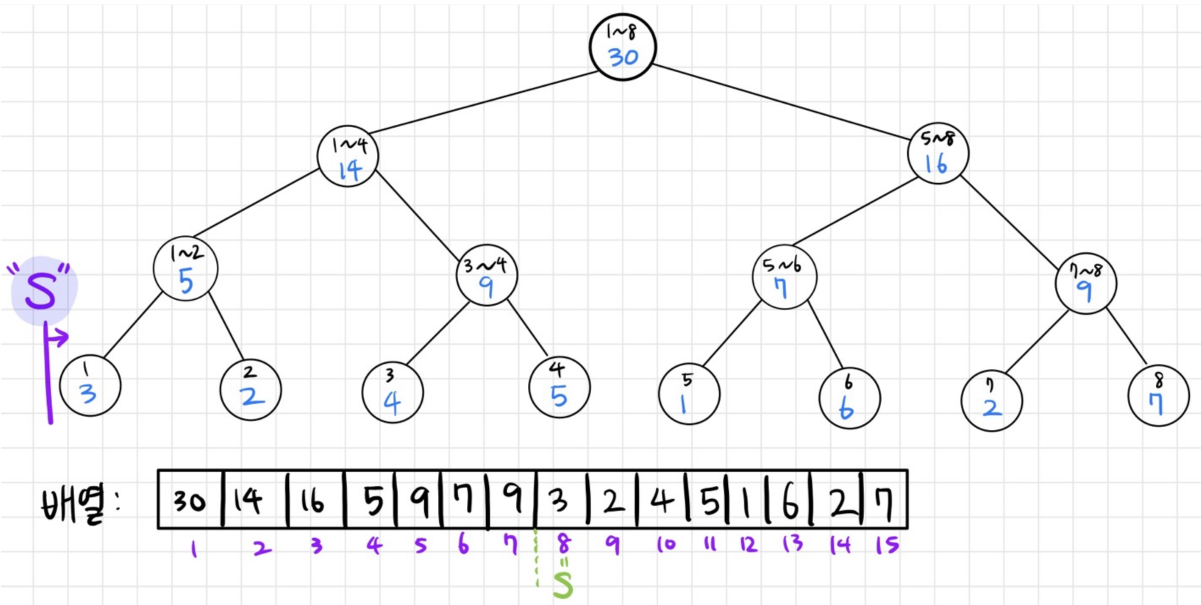
인덱스 트리 (Indexed Tree)

정의

- 포화 이진트리 형태의 자료구조
- 구간 합을 구하는 데에 쓰이는 자료구조



특징

- Leaf 노드 개수가 N개 이상

    - 높이는 최소 **logN**
    - 비어있는 공간 발생 시, 구조에 지장이 가지 않도록 초기값 설정
        - 0을 값으로 갖는 Leaf 노드 추가

주로 쓰이는 곳

- **배열의 특정 인덱스의 변경이 계속 발생**하는 상황에서 부분합을 계속해서 구해야 할 때

🎈 변경이 일어나지 않을 때 부분합 구하는 방법 & 시간복잡도

1. 일반적인 방법

- 구간 left ~ right의 부분합을 구할 때, 배열의 left부터 right까지 전부 더해야하기 때문에 O(N)

2. 누적합 배열

- 누적합 배열 자체를 구할 때는 모든 데이터를 한번씩은 거쳐야 하므로 O(N)
- 하지만 누적합 배열을 구했다는 전제 하에 특정 구간의 부분합을 구하는 건 O(1)
    - 예) 3~5의 구간의 부분합 = 누적합[5] - 누적합[2]

🤔 하지만 변경이 일어난다면?

- 특정 인덱스를 변경하게 된다면, 부분합 배열의 많은 변경이 일어나게 된다. 그 변경 시간 → O(N)
- 데이터의 변경과 부분합을 구하는 게 M번 실행된다면 (누적합 배열 사용 전제) **O(1 * N * M)**의 시간복잡도를 갖는다.

> 하지만 우리는 트리의 특성을 이용할 것이기 때문에 부분합을 구하는데 O(logN), 특정 인덱스 변경에 의한 부분합 배열 변경시간 O(logN)을 갖는다. 결론적으로 M번 실행한다면 **O(MlogN)**의 시간복잡도를 갖는다.

구성 방법

1. 리프 노드 개수가 N개 이상이 되도록 높이가 T인 배열 생성, 즉 2^T의 크기를 갖는 배열 생성
2. 리프 노드에 데이터 입력
3. 내부노드에 양쪽 자식값 참조하여 데이터 입력

함수별 로직

각 함수는 Top-down / Botton-up 의 두 방식으로 구현될 수 있다.

1. Init

(1) Top-down : init(int left, int right, int node) → Root부터 시작

> node : 노드의 값이 아닌, 노드 번호

1. 내부 노드일 경우 (left ≠ right)
    a. 왼쪽 자식 init(left, mid, node*2) 호출
    b. 오른쪽 자식 init(mid+1, right, node*2+1) 호출
    c. 왼쪽 자식 + 오른쪽 자식 → 현재 노드에 저장
    d. 노드값 리턴
2. 리프 노드일 경우 (left == right)
    a. 노드에 배열값 저장
    b. 노드값 리턴

(2) Bottom-up → init은 주로 이 방식으로 진행

1. 리프 노드 순회 (트리에서의 인덱스는 S ~ S+N-1)
    a. 노드에 배열값 저장
2. 내부 노드 순회 (트리에서의 인덱스는 S-1 ~ 1)
    a. 왼쪽 자식: index*2
    b. 오른쪽 자식: index*2+1

c. 왼쪽 자식 + 오른쪽 자식 → 노드에 저장

2. Query

(1) Top-down : query(int left, int right, int node, int queryLeft, int queryRight)

1. 노드가 query 범위 밖에 위치한 경우 → 연관X
2. 노드가 query 범위 안에 들어올 경우 → 판단 가능
    a. 현재 노드값 리턴
3. 노드가 query 범위에 걸쳐 있을 경우 → 현재 노드에서는 판단할 수 없으니 자식노드에게 맡기자.
    a. 왼쪽 query(left, mid, node*2, queryLeft, queryRight) 호출
    b. 오른쪽 query(mid+1, right, node*2+1, queryLeft, queryRight) 호출
    c. 왼쪽 query값 + 오른쪽 query값 → 리턴

(2) Bottom-up : query(int queryLeft, int queryRight)

- 자식 기준,
    - 내 값을 다이렉트로 쓸 거냐(부모의 값이 유효하지 않음)
    - 부모의 값을 쓸 거냐 (부모의 값이 범위 안에 들어와서 유효함)

1. 리프 노드부터 시작
    a. S 활용하여 바로 접근 가능
        i. leftNode = S + queryLeft - 1
        ii. rightNode = S + queryRight - 1
2. while (leftNode ≤ rightNode)
    a. leftNode 분기 조건
        i. leftNode % 2 == 0
            1. 부모값 사용 → leftNode /= 2
        ii. leftNode % 2 ≠ 0

            1. 현재 본인 값 사용 → leftNode = (leftNode+1) / 2

   b. rightNode 분기 조건

      i. rightNode % 2 == 0

         1. 현재 본인 값 사용 → rightNode = (rightNode-1) / 2

      ii. rightNpde % 2 ≠ 0

         1. 부모값 사용 → righNode /= 2

3. Update

(1) Top-down : update(int left, int right, int node, int target, int diff)

1. 노드가 target을 포함하지 않는 경우 → 연관X, 무시하면 됨
2. 노드가 target을 포함하는 경우
   a. 현재 노드에 diff 반영
   b. 자식이 있을 경우
      i. 왼쪽 자식: update(left, mid, node*2, target, diff)
      ii. 오른쪽 자식: update(mid+1, right, node*2+1. target, diff)

(2) Bottom-up : update(int target, int value)

1. 리프노드부터 시작
   a. node = S+target - 1
2. 노드를 해당 value로 갱신
3. 부모로 이동 : node /= 2
4. while(node ≥ 1)
   a. 좌측 노드의 값과 우측 노드의 값을 해당 노드에 저장
      i. tree[node] = tree[node*2] + tree[node*2+1]
   b. 부모로 이동: node /= 2