Segment Tree (1/4)

세그먼트 트리의 개념 및 필요성

서울대학교 컴퓨터공학부 김동현

수열과 쿼리

길이 N의 수열 $A_1, A_2, \cdots, A_N$이 주어진다. 다음 쿼리를 처리하라.
- L, R이 주어지면 $A_L + \cdots + A_R$을 출력하라.

- 위 문제는 누적합 배열 ($S_i = A_1 + A_2 + \cdots + A_i$)를 만드는 것으로 간단하게 O(N) 전처리 + 쿼리당 O(1)에 해결할 수 있습니다.

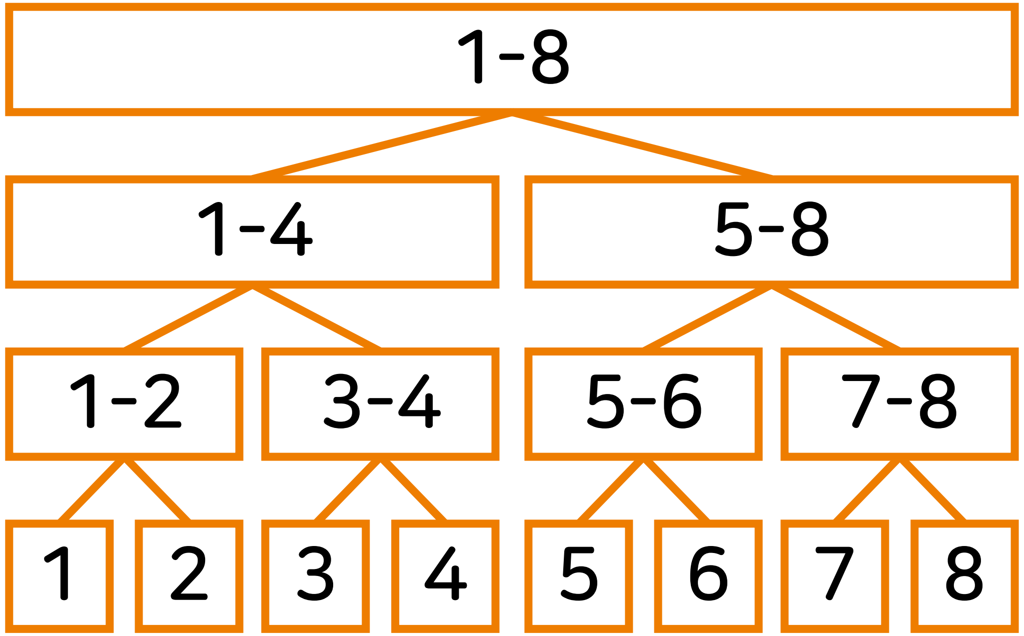
수열과 쿼리

길이 N의 수열 $A_1, A_2, \cdots, A_N$이 주어진다. 다음 쿼리를 처리하라.

- L, R이 주어지면 $A_L + \cdots + A_R$을 출력하라.
- X, V가 주어지면 A_X를 V로 바꾸어라.

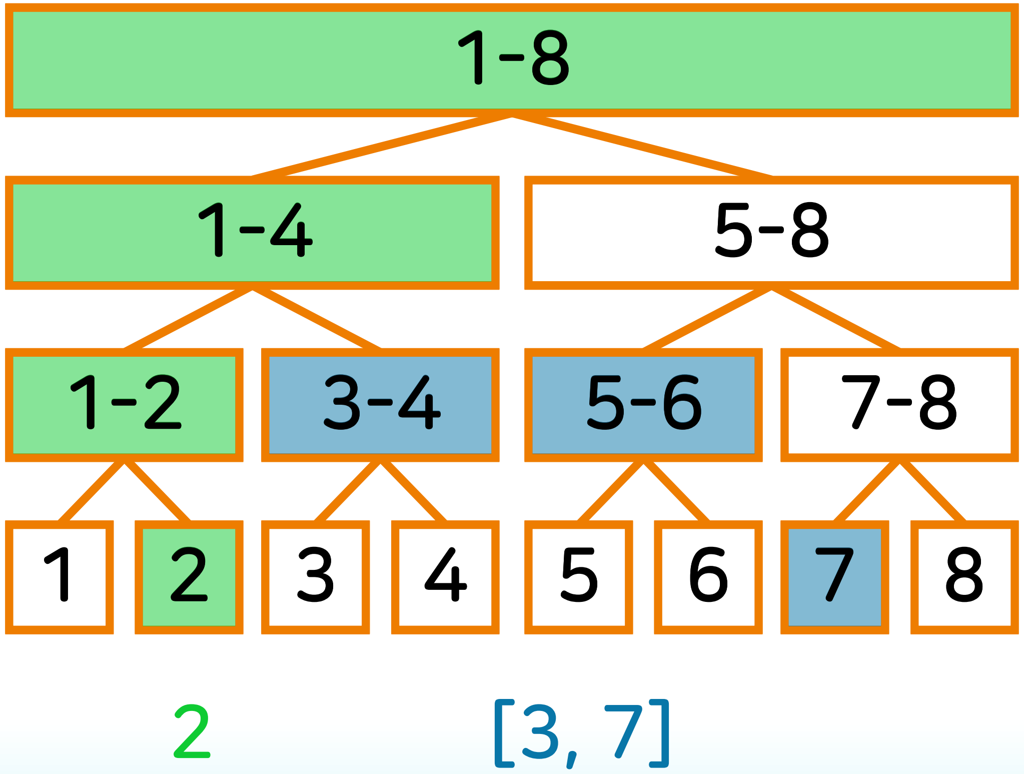
- 누적합 배열을 사용하면 값 변경 쿼리를 O(N)에 할 수 밖에 없습니다.
- 세그먼트 트리를 사용하면 두 쿼리를 각각 O(logN)에 처리할 수 있습니다!

세그먼트 트리의 기본 원리



- 각 노드는 특정 구간을 대표합니다.
- 노드들은 이진 트리 구조를 이룹니다.
  - 부모 노드가 대표하는 구간은 자식 노드 두 개가 대표하는 구간의 합집합입니다.
- 수열의 총 길이는 어떤 자연수여도 상관 없지만, 2^k 꼴의 수를 택하면 비재귀 방식의 구현이 편리해집니다.
  - 보통 원래 수열의 길이 N을 넘는 가장 작은 2^k을 택합니다.

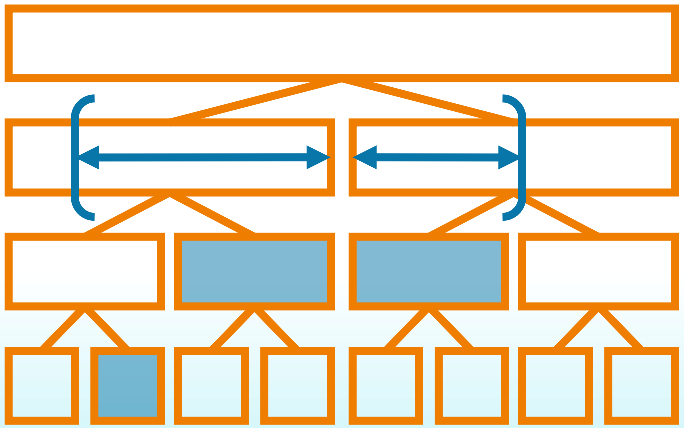
세그먼트 트리의 기본 원리



- 이런 식으로 구성한 트리는 다음과 같은 유용한 성질을 가집니다.
  - 임의의 인덱스(점)에 대해, 그 점을 포함하는 노드는 O(logN)개 존재합니다.
  - 임의의 구간에 대해, 서로 표현하는 구간이 겹치지 않는 O(logN)개의 노드로 그 구간을 쪼갤 수 있습니다.

세그먼트 트리의 기본 원리 - 구간 쪼개기

- (길이가 2 이상인) 어떤 구간에 대해, 그 구간을 포함하는 가장 아래쪽 노드를 루트로 하는 서브트리를 살펴봅시다.
- 가정에 의해, 바로 밑 절반에서 이 구간은 둘로 나뉩니다.
- 나뉜 두 부분 각각에 대해, O(logN)개의 겹치지 않는 노드들로 해당 구간을 표현할 수 있습니다: 구간 길이의 이진수 표현을 생각



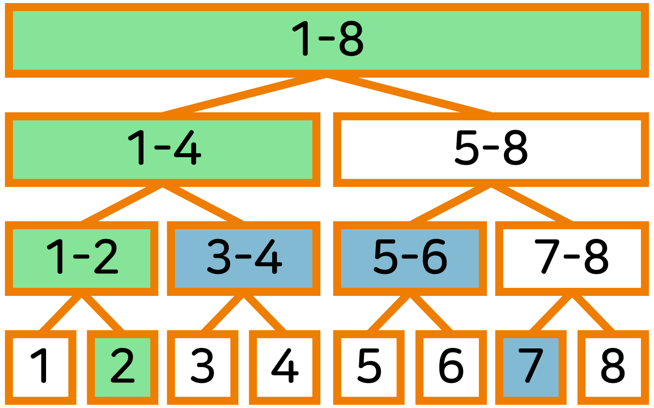
왼쪽 : 3 = $11_{(2)}$
오른쪽 : 2 = $10_{(2)}$

세그먼트 트리의 기본 원리

점 갱신 + 구간 쿼리

길이 N의 수열 A_1, A_2, …, A_N과 쿼리가 주어진다.

- L, R이 주어지면 A_L + … + A_R을 출력하라.
- X, V가 주어지면 A_X를 V로 바꾸어라.



- 세그먼트 트리의 각 노드는 자신이 담당하는 구간의 합을 저장합니다.
- 점 갱신 쿼리 : 그 점을 포함하는 O(logN)개의 노드에 대해서 갱신
- 구간 합 쿼리 : 구간을 표현하는 O(logN)개의 노드들을 구해서 각각에 적힌 값들을 모두 합해서 출력
  - 그 노드들을 어떻게 구하는지는 다음 영상에서 살펴볼 예정입니다.

세그먼트 트리의 장점 - 확장성

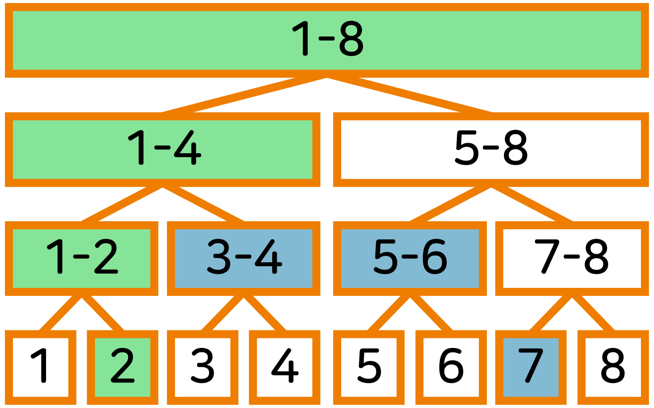
- 세그먼트 트리를 이용해 풀 수 있는 문제는 매우 다양합니다.
  - 한 점의 값 갱신 / 구간의 최댓값(최솟값) 구하기
  - 구간에 값 더하기 / 한 점에 적힌 값 구하기
  - 구간에 A_i = max(A_i, X) 연산 취하기 / 한 점에 적힌 값 구하기
  - …
  - 구간에 값 더하기 / 구간의 합 구하기 / 구간의 최댓값 구하기
  - …
  - 한 점의 값 갱신 / 어떤 구간에서 최대 부분합 구하기
  - …

세그먼트 트리의 기본 원리

구간 갱신 + 점 쿼리

길이 N의 수열 $A_1, A_2, \cdots, A_N$과 쿼리가 주어진다.

- X가 주어지면 A_X를 출력하라.
- L, R, V가 주어지면 L과 R 사이의 모든 A_i에 V를 더해라.



- 세그먼트 트리의 각 노드는 자신이 담당하는 구간에 공통적으로 더해진 수를 저장합니다.
- 점의 값 쿼리 : 그 점을 포함하는 O(logN)개의 노드에 대해 합 구하기
- 구간 갱신 쿼리 : 구간을 표현하는 O(logN)개의 노드들을 구해서 각각의 노드에 V를 더해 줌

세그먼트 트리의 구현 방식

	재귀	비재귀
장점	확장성이 뛰어남 (좀 더 복잡한 형태의 구조도 구현 가능)	구현이 매우 간단하다 작동이 빠르다
단점	구현이 상대적으로 길다 작동이 느리다	간단한 형태만 구현 가능

감사합니다

Segment Tree (2/4)

세그먼트 트리의 재귀/비재귀 구현

서울대학교 컴퓨터공학부 김동현

세그먼트 트리의 구현 방식 (다시)

	재귀	비재귀
장점	확장성이 뛰어남 (좀 더 복잡한 형태의 구조도 구현 가능)	구현이 매우 간단하다 작동이 빠르다
단점	구현이 상대적으로 길다 작동이 느리다	간단한 형태만 구현 가능

세그먼트 트리의 재귀 구현

점 갱신 + 구간 쿼리

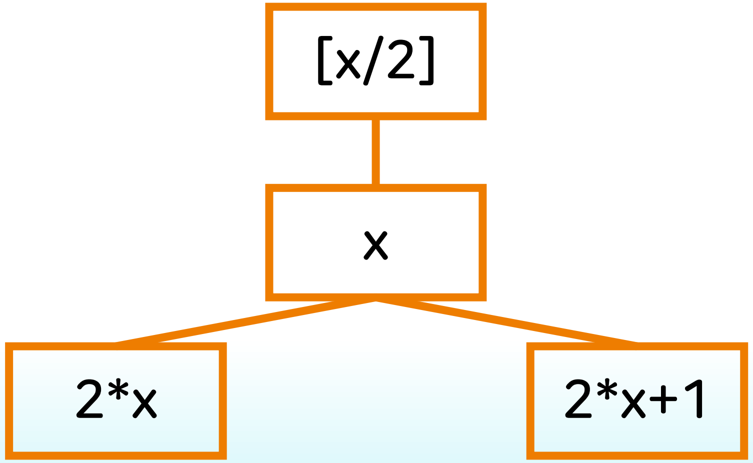
길이 N의 수열 $A_1, A_2, \cdots, A_N$과 쿼리가 주어진다.

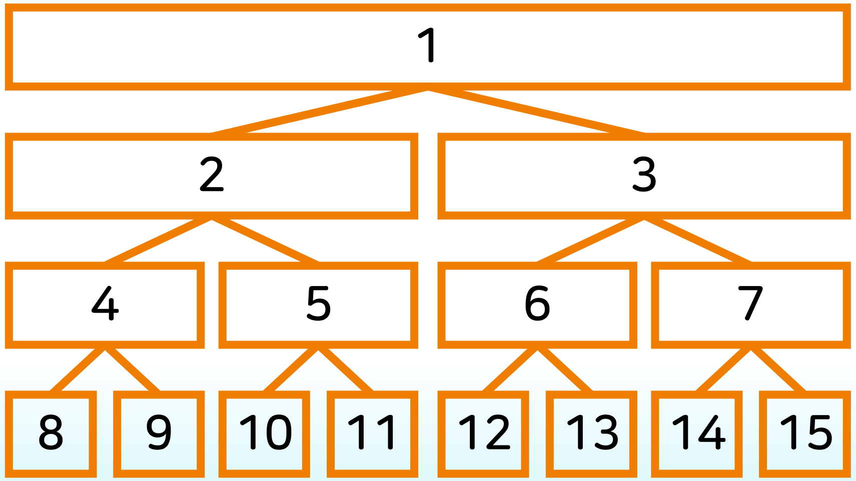
- X, V가 주어지면 A_X를 V로 바꾸어라.
- L, R이 주어지면 $A_L + \cdots + A_R$을 출력하라.

- 이 문제를 해결하는 데 사용할 세그먼트 트리를 재귀함수를 이용해 구현하여 봅시다.

세그먼트 트리의 재귀 구현

- 트리 데이터의 저장 : 배열!
  - 루트 노드의 번호는 1
  - 노드 x의 자식 : 2*x, 2*x+1
  - 노드 x의 부모 : [x/2]
  - 수열 길이가 2^k일 때 배열 크기 : 2^(k+1)

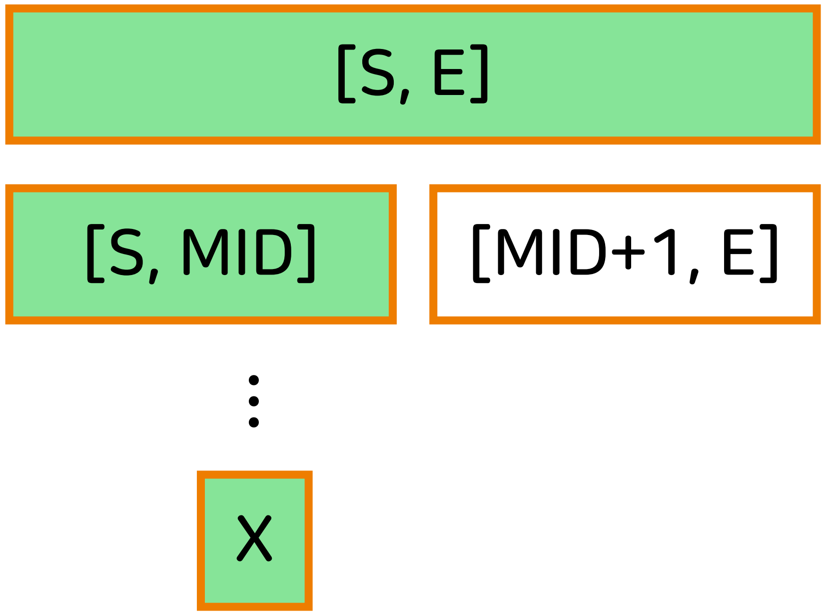




세그먼트 트리의 재귀 구현

- Update 함수 (점 갱신)

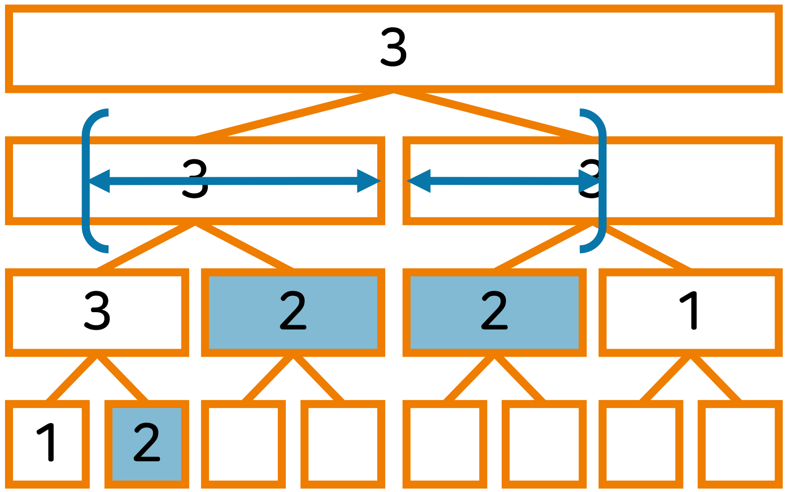
```
#define SIZE (1 << 20) // 2^20 (=1048576)
int tree[2 * SIZE];

// A[X] = V로 갱신.
// 초기 호출 : update(X, V, 1, 1, SIZE)
void update(int X, int V, int node, int S, int E){
    if(S == E){
        tree[node] = V;
        return;
    }
    int MID = (S + E) / 2;
    if(X <= MID) update(X, V, 2 * node, S, MID);
    else update(X, V, 2 * node + 1, MID + 1, E);
    tree[node] = tree[2 * node] + tree[2 * node + 1];
}
```



세그먼트 트리의 재귀 구현

- Query 함수 (구간 합 쿼리)

```
// A[L] + … + A[R] 구하기.
// 초기 호출 : query(L, R, 1, 1, SIZE)

int query(int L, int R, int node, int S, int E){
    if(R < S || E < L) return 0; // 1
    if(L <= S && E <= R) return tree[node]; // 2
    int MID = (S + E) / 2;
    return query(L, R, 2 * node, S, MID) +
           query(L, R, 2 * node + 1, MID + 1, E); // 3
}
```



- 관여하는 노드의 총 개수는 O(logN)개.

세그먼트 트리의 비재귀 구현

점 갱신 + 구간 쿼리

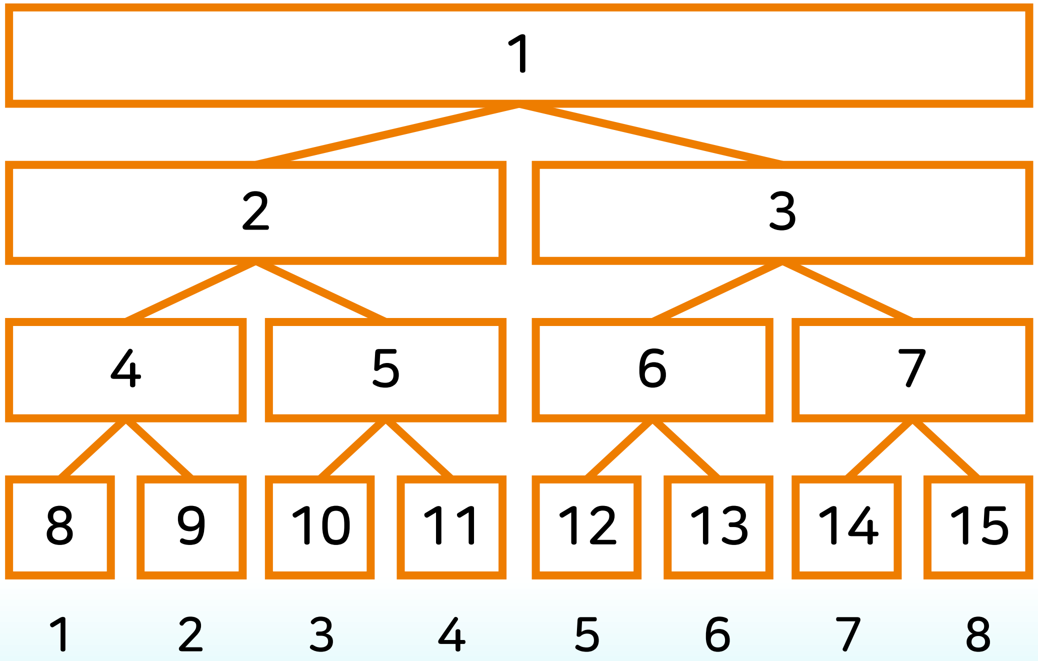
길이 N의 수열 $A_1, A_2, \cdots, A_N$과 쿼리가 주어진다.

- X, V가 주어지면 A_X를 V로 바꾸어라.
- L, R이 주어지면 $A_L + \cdots + A_R$을 출력하라.

- 이번에는 재귀함수를 사용하지 않고 구현해 봅시다.

세그먼트 트리의 비재귀 구현

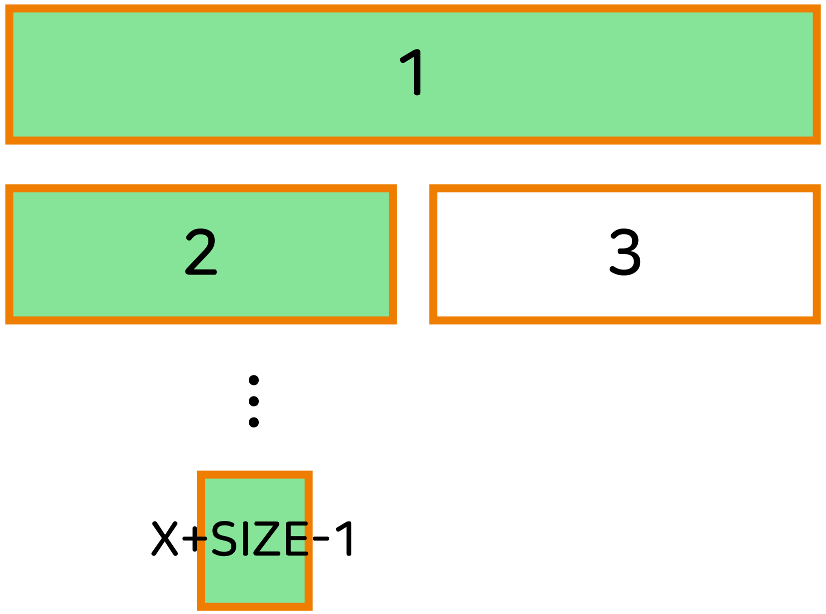
- A[x]에 해당하는 리프 노드의 번호 : x + SIZE – 1.
  - 원래 수열의 길이 N이 SIZE 미만이라면 (x + SIZE)라고 해도 무방.



세그먼트 트리의 비재귀 구현

- Update 함수 (점 갱신)

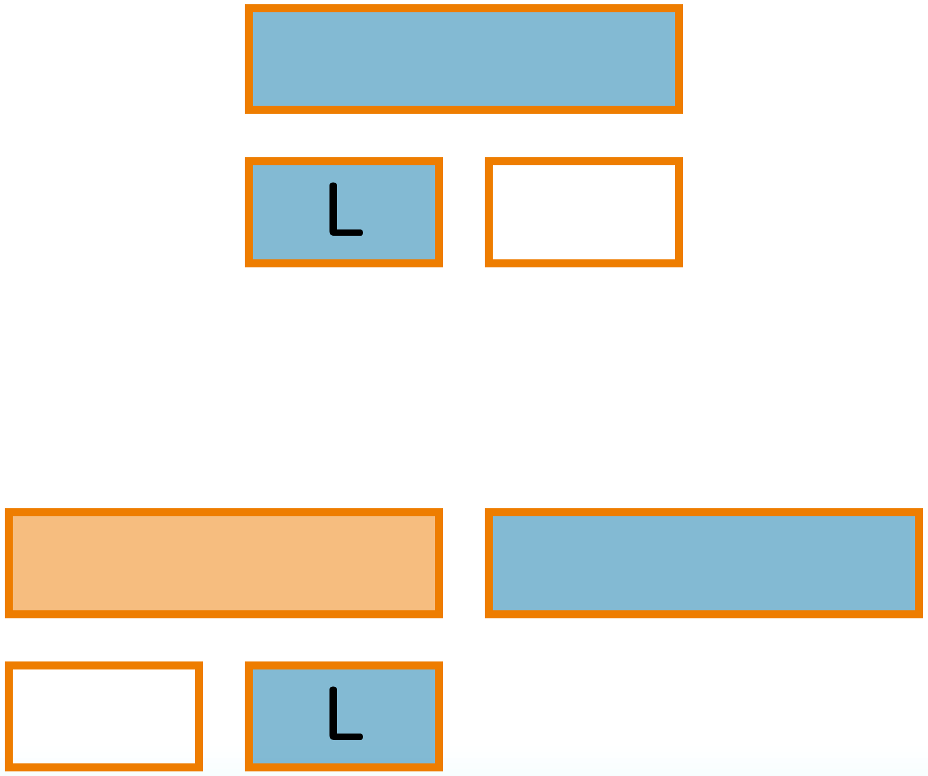
```
#define SIZE (1 << 20) // 2^20 (=1048576)
int tree[2 * SIZE];

// A[X] = V로 갱신.
void update(int X, int V){
    X += SIZE - 1;
    tree[X] = V;
    for(X /= 2; X >= 1; X /= 2){
        tree[X] = tree[2 * X] + tree[2 * X + 1];
    }
}
```



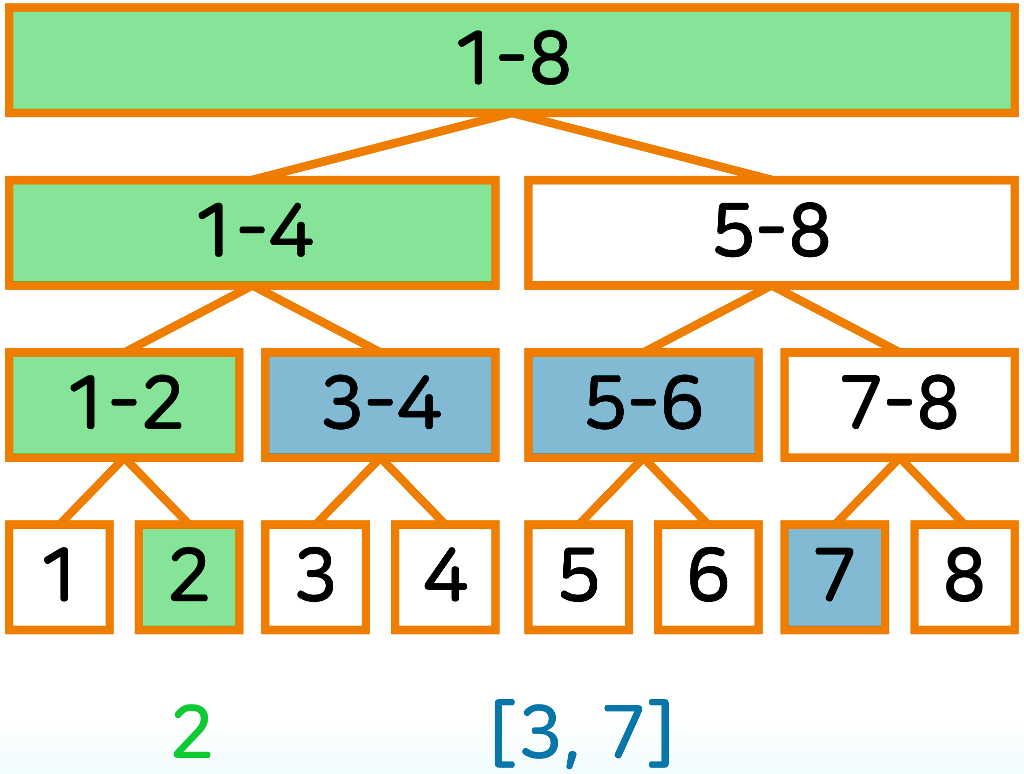
세그먼트 트리의 비재귀 구현

- Query 함수 (구간 합 쿼리)

```
// A[L] + … + A[R] 구하기.
int query(int L, int R){
    L += SIZE – 1;
    R += SIZE – 1;
    int res = 0;
    for(; L <= R; L /= 2, R /= 2){
        if(L % 2 == 1) res += tree[L++];
        if(R % 2 == 0) res += tree[R--];
    }
    return res;
}
```



점 쿼리 + 구간 갱신 ⇔ 구간 쿼리 + 점 갱신



- 점 : 그 점을 포함하는 O(logN)개 노드들을 고려
- 구간 : 그 구간을 O(logN)개의 노드로 겹치지 않게 쪼개기

- 쿼리인지 갱신인지에 따라 약간만 다르게 구현하면 됩니다.
- 앞의 구현을 매우 간단하게 응용하면 충분합니다.

감사합니다

Segment Tree (3/4)

세그먼트 트리의 응용

서울대학교 컴퓨터공학부 김동현

세그먼트 트리로 풀 수 있는 다양한 문제

- 집합(multiset)에 수 추가 / 삭제 / k번째 수 구하기 / x 이상인 가장 작은 수 구하기 / x 이하인 가장 큰 수 구하기
- 점의 값 변경 / 어떤 구간 내에 존재하는 최대 연속 부분합 구하기
- 집합의 배열 $S_1, S_2, \cdots, S_N$이 있을 때
  S_i에 수 추가(삭제)하기 / $S_L \cup S_{L+1} \cup \cdots \cup S_R$의 최소 원소 구하기
- …
- 구간에 값 더하기 / 구간의 합 구하기 / 구간의 최댓값 구하기 …
  - 이건 다음 영상에

세그먼트 트리로 구현한 multiset

집합(multiset)에 수 추가 / 삭제 / k번째 수 구하기 /
x 이상인 가장 작은 수 구하기 / x 이하인 가장 큰 수 구하기

- 가능한 수의 범위 : 1 ~ SIZE
- 기본 형태 : 점 갱신 + 구간 합 쿼리를 지원하는 세그먼트 트리
- 여기서는 비재귀 구현 위주로 설명합니다

세그먼트 트리로 구현한 multiset

- insert / erase 연산 (수 삽입 / 삭제)
  - insert(x) == update(x, 1)
  - erase(x) == update(x, -1)

- 한 번에 같은 수를 여러 개 삽입/삭제할 수도 있다!

세그먼트 트리로 구현한 multiset

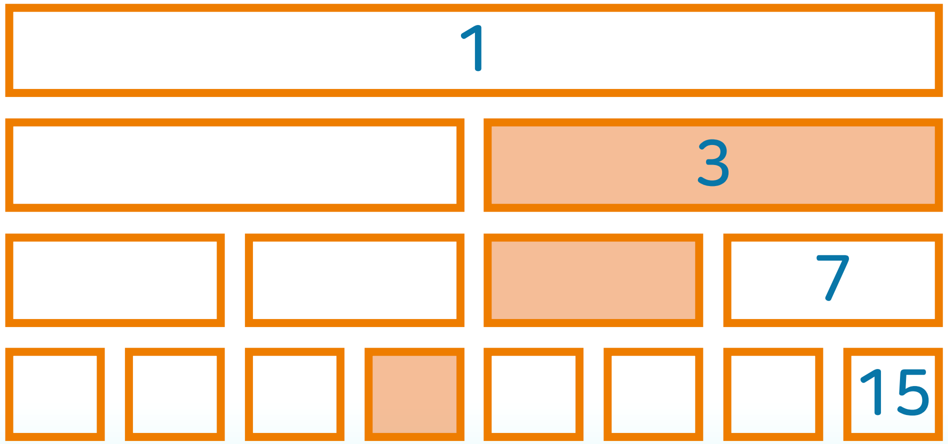
```
// 현재 multiset에서 크기 순으로 k번째인 수 구하기
// k번째 수가 없다면 -1 반환
int kth(int k){
    if(k <= 0 || tree[1] < k) return -1;
    int cur = 1;
    while(cur <= SIZE){
        if(tree[2 * cur] >= k) cur = 2 * cur;
        else{
            k -= tree[2 * cur];
            cur = 2 * cur + 1;
        }
    }
    return cur – (SIZE – 1);
}
```

- kth 연산 (현재 multiset에서 크기 순으로 k번째인 수 구하기)
  - query() 함수를 O(logN)번 호출하는 이분 탐색 → $O(\log^2 N)$
  - 세그먼트 트리의 노드에서 이분 탐색을 직접 수행 → O(logN)

세그먼트 트리로 구현한 multiset

```
// 현재 multiset에서 x 이상인 가장 작은 수
// 그런 수가 없다면 -1 반환
int lower_bound(int x){
    x += (SIZE - 1);
    while((x & (x + 1)) != 0){
        if(tree[x] > 0) break;
        x = (x + 1) / 2;
    }
    if(tree[x] == 0) return -1;
    while(x < SIZE){
        if(tree[2 * x] > 0) x = 2 * x;
        else x = 2 * x + 1;
    }
    return x - (SIZE - 1);
}
```

- lower_bound 연산 (현재 set에 있는 x 이상인 가장 작은 수)
    - x 이하인 가장 큰 수도 유사하게



분할 정복 + 세그먼트 트리

점의 값 변경 / 어떤 구간 내에 존재하는 최대 연속 부분합 구하기

- 문제를 분할 정복으로 해결하는 과정을 세그먼트 트리에 적용할 수 있습니다.
- 재귀 / 비재귀 구현 모두 가능합니다.
- 여기서는 비재귀 구현을 설명합니다.

분할 정복 + 세그먼트 트리

점의 값 변경 / 어떤 구간 내에 존재하는 최대 연속 부분합 구하기

- “최대 연속 부분합”이란, 수열에서 1개 이상의 연속한 원소를 골라 합한 값들 중 최댓값을 말합니다.
  - [3, 2, -9, 5, 1, 3]의 최대 연속 부분합은 5 + 1 + 3 = 9
- 이 문제를 분할 정복으로 풀려면 어떻게 하면 될까요?

분할 정복 + 세그먼트 트리

- 수열을 반으로 자르고, 각각에서 아래의 4가지 값을 계산했다고 합시다.
  - prefix : 맨 앞의 원소를 포함하는 최대 연속 부분합
  - suffix : 맨 뒤의 원소를 포함하는 최대 연속 부분합
  - maxsum : 해당 수열 내에 존재하는 최대 연속 부분합 (우리가 구하는 답)
  - sum : 해당 수열의 모든 원소의 합

L.prefix	L.suffix
L.maxsum	L.sum

R.prefix	R.suffix
R.maxsum	R.sum

분할 정복 + 세그먼트 트리

- 자르기 전 수열의 4가지 값을 O(1)에 계산할 수 있습니다!

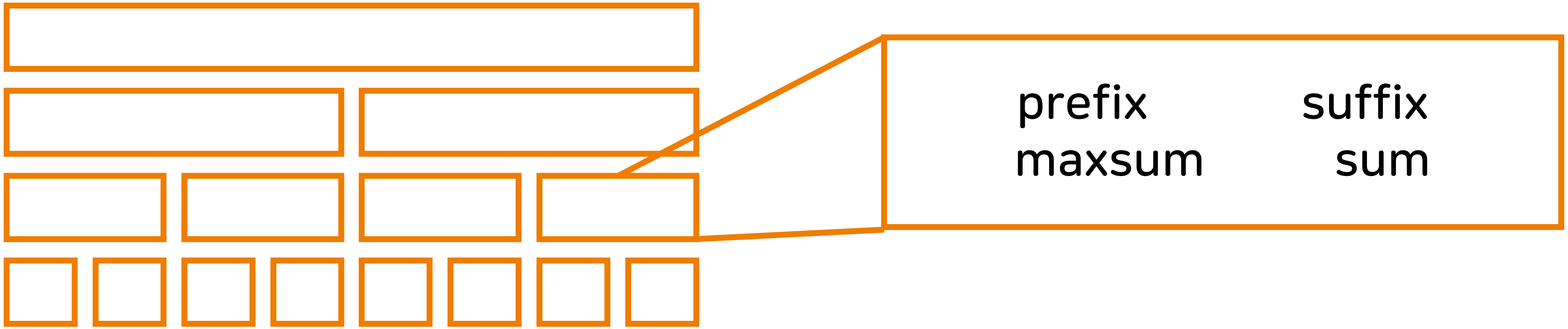
prefix = max(L.prefix, L.sum + R.prefix)
suffix = max(R.suffix, R.sum + L.suffix)
maxsum = max(L.maxsum, R.maxsum, L.suffix + R.prefix)
sum = L.sum + R.sum

L.prefix	L.suffix
L.maxsum	L.sum

R.prefix	R.suffix
R.maxsum	R.sum

분할 정복 + 세그먼트 트리

- 이제 세그먼트 트리의 각 노드에 앞에서 본 4가지 값을 넣어 봅시다.



분할 정복 + 세그먼트 트리

- 앞에서 봤던 “두 수열을 합치는 연산“을 더하기(+) 연산이라 하면, 점 갱신 / 구간 쿼리 세그먼트 트리와 정확히 같습니다!
  - 결합법칙은 성립하지만, 교환법칙은 성립하지 않음에 유의해야 합니다.

```
// 새롭게 정의한 Node 구조체.
// 앞에서 살펴본 4가지 값을 가지고 있음.
struct Node {
    int prefix, suffix, maxsum, sum;
};

Node tree[2 * SIZE];
```

```
// 새로 정의한 더하기 연산.
// max 함수는 <algorithm> header에 있음
Node operator+(Node l, Node r){
    return (Node){
        max(l.prefix, l.sum + r.prefix),
        max(r.suffix, r.sum + l.prefix),
        max(max(l.maxsum, r.maxsum),
            l.suffix + r.prefix)),
        l.sum + r.sum
    };
}
```

분할 정복 + 세그먼트 트리

- update 함수는 거의 유사하게 구현

```
// A[X] = V로 갱신.
void update(int X, int V){
    X += SIZE - 1;
    tree[X] = (Node){V, V, V, V};
    for(X /= 2; X >= 1; X /= 2){
        tree[X] = tree[2 * X] + tree[2 * X + 1];
    }
}
```

분할 정복 + 세그먼트 트리

- update 함수는 거의 유사하게 구현되는 반면, query 함수는 약간의 수정이 필요합니다 → 교환법칙을 만족하지 않아도 작동하도록!

```
// [L, R] 구간의 최대 연속 부분합 구하기.
// INF는 충분히 큰 값으로 미리 정의.
int query(int L, int R){
    L += SIZE - 1;
    R += SIZE - 1;
    Node lres = {-INF, -INF, -INF, 0};
    Node rres = {-INF, -INF, -INF, 0};
    for(; L <= R; L /= 2, R /= 2){
        if(L % 2 == 1) lres = lres + tree[L++];
        if(R % 2 == 0) rres = tree[R--] + rres;
    }
    return (lres + rres).maxsum;
}
```

그 밖의 다양한 응용

- 보통 세그먼트 트리의 각 노드가 뭘 담고 있는지에 대해서 응용합니다
  - 각 노드가 multiset<int> : 집합의 배열 S_1, S_2, …, S_N이 있을 때 S_i에 수 추가(삭제)하기 / S_L U S_{L+1} U … U S_R의 최소 원소 구하기
  - 각 노드가 해당 구간의 수열을 정렬해서 저장 (Merge Sort Tree) : 구간에서 k 이상인 수의 개수 세기
  - 각 노드가 또 하나의 세그먼트 트리 (2D Segment Tree) : 직사각형 영역의 점 세기 / 기타 여러 가지 쿼리
- 세그먼트 트리를 실시간으로 구축해 나갈 수도 있습니다
  - 2D 세그먼트 트리 / PST(Persistent Segment Tree) 등의 구현에 사용
  - 기본적으로 재귀 구현 / 자세한 구현 설명은 일단은 생략

감사합니다

Segment Tree (4/4)

Lazy Propagation

서울대학교 컴퓨터공학부 김동현

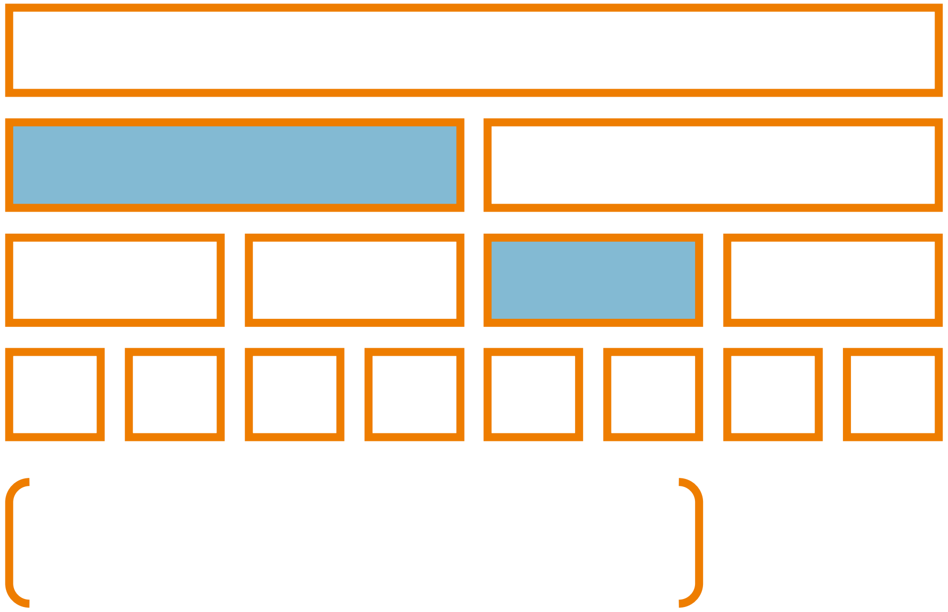
구간 갱신 + 구간 쿼리

- 점 갱신 + 구간 쿼리 / 구간 갱신 + 점 쿼리는 간단했습니다
- 구간 갱신 + 구간 쿼리도 빠르게 할 수 있을까요?
- 물론 할 수 있습니다!
  - 다만 좀 더 어렵습니다

기본적인 아이디어

- 재귀 구현을 기반으로 합니다
  - 비재귀 구현으로도 가능은 하나, 시간 복잡도에서 손해를 봅니다
- 갱신 또는 쿼리가 한 번 수행될 때 관여되는 노드는 O(logN)개입니다.
- 더 내려가지 않고 멈출 때 마다, 해당 노드들에 "표시"를 해 놓읍시다
- 나중에 "표시"를 다시 만나면, 더 내려가기 전에 "표시"에 대한 정보를 아래로 전파(Propagate) 해 줍니다.
- 전파를 그때그때 시키지 않고 필요할 때만 '게으르게' 한다고 해서 Lazy Propagation이라 부릅니다.

기본적인 아이디어



표시를 해 놓고

나중에 전파한다

구현

구간에 일정한 수 더하기 / 구간의 합 구하기

- 트리의 각 노드가 값 2개를 가집니다
  - 보통 배열 2개로 구현 (더 많아지면 구조체를 사용하기도…)
  - tree[] : 각 노드에 대해 실제로 구하는 값 (구간의 합)
  - lazy[] : 각 노드에 대한 '표시' (구간에 공통적으로 더해진 수)

구현

- propagate 함수
  - 내려가기 전에 표시를 전파하는 함수입니다.

```
// node번 노드 ([S, E] 구간을 담당) 에서 아래로 전파
void propagate(int node, int S, int E){
    int MID = (S + E) / 2;
    tree[2 * node] += (MID – S + 1) * lazy[node];
    lazy[2 * node] += lazy[node];
    tree[2 * node + 1] += (E – MID) * lazy[node];
    lazy[2 * node + 1] += lazy[node];
    lazy[node] = 0;
}
```

구현

- update 함수

```
// A[L], …, A[R]에 각각 V를 더함
// 초기 호출 : update(L, R, V, 1, 1, SIZE)
void update(int L, int R, int V, int node, int S, int E){
    if(R < S || E < L) return;
    if(L <= S && E <= R){
        tree[node] += (E - S + 1) * V;
        lazy[node] += V;
        return;
    }
    propagate(node, S, E);
    int MID = (S + E) / 2;
    update(L, R, V, 2 * node, S, MID);
    update(L, R, V, 2 * node + 1, MID + 1, E);
    tree[node] = tree[2 * node] + tree[2 * node + 1];
}
```

구현

- query 함수

```
// A[L] + … + A[R] 구하기
// 초기 호출 : query(L, R, 1, 1, SIZE)
int query(int L, int R, int V, int node, int S, int E){
    if(R < S || E < L) return;
    if(L <= S && E <= R) return tree[node];
    propagate(node, S, E);
    int MID = (S + E) / 2;
    return query(L, R, 2 * node, S, MID) +
           query(L, R, 2 * node + 1, MID + 1, E);
}
```

응용

구간에 일정한 수 더하기 / 구간의 최솟값(최댓값) 구하기

- 앞에서 살펴본 구현과 매우 유사합니다.
  정의만 바꾸어서 그대로 구현하면 됩니다.
- 약간만 더 응용하면, 구간에서 최솟값(최댓값)의 개수까지 같이 구할 수 있습니다.
  - 이 연산을 지원하는 세그먼트 트리로 풀 수 있는 문제에는
    직사각형 N개의 합집합 넓이 구하기 등이 있습니다.

응용

구간에 일정한 수 더하기 / 구간에 일정한 수 곱하기 /
구간의 합 구하기

- lazy에 해당하는 값을 두 개로 쪼개서, 각각
  공통적으로 곱해진 수 / 공통적으로 더해진 수를 관리하게 하면 됩니다.
- propagate 함수를 조금 더 신경써서 구현해야 합니다.

```
tree[2 * node] = tree[2 * node] * mul[node] + (MID – S + 1) * add[node];
mul[2 * node] *= mul[node];
add[2 * node] = add[2 * node] * mul[node] + add[node];
```

감사합니다