [(3 1 7), (4 4 6), (1 3 9), (7 2 3), (8 5 2), (2 6 8), (5 8 5), (6 7 4)]
- X 내림차순 정렬 [(8 5 2), (7 2 3), (6 7 4), (5 8 5), (4 4 6), (3 1 7), (2 6 8), (1 3 9)]
- index = Y 좌표
- value = 해당 Y 좌표의 탱크 점수
- 풀이 원리
  - Indexed Tree에는 기준을 하나 밖에 적용하지 못함 (문제는 X, Y 기준이 두 가지)
  - X 내림차순 정렬하여 X가 큰 순서대로 Tree에 넣으면서 진행
  - 선택된 탱크를 넣기 직전에는 tree에 자신보다 X가 큰 탱크만 들어가 있음
  - 이때 자신보다 Y가 큰 탱크의 점수를 합하면 됨.



두 가지 비교조건 (입력이 곧 쿼리)

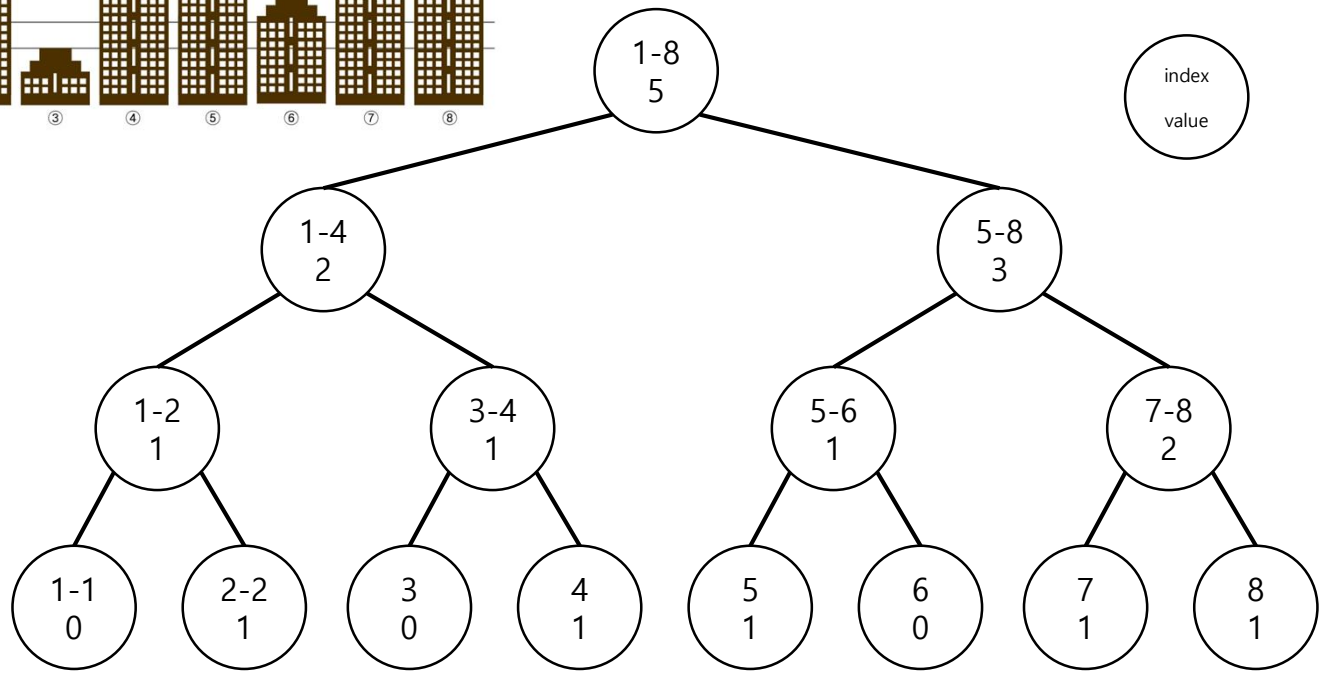
N	1	2	3	4	5	6	7	8
H	2	5	1	6	4	3	5	8
P	2	1	5	3	0	1	2	0



- index = 빌딩 번호
- value = 빌딩 의 수
- 풀이 원리
  - 문제에 주어진 조건 : 빌딩 높이, 빌딩 번호
  - 높이 내림차순으로 정렬하여 하나씩 넣으면 조건을 하나로 단축 가능

N	8	4	2	7	5	6	1	3
H	8	6	5	5	4	3	2	1
P	0	3	1	2	0	1	2	5

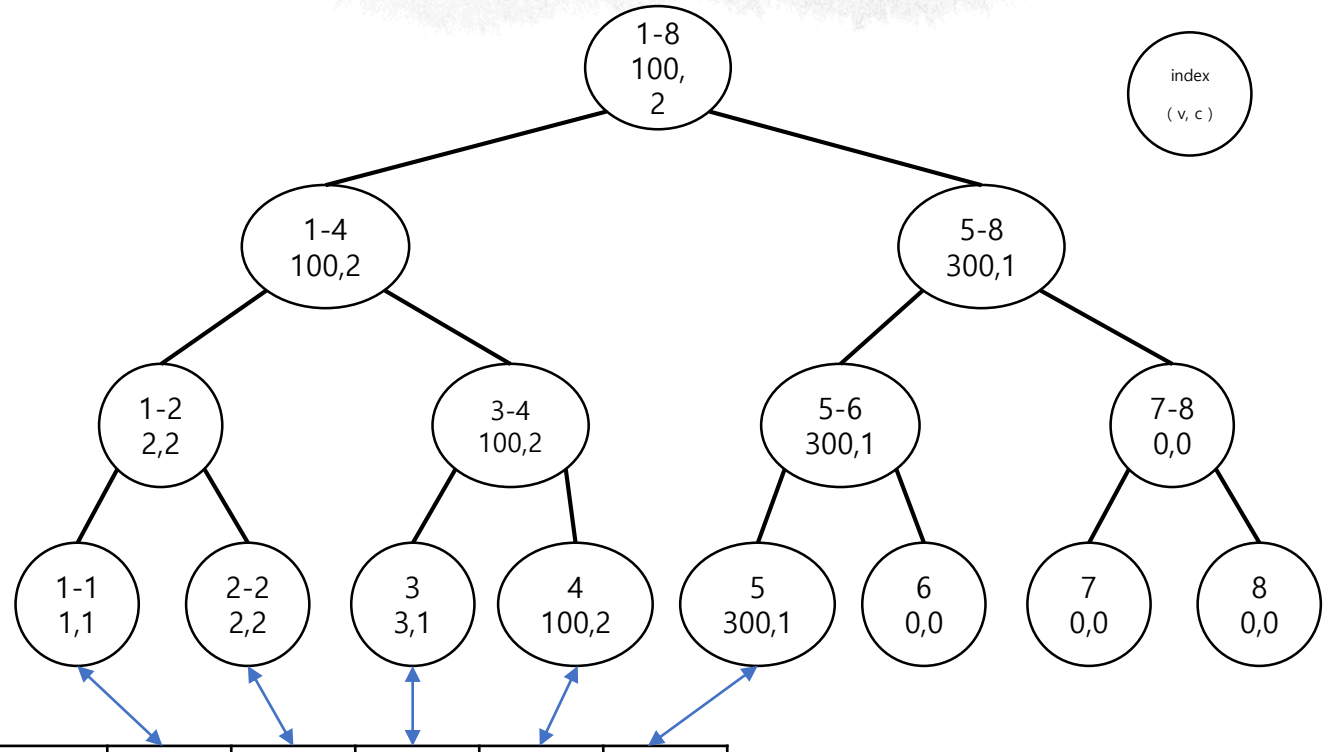
  - 빌딩을 높은 순으로 하나씩 선택
  - 트리에서 빌딩의 왼쪽에 있는 빌딩 수 = 현재 빌딩보다 높거나 같고 왼쪽에 있는 수
  - 트리에서 왼쪽 수 + 파워 + 1번째 빌딩을 찾음 => 화살이 맞을 빌딩
  - 선택한 빌딩을 트리에 넣음



리프 노드 압축

i	1	2	3	4	5	6	7
v	1	2	100	300	3	100	2

- index = 수열을 이루는 수
- value = 최빈값과 개수 (value, count)
- 풀이 원리
  - 부분 수열의 수를 tree에 넣는다.
  - Root의 value 의미 => 부분 수열의 최빈값
  - 부분 수열을 오른쪽으로 한 칸 이동
    - 제일 왼쪽 값 제거
    - 오른쪽에 새로운 값 추가
- 리프 노드 크기 최적화
  - 열을 이루는 수가 최대 10억 -> 트리노드가 최소 20억 개 이상 필요
  - 수의 개수는 30만 개 -> 맵핑 이용
  - 수열의 값을 HashMap으로 IndexMap을 만듬
  - ex) 수열[3] 값 100을 Tree에 추가 -> 100은 4번째 숫자
    -> 4번째 리프노드 업데이트



Value	번째	1	2	3	4	5
Key	값	1	2	3	100	300

UPDATE 최적화 (가지치기)

- 맞춘 범위 (시작,끝) [(1, 3), (3, 4), (2, 5)]
- Index = 과녁 번호
- Value = 이미 맞춘 과녁의 수
- 풀이 원리
  - 손님을 순서대로 선택
  - 손님이 맞춘 과녁의 개수 (b – a + 1) 에서 이미 맞춘 과녁 개수 제외 (구간합 a ~ b)
  - 손님이 맞춘 과녁을 맞췄다고 갱신
- 갱신 효율화
  - 범위 갱신 : Top-down, Bottom-Up 가능
  - 가지치기 : 구간의 크기와 노드에 값이 같을 경우 갱신 중단 가능
  - 가지치기는 Top-Down만 가능함.