Data Structures - 2

자료구조 – 2

競技プログラミングの鉄則 – 米田優峻

KPSC Algorithm Study 25/2/13 Thu.

by Haru_101

Data Structures

- 이번 시간엔 데이터를 다루는 자료구조, 그리고 쿼리와 관련된 문제들을 풀어보겠습니다.

Data Structures

- 다음 문제를 풀어봅시다. (https://atcoder.jp/contests/tessoku-book/tasks/tessoku_book_be)

問題文

N 個の穴がある砂場に、一匹のアリが住んでいます。このアリは規則的な動きをすることが知られており、穴 i（$1 \leq i \leq N$）に入った翌日には穴 A_i に移動します。

それについて、以下の Q 個のクエリを処理してください。

- j 個目のクエリ：いま穴 X_j にいるとき、Y_j 日後にはどの穴にいるか？

制約

- 入力はすべて整数である
- $1 \leq N \leq 100000$
- $1 \leq Q \leq 100000$
- $1 \leq A_i \leq N$
- $1 \leq X_j \leq N$
- $1 \leq Y_j \leq 10^9$

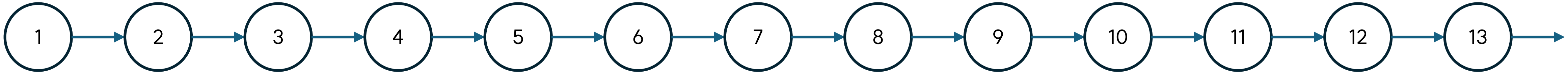
- N개의 구멍이 있는 모래사장에 한 마리의 개미가 살고 있습니다. 이 개미는 규칙적으로 이동하는 것이 알려져 있으며, 구멍 i 에 있는 개미는 다음날 구멍 A_i로 이동합니다.
- 이때, 다음 Q개의 쿼리를 수행하는 프로그램을 작성하세요.
  - 쿼리 j : 지금 구멍 X_j에 있는 개미는, Y_j일 뒤에 어떤 구멍으로 이동하는가?

Data Structures

- 단순하게, $f(i) = A_i$라고 하고, $f \dots f(f(i))$의 값을 찾으면 될 것 같습니다.
- 하지만, $N \le 100{,}000$이고, $Q \le 100{,}000$이므로 $O(NQ)$의 시간에 해결하지 못할 것 같습니다.
- 어떻게 최적화를 할 수 있을까요?

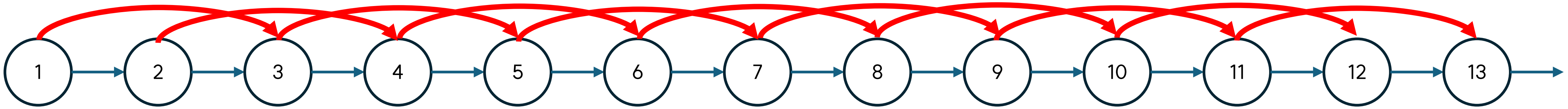
Data Structures

- 아래 그림을 한번 봐 봅시다.



Data Structures

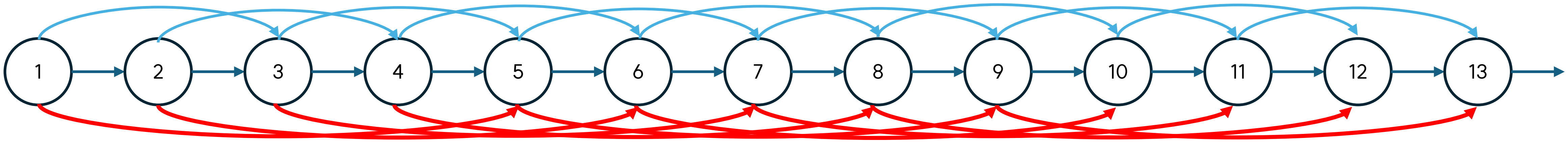
- 이 그림에서 $f(1) = 2$, $f(2) = 3$임을 알 수 있고, $f\big(f(1)\big) = 3$임을 알 수 있습니다.
- 그렇다면, $f(f(1))$을 알기 위해선, 1 -> 어딘가, 어딘가 -> 3의 정보만 알면 된다는 것을 알 수 있습니다.
- $f\big(f(i)\big) = \mathrm{g}$라는 함수로 새롭게 써보고 그림에 표시해봅시다.



- 빨간색 선분은 1일치 + 1일치 = 2일치의 이동을 한번에 표현해주는 역할을 하게 됩니다.

Data Structures

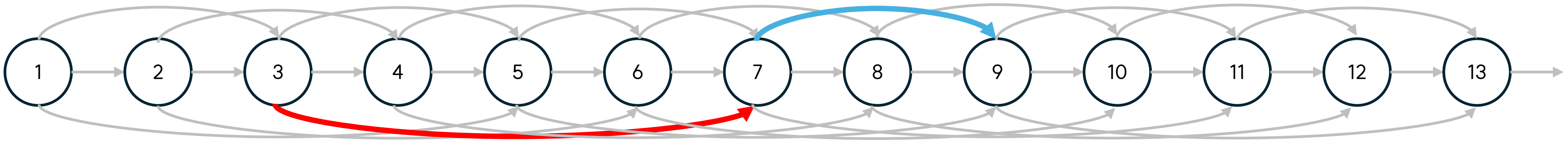
- 이제 이 그림에서 $g(1) = 3$, $g(3) = 5$임을 알 수 있고, $g\big(g(1)\big) = 5$임을 알 수 있습니다.
- 그렇다면, $g(g(1))$을 알기 위해선, 1 -> 어딘가, 어딘가 -> 5의 정보만 알면 된다는 것을 알 수 있습니다.
- $g\big(g(i)\big) = h$라는 함수로 새롭게 써보고 그림에 표시해봅시다.



- 빨간색 선분은 2일치 + 2일치 = 4일치의 이동을 한번에 표현해주는 역할을 하게 됩니다.

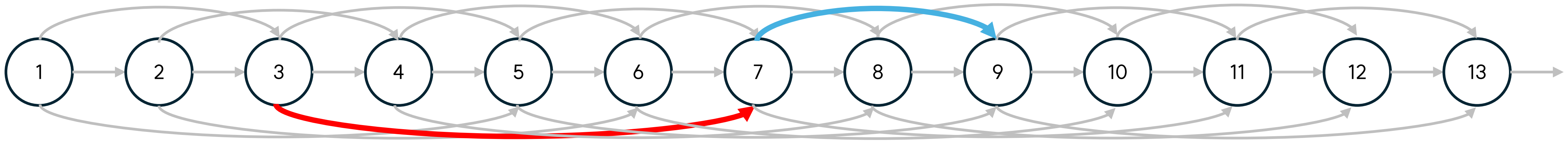
Data Structures

- 그렇다면 $i = 3$이고, 6일뒤 위치는 어디인지 어떻게 알 수 있을까요?
- 6을 이진수로 나타내면, 110_2이고, 이는 각각 4일치 이동과 2일치 이동이 필요함을 의미합니다.
- 따라서 4일치 이동을 하고 난 후, 2일치 이동을 하면 정답을 빠르게 알 수 있습니다.



Data Structures

- 즉, 이동 일 수를 나타내는 Y_j를 이진수로 나타내고, 각 비트마다 1이 있다면 2^k일치의 이동을 수행하면 됩니다.
- 2^k일치 이동은 DP테이블을 이용해서 (DP[k][x] = x에서 2^k일치 이동을 했을 때 도착점)을 전처리 후 쿼리를 수행하면 됩니다.



Data Structures

- 코드로 나타내면 다음과 같습니다. (i 범위 주의)

```
int dp[32][100005];
int main() {
    fastio();
    int N, Q;
    cin >> N >> Q;
    for(int i=1; i<=N; i++) {
        cin >> dp[1][i];
    }
    for(int k=2; k<=31; k++) {
        for(int i=1; i<=N; i++) {
            int temp = dp[k-1][i];
            int next = dp[k-1][temp];
            dp[k][i] = next;
        }
    }
    while(Q--) {
        int x, y;
        cin >> x >> y;
        int now = x;
        int gob = 1;
        while(y!=0) {
            if(y%2==1) {
                now = dp[gob][now];
            }
            y = y/2;
            gob+=1;
        }
        cout << now << '\n';
    }
}
```

Data Structures

- 다음 문제를 풀어봅시다. (https://atcoder.jp/contests/tessoku-book/tasks/tessoku_book_bg)

問題文

長さ N の数列 $A = (A_1, A_2, \ldots, A_N)$ があり、最初はすべての要素が 0 になっています。以下の 2 種類のクエリを処理してください。

- **クエリ 1**：A_{pos} の値を x に更新する。
- **クエリ 2**：$A_l, A_{l+1}, \ldots, A_{r-1}$ の合計値を答える。

ただし、与えられるクエリの数は全部で Q 個であるとします。

制約

- 入力はすべて整数である
- $1 \leq N \leq 100000$
- $1 \leq Q \leq 100000$
- $1 \leq \mathrm{pos} \leq N$
- $0 \leq x \leq 1000$
- $1 \leq l < r \leq N+1$

- 길이 N의 수열 $A = (A_1, A_2, \cdots, A_N)$가 있고, 처음엔 모든 원소가 0입니다. 아래 두 종류의 쿼리 Q개를 수행하는 프로그램을 작성하세요.
  - 쿼리 1 : A_{pos}를 x로 바꾼다.
  - 쿼리 2 : $A_l, A_{l+1}, \cdots, A_{r-1}$의 합을 출력한다.

Data Structures

- 예전에 배웠던 누적합 알고리즘을 생각해보면, 쿼리 2정도는 해결을 손쉽게 할 수 있을 것 같습니다.
- 하지만, 쿼리 1에 의해 중간중간 값이 계속 바뀌고 이에 따라 누적합을 계속 변경해줘야 합니다.
- 그렇다면 누적합 말고 어떻게 접근하는 것이 좋을까요?

Data Structures

- 값이 계속 변하고 이런 구간에 대한 쿼리를 처리하기 좋은 자료구조인 세그먼트 트리를 소개합니다.
- 세그먼트 트리는 기본적으로 이진트리라고 가정합니다.
- 원소의 개수가 $N = 10$이라고 가정하고, 세그먼트 트리를 만들어 보겠습니다.

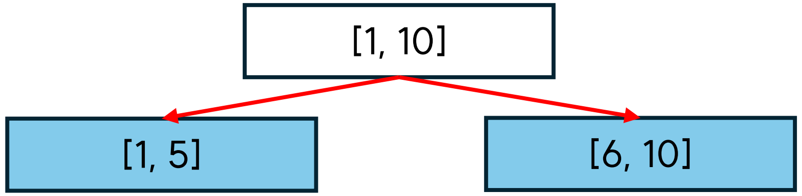
Data Structures

- 먼저 세그먼트 트리의 루트 노드 (최상위 노드)는 모든 원소를 포함하는 구간을 관리하는 노드입니다.
- 각 노드에 있는 표기는 $[a, b]$라고 쓰고, $A_a, A_{a+1}, \cdots, A_b$ 원소를 관리하는 노드라는 뜻입니다.

[1, 10]

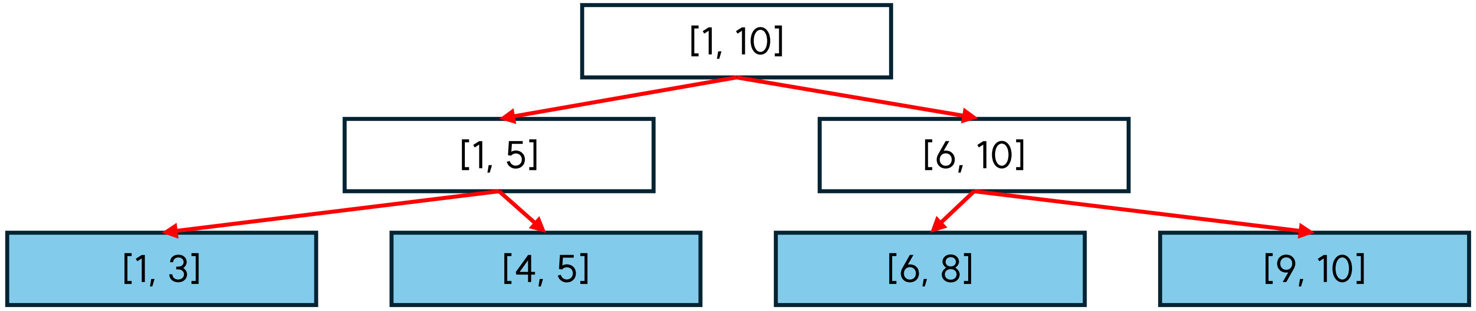
Data Structures

- 이제 각 노드마다 구간에 포함된 원소의 개수가 2 이상일때까지 자식 노드를 만들어줍시다.
- 현재 노드의 구간이 $[l, r]$일때, $m = \frac{l+r}{2}$ (소숫점 버림)을 구하고, 왼쪽 자식 노드는 $[l, m]$, 오른쪽 자식 노드는 $[m+1, r]$을 관리하게 노드를 만들어 줍니다.
- 이렇게 만들게 되면, $[1, 10]$의 구간합은 $[1, 5]$의 구간합 + $[6, 10]$의 구간합과 동일하게 됩니다.



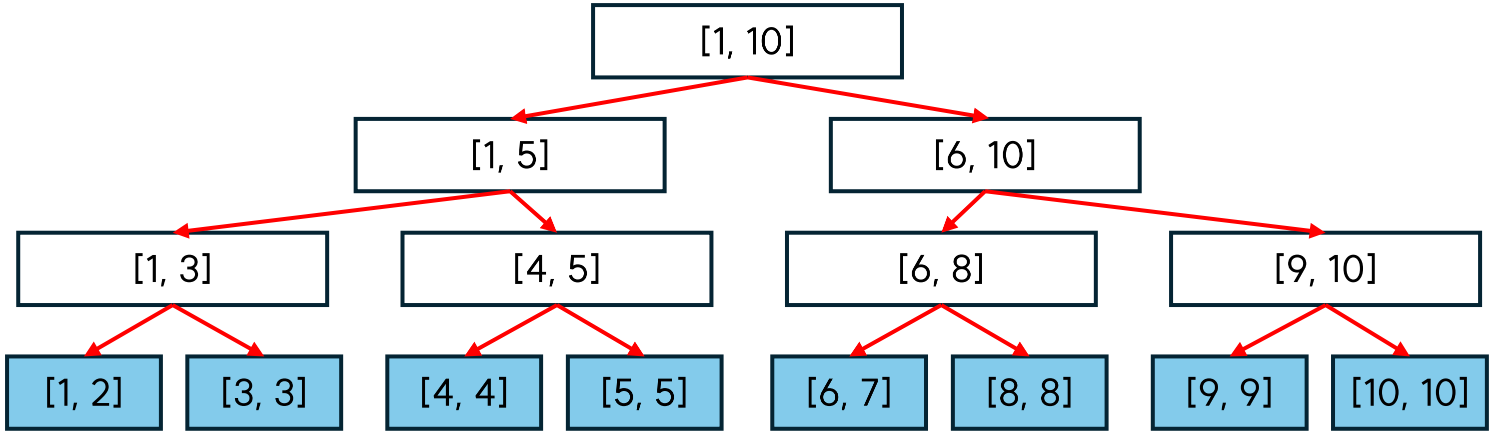
Data Structures

- $[1,5]$의 구간합은 $[1,3]$의 구간합 + $[4,5]$의 구간합이 되고, $[6,10]$의 구간합은 $[6,8]$의 구간합 + $[9,10]$의 구간합이 됩니다.



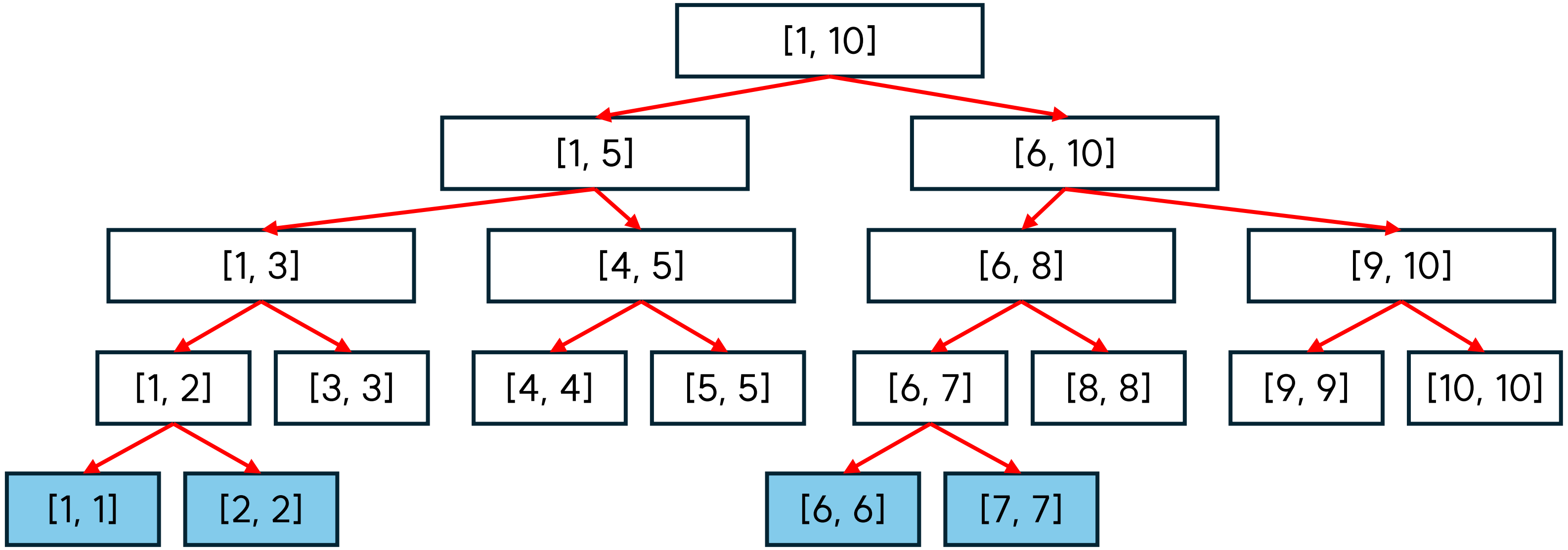
Data Structures

- $[1, 3]$의 구간합은 $[1, 2]$의 구간합 + $[3, 3]$의 구간합이 되고, $[4, 5]$의 구간합은 $[4, 4]$의 구간합 + $[5, 5]$의 구간합이 됩니다.
- $[6, 10]$에 대해서도 동일하게 적용해주면 됩니다.



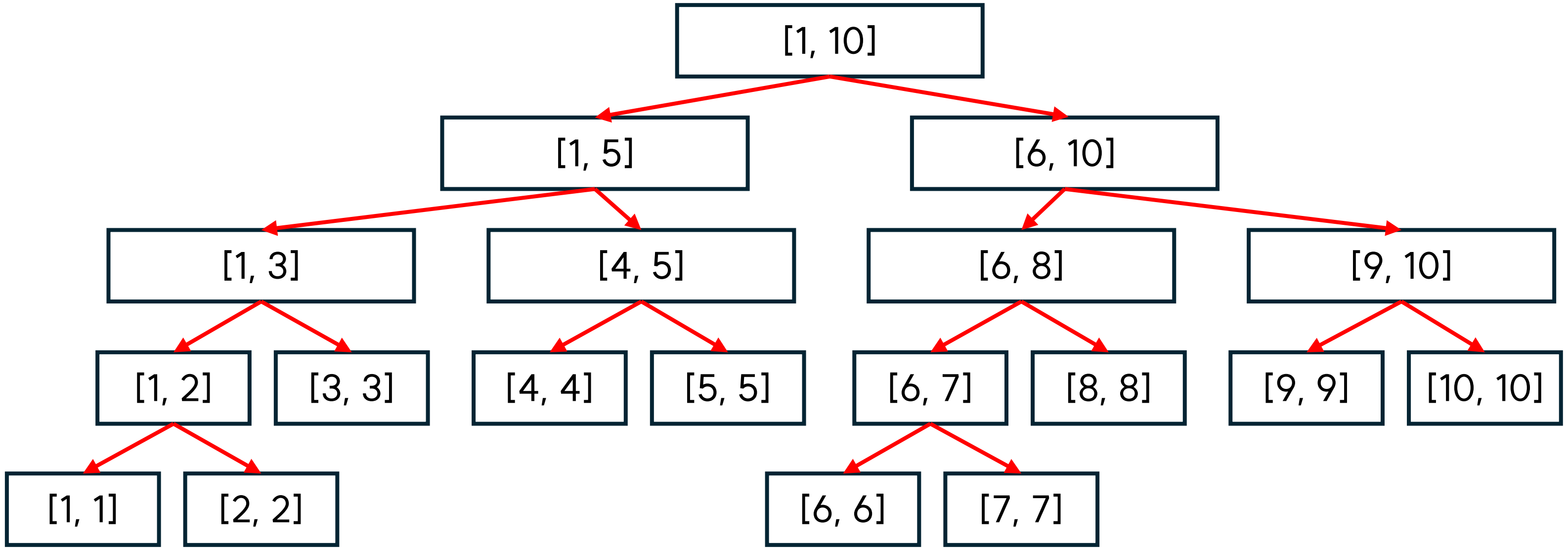
Data Structures

- 이제 구간의 길이가 2 이상인 구간에 대해 다시 한번 분할을 해줍니다.



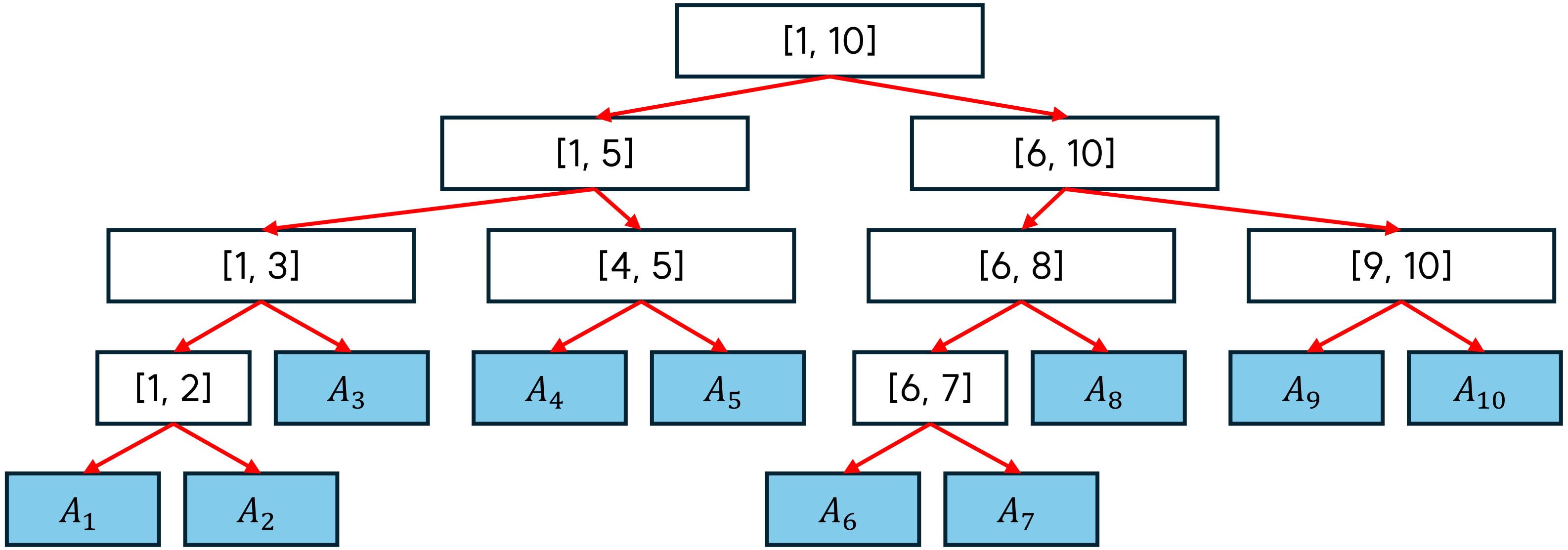
Data Structures

- 이제 세그먼트 트리의 기본적인 구조가 완성되었습니다.



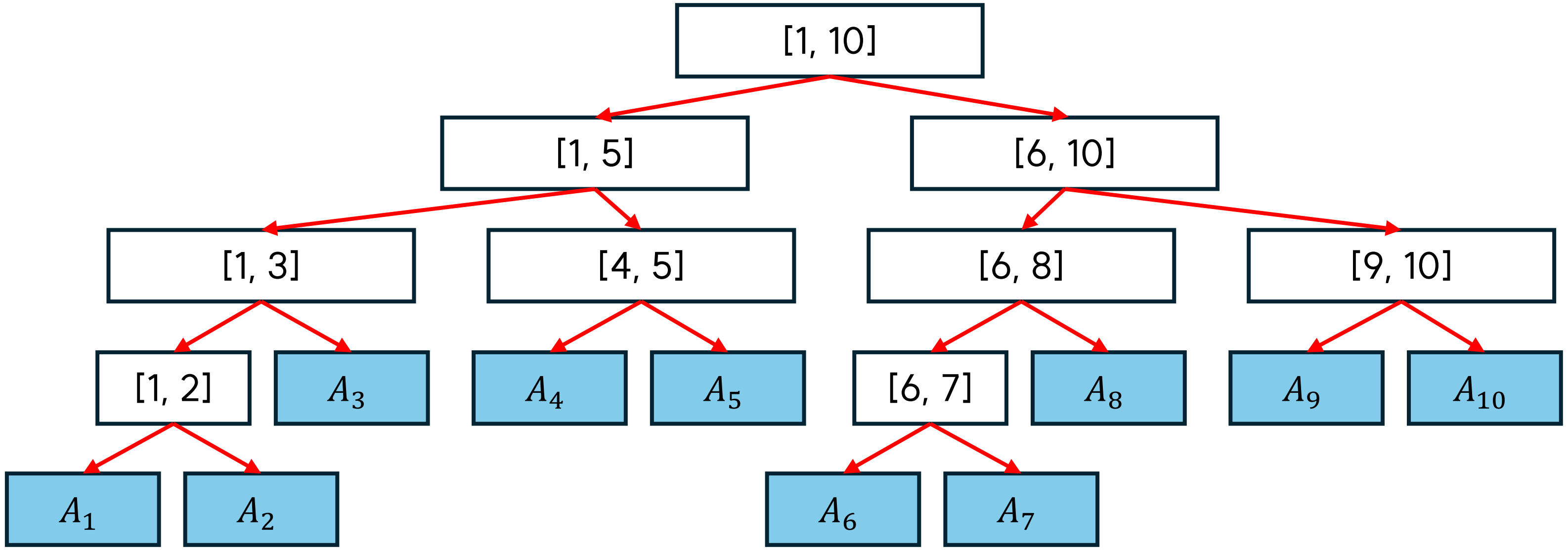
Data Structures

- 구간의 길이가 1인 $[l, l]$노드의 구간합은 A_l과 같기 때문에 이 노드에 A_l $(l = 1, 2, \cdots, N)$를 저장해주면 됩니다.



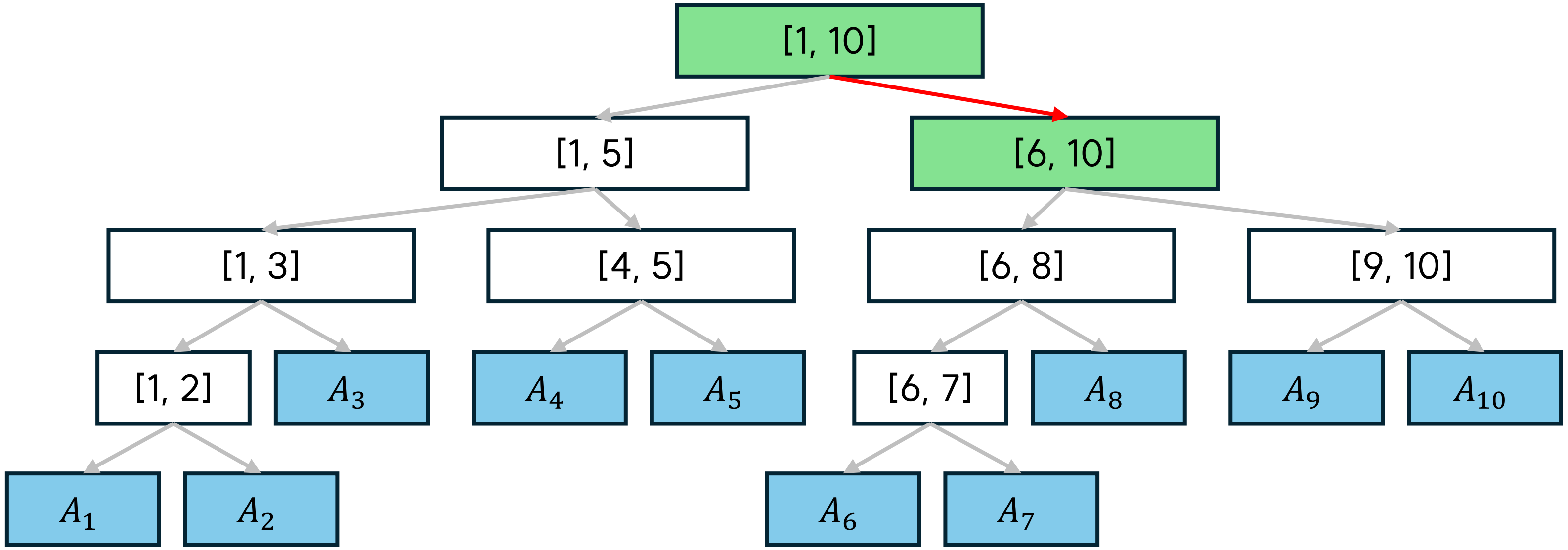
Data Structures

- 이제 쿼리 1에 대해서 어떻게 값을 변경해야 하는지 한번 살펴보겠습니다.
- 먼저 $A_6 = 10$으로 바꾼다라고 가정하고 살펴보겠습니다.



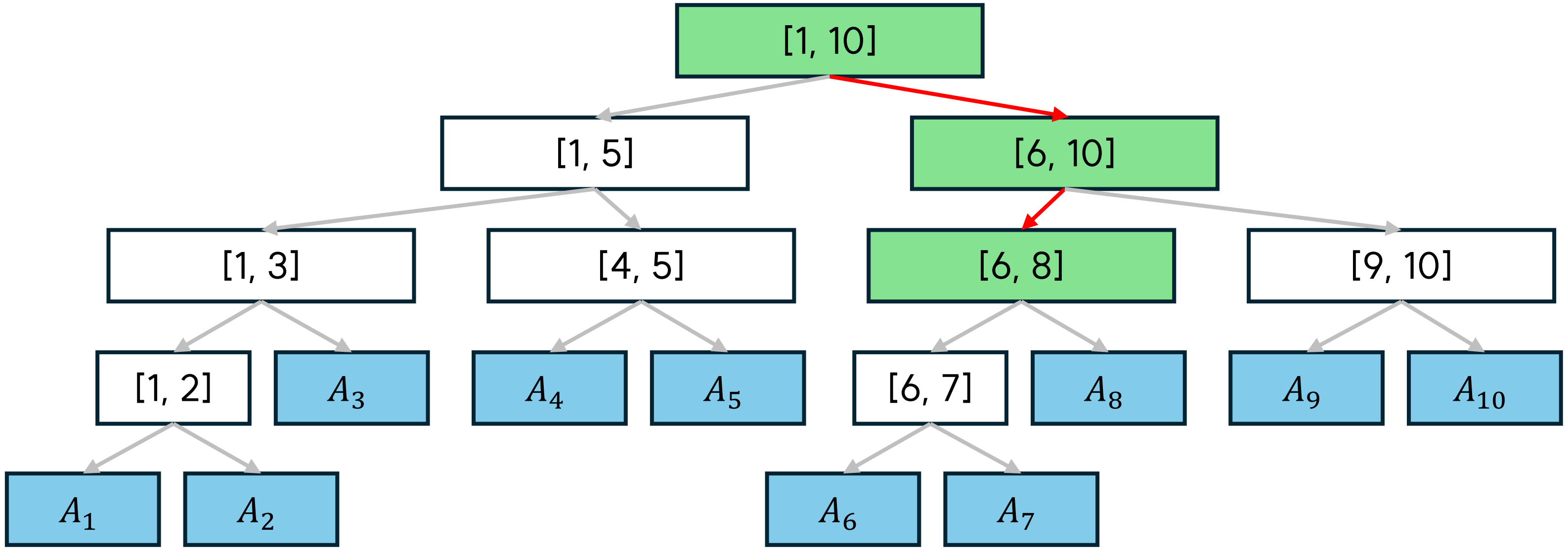
Data Structures

- 먼저 최상위 노드에서 출발을 합니다.
- 최상위 노드의 구간의 길이가 1이 아니기 때문에 (= 단일 원소를 저장하는 노드가 아니기 때문에) 자식 노드로 탐색 범위를 넓힙니다.
- 6은 $[1, 5]$구간에 없고 $[6, 10]$ 구간에 있으므로 오른쪽 자식 노드로 이동합니다.



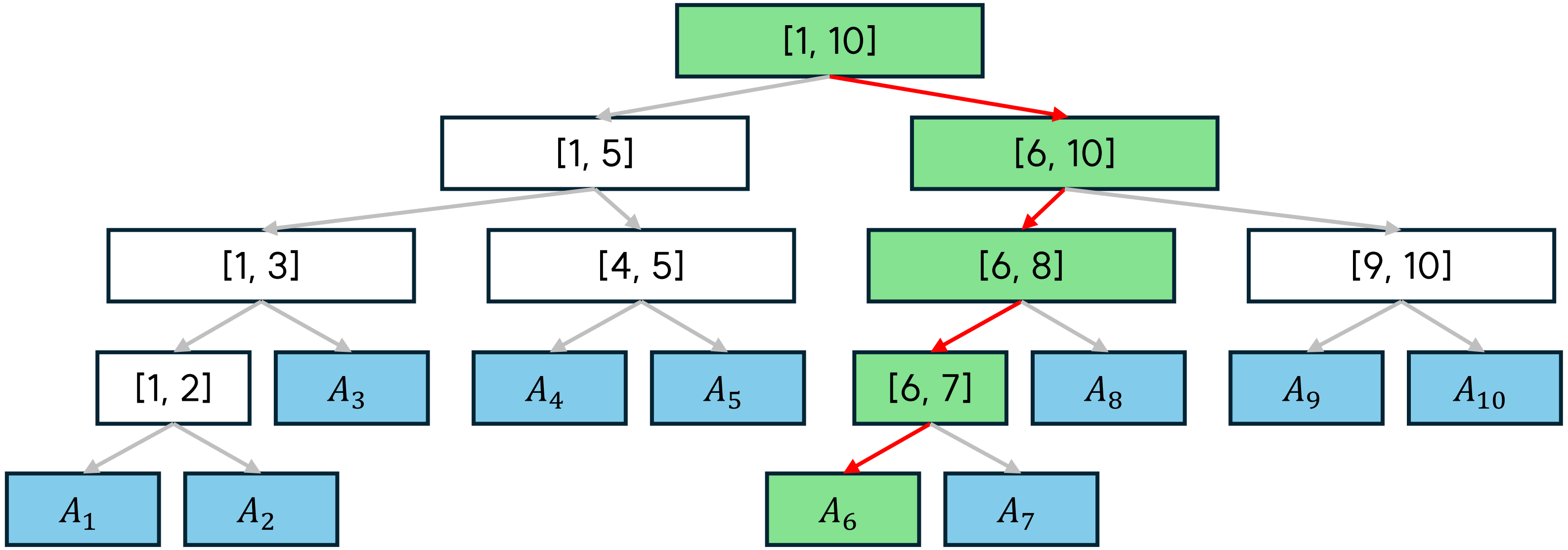
Data Structures

- 그 다음, $[6, 10]$ 노드에 대해서도 방금 했던 과정을 반복합니다.
- 6은 $[6, 8]$ 구간에 있으므로 왼쪽 노드로 이동합니다.



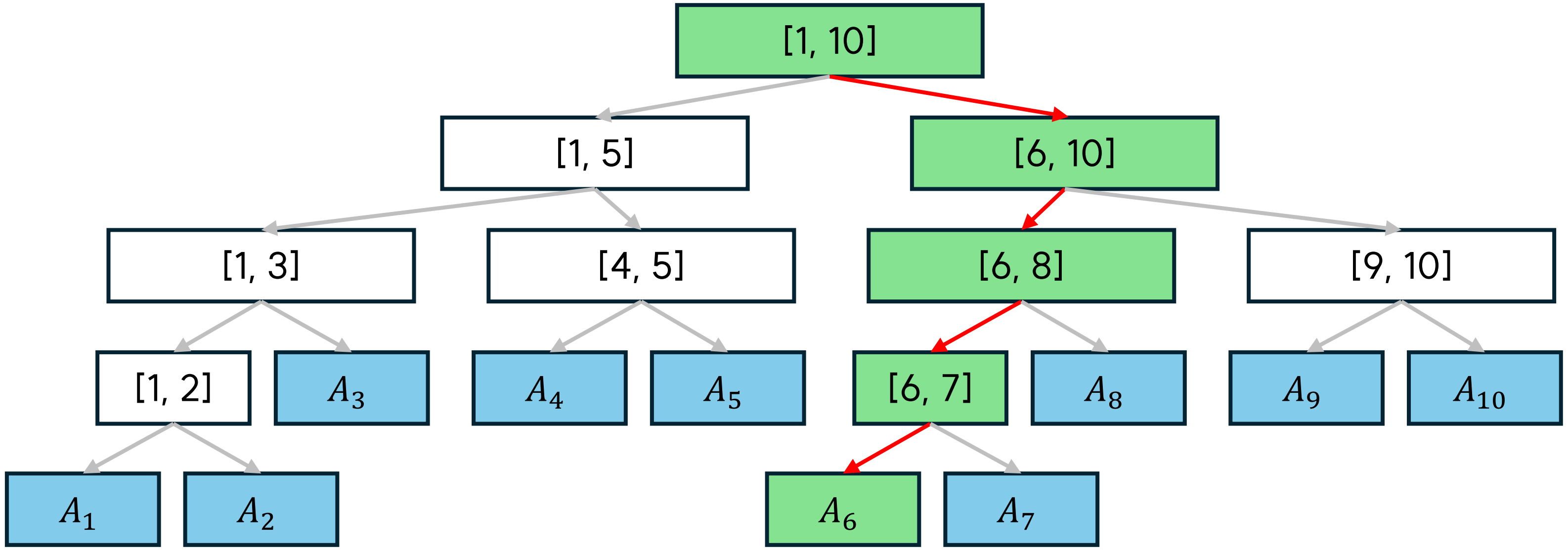
Data Structures

- 그 다음, $[6, 6]$ 노드에 대해서도 방금 했던 과정을 반복합니다.
- 6은 $[6, 6]$ 구간에 있으므로 왼쪽 노드로 이동합니다.



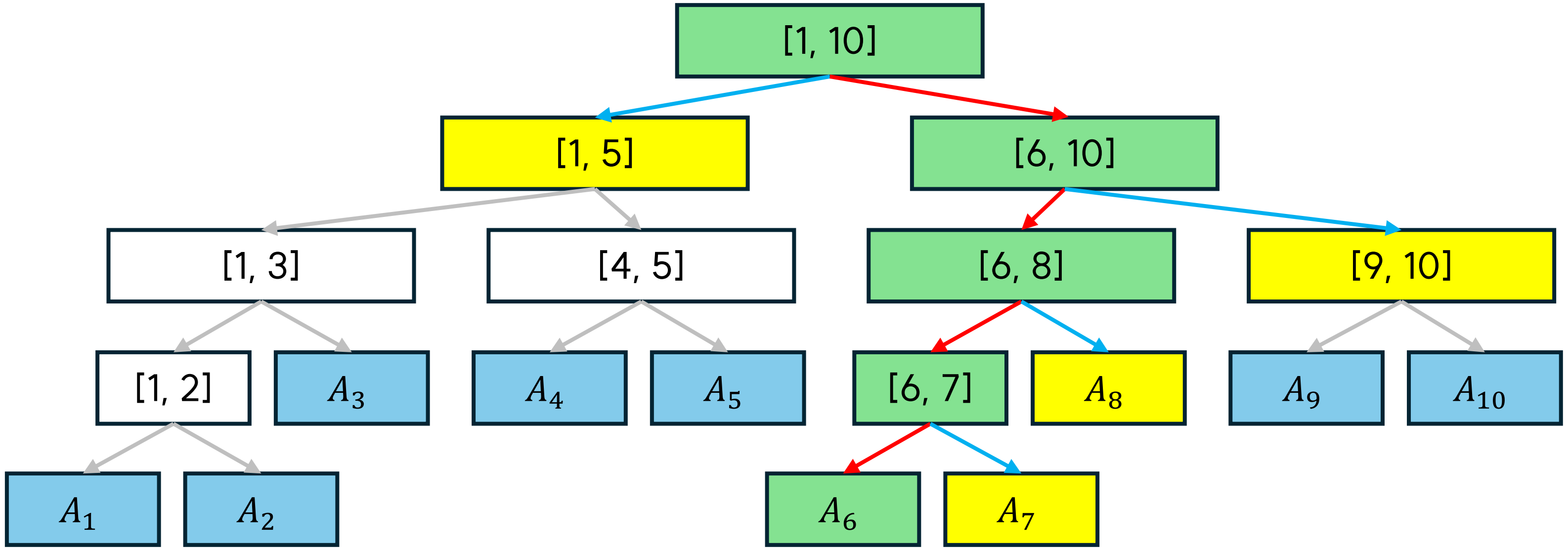
Data Structures

- 이제 구간의 길이가 1이므로 해당 노드의 값을 10으로 바꿉니다.
- 하지만, 단일 원소의 값만 바꾸었을 뿐, 그 원소를 포함하는 구간들의 합은 업데이트 하지 않은 상태입니다.
- 따라서, 초록색으로 표시한 구간들의 값도 바꿔줘야 합니다.



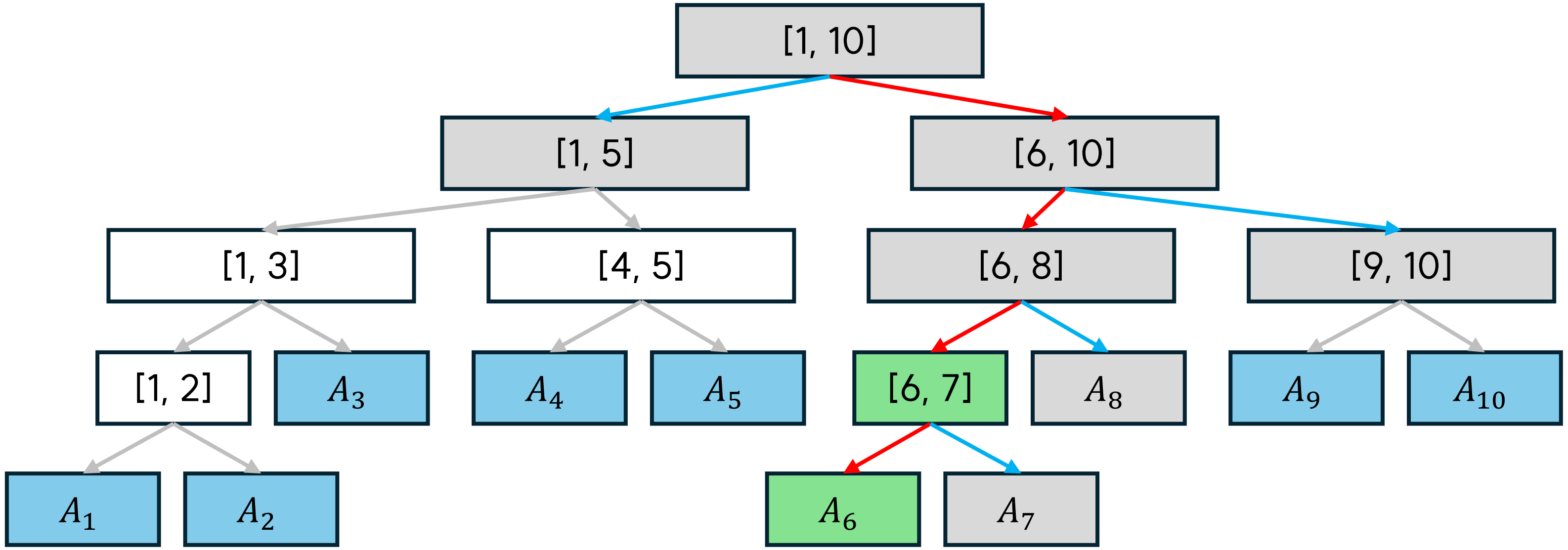
Data Structures

- 즉, 방문하지 않은 노드도 방문을 해서 해당 노드의 값을 가져와서 구간의 값을 다시 업데이트를 해줘야합니다.
- 하지만, 업데이트한 원소의 값이 포함되어 있지 않은 구간 노드는 해당 노드의 자식 노드까지 더 탐색할 필요가 없으므로 단순히 해당 노드의 값을 리턴해주면 됩니다.



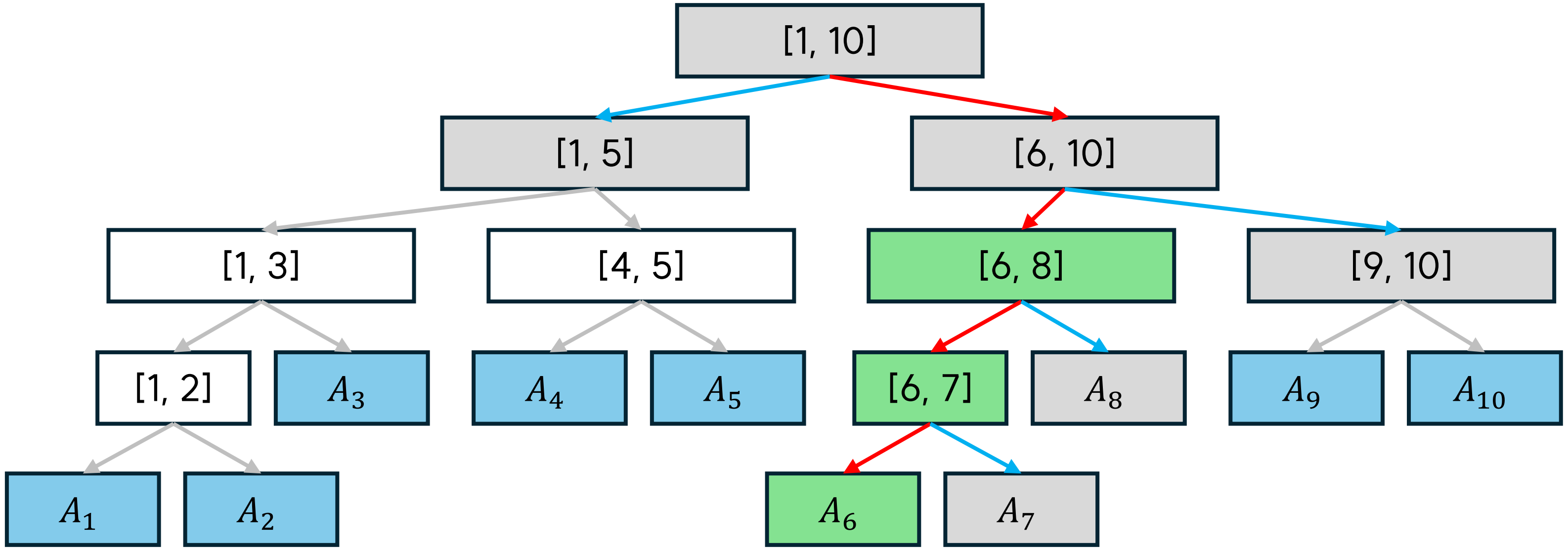
Data Structures

- 이제 방문한 순서의 역으로 구간합 업데이트를 수행해주면 됩니다.



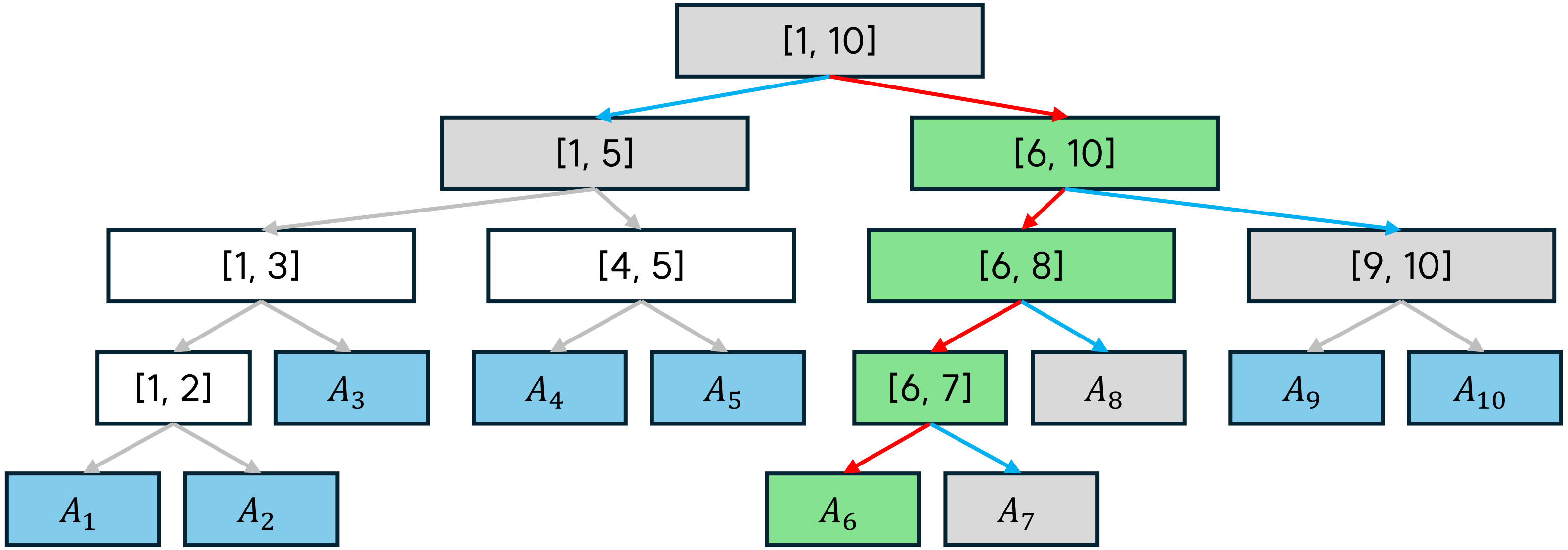
Data Structures

- 이제 방문한 순서의 역으로 구간합 업데이트를 수행해주면 됩니다.



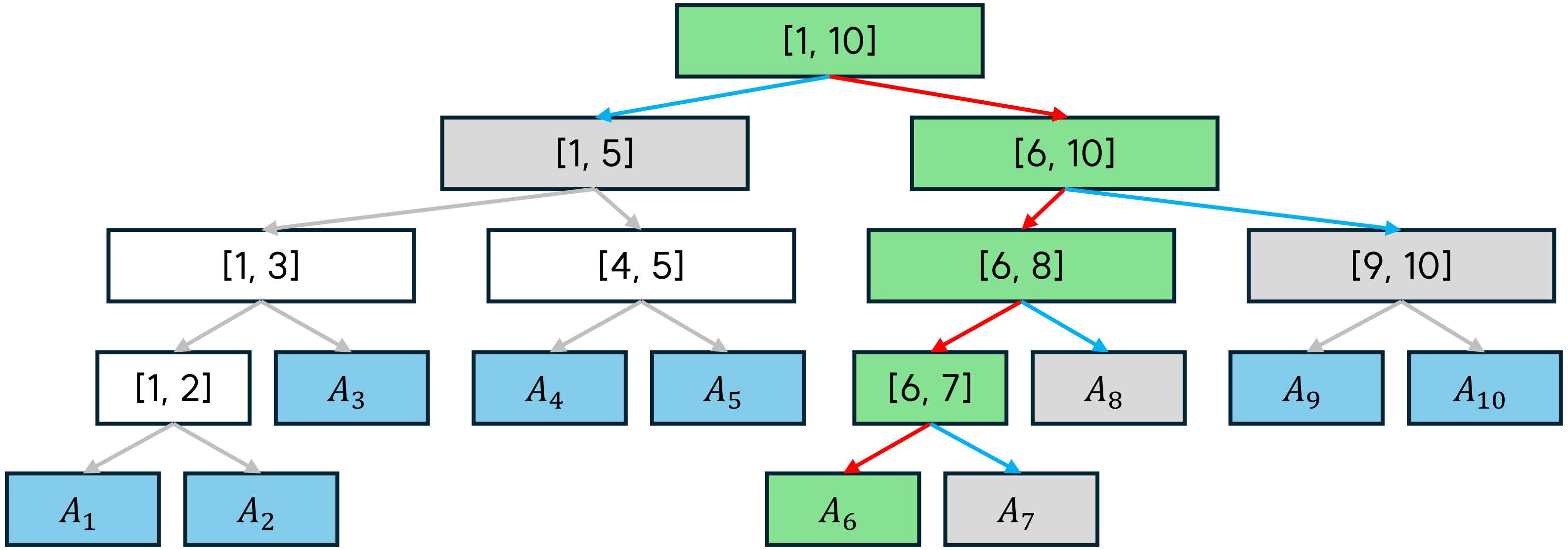
Data Structures

- 이제 방문한 순서의 역으로 구간합 업데이트를 수행해주면 됩니다.



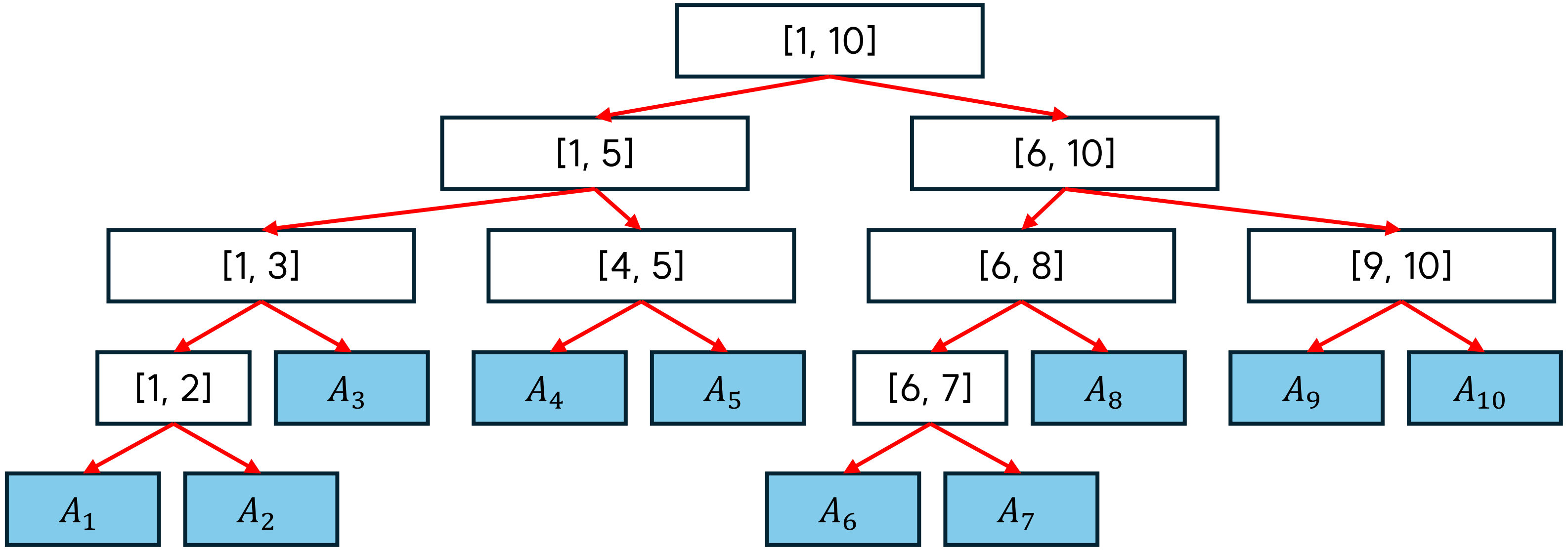
Data Structures

- 이제 방문한 순서의 역으로 구간합 업데이트를 수행해주면 됩니다.



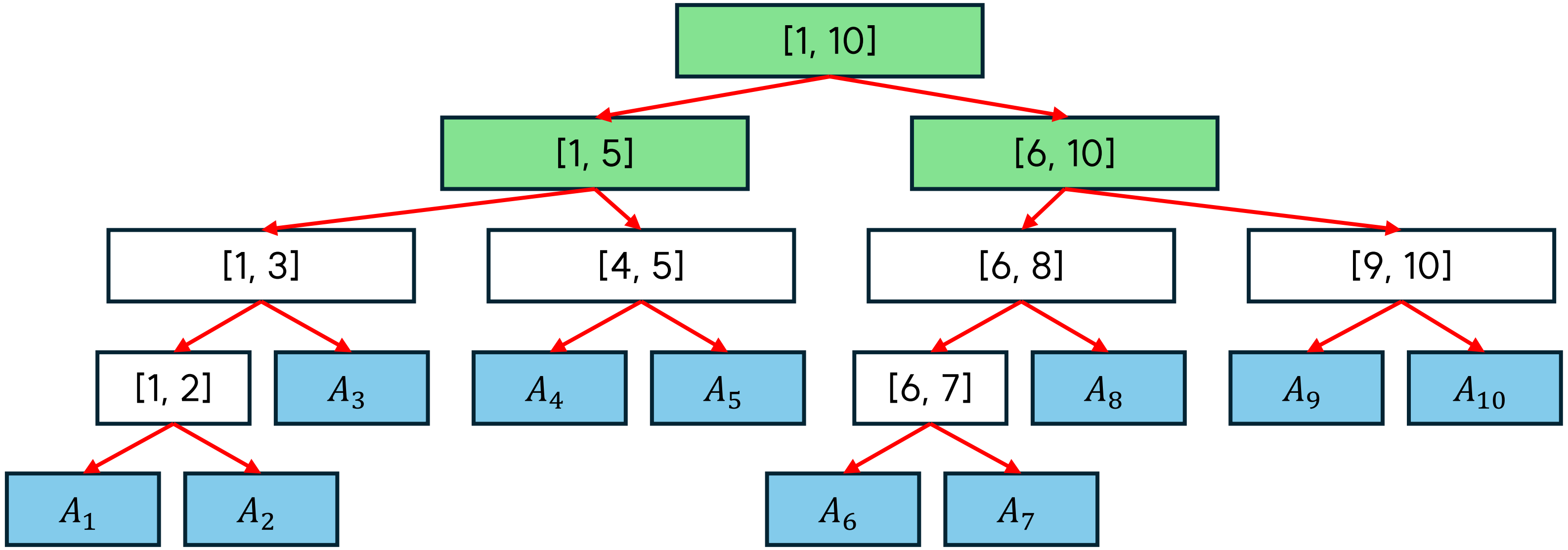
Data Structures

- 이제 쿼리 2에 대해서 어떻게 값을 변경해야 하는지 한번 살펴보겠습니다.
- 먼저 구간 $[2,7]$의 구간합을 구한다고 가정하고 살펴보겠습니다. $(L = 2, R = 7)$



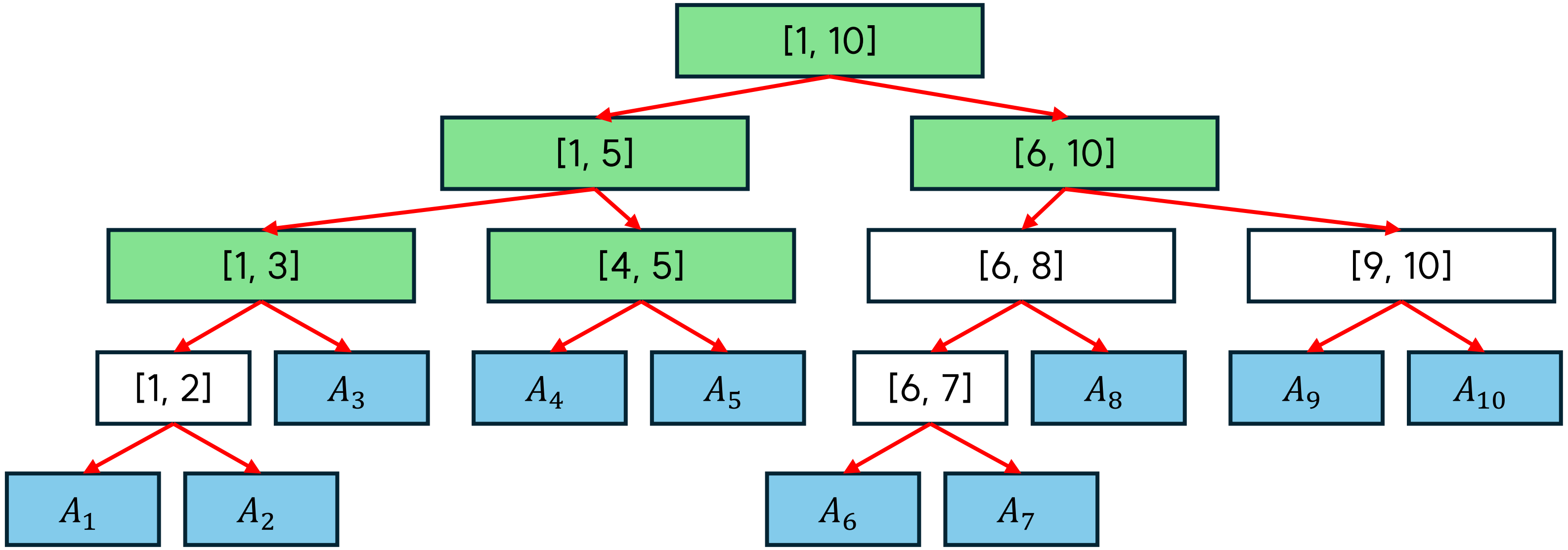
Data Structures

- 일단은 최상위 노드에서 시작을 합니다.
- 그 다음, $[l, r] = [1, 10]$과 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 일단, 겹치는 구간의 크기가 1 이상이지만, $[l, r]$이 $[L, R]$안에 속하지 않으므로 자식 노드로 한번 더 이동합니다.
  ($L \leq l$ 이면서 $r \leq R$)



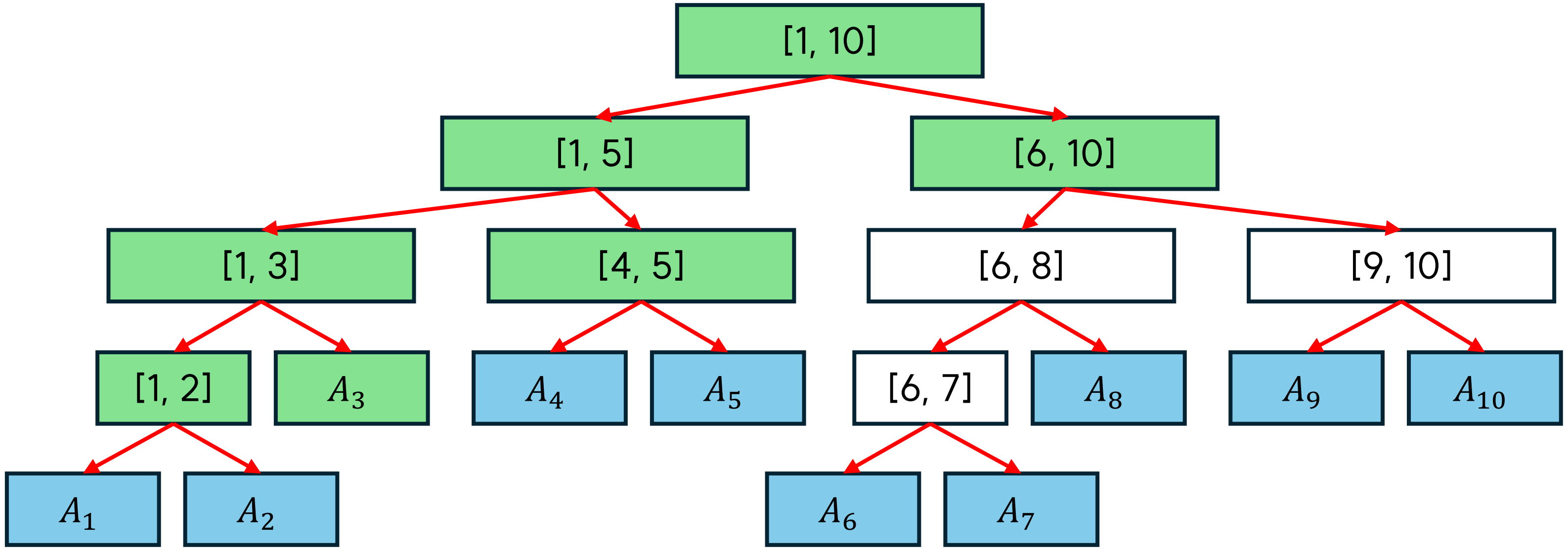
Data Structures

- 그 다음, $[l, r] = [1, 5]$와 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 일단, 겹치는 구간의 크기가 1 이상이지만, $[l, r]$이 $[L, R]$안에 속하지 않으므로 자식 노드로 한번 더 이동합니다.
  ($L \leq l$ 이면서 $r \leq R$)



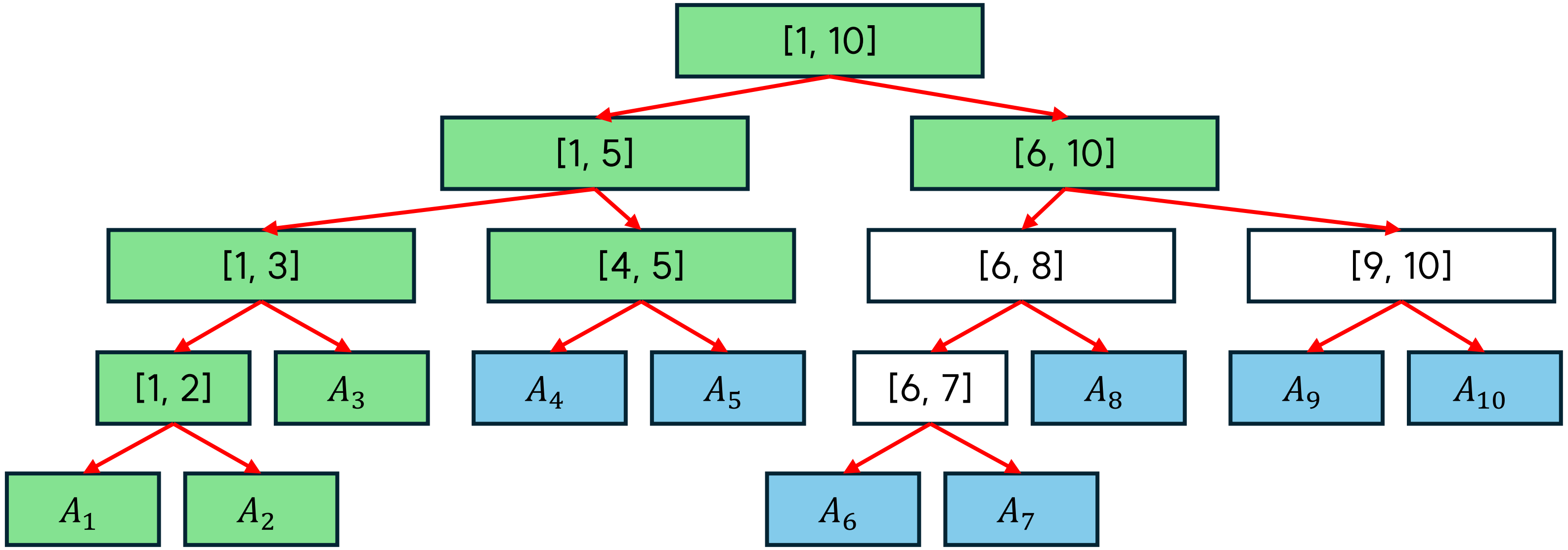
Data Structures

- 그 다음, $[l, r] = [1, 3]$과 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 일단, 겹치는 구간의 크기가 1 이상이지만, $[l, r]$이 $[L, R]$안에 속하지 않으므로 자식 노드로 한번 더 이동합니다. ($L \le l$ 이면서 $r \le R$)



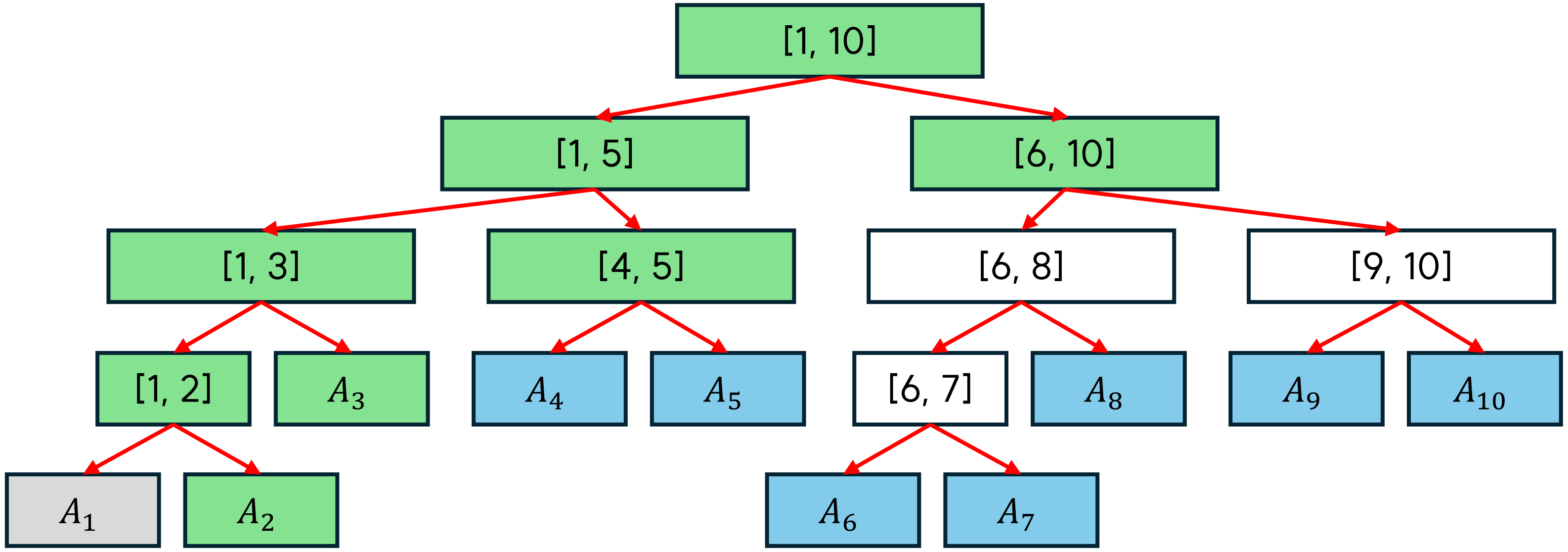
Data Structures

- 그 다음, $[l, r] = [1, 2]$와 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 일단, 겹치는 구간의 크기가 1 이상이지만, $[l, r]$이 $[L, R]$안에 속하지 않으므로 자식 노드로 한번 더 이동합니다. ($L \leq l$ 이면서 $r \leq R$)



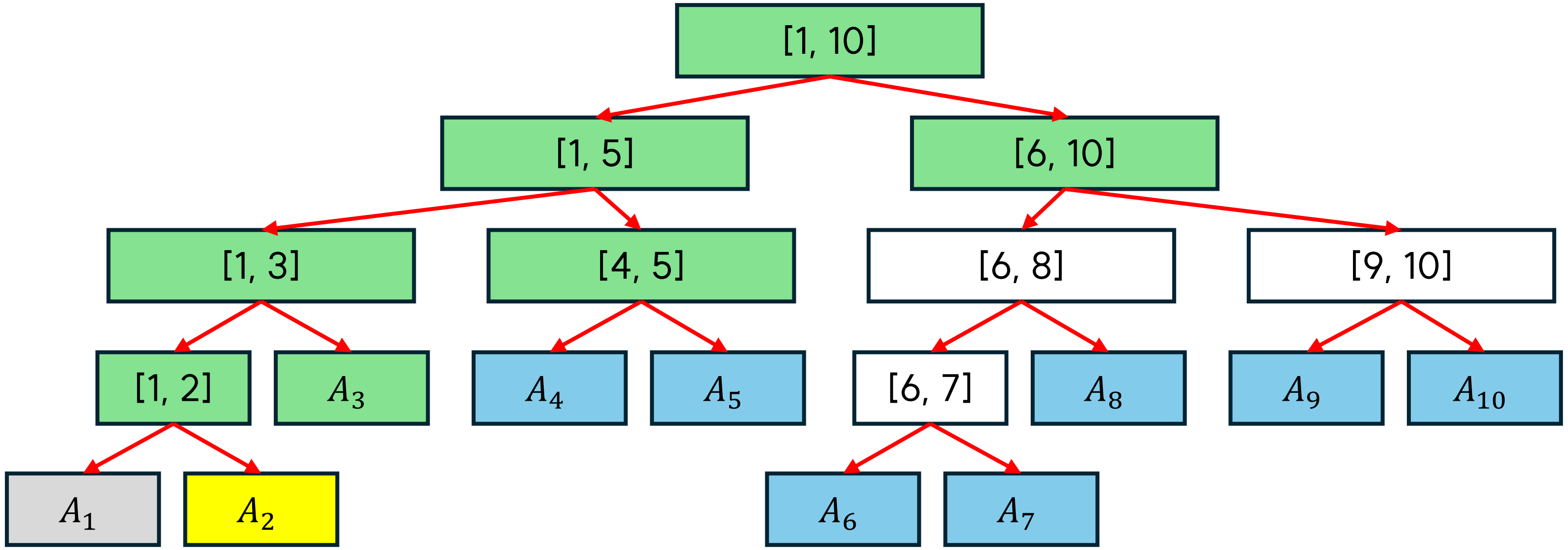
Data Structures

- 그 다음, $[l, r] = [1, 1] = A_1$과 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 겹치는 부분이 없습니다. 따라서 리턴값에 0을 더합니다.



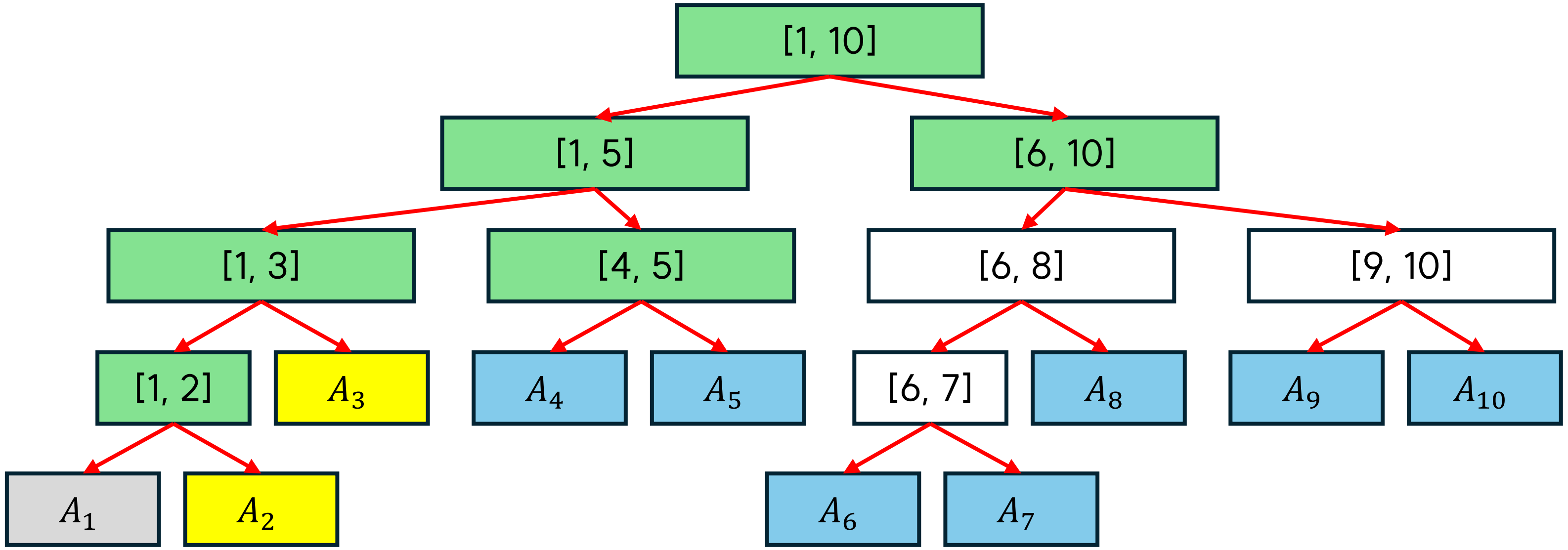
Data Structures

- 그 다음, $[l, r] = [2, 2] = A_2$와 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 겹치는 구간의 크기가 1 이상이고, $[l, r]$이 $[L, R]$에 속하므로 리턴값에 $[l, r]$노드의 값을 더합니다.



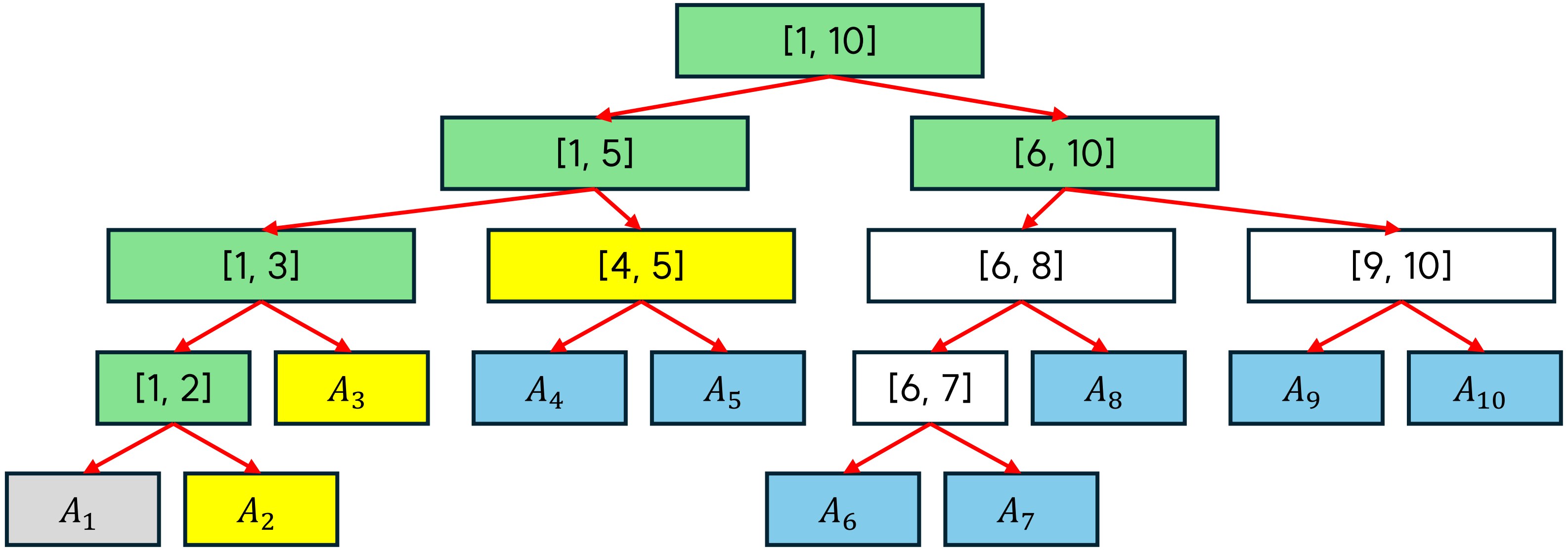
Data Structures

- 그 다음, $[l, r] = [3, 3] = A_3$와 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 겹치는 구간의 크기가 1 이상이고, $[l, r]$이 $[L, R]$에 속하므로 리턴값에 $[l, r]$노드의 값을 더합니다.



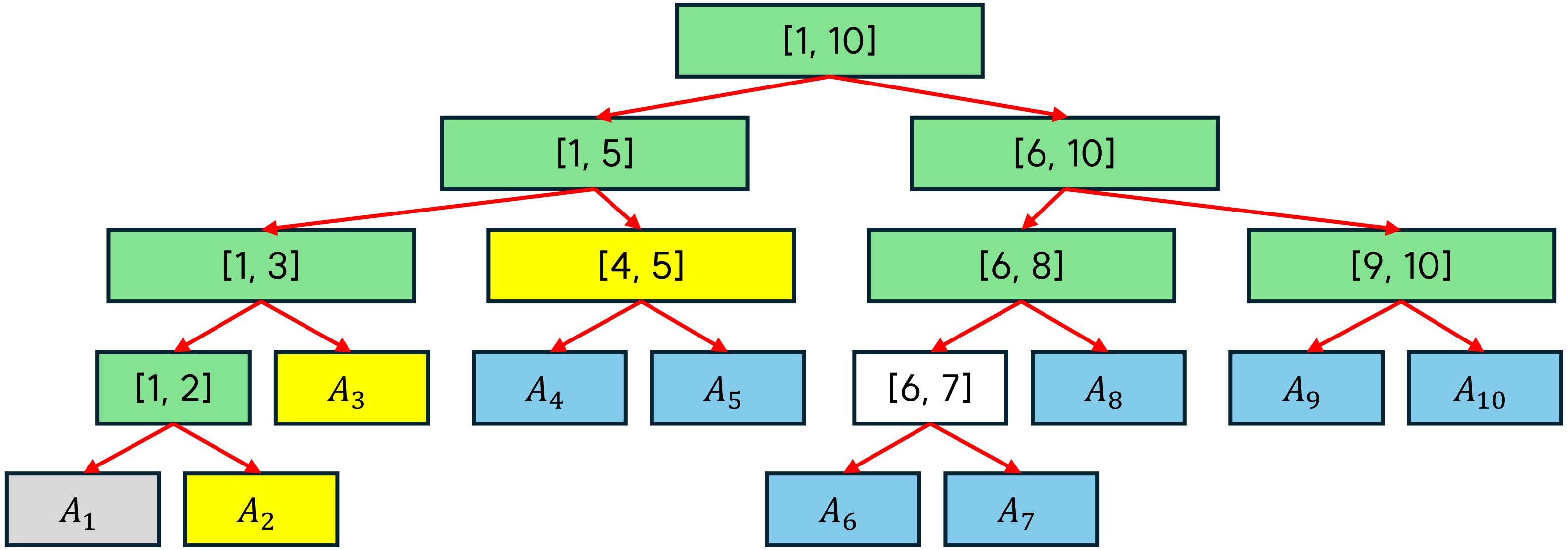
Data Structures

- 그 다음, $[l, r] = [4, 5]$와 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 겹치는 구간의 크기가 1 이상이고, $[l, r]$이 $[L, R]$에 속하므로 리턴값에 $[l, r]$노드의 값을 더합니다.



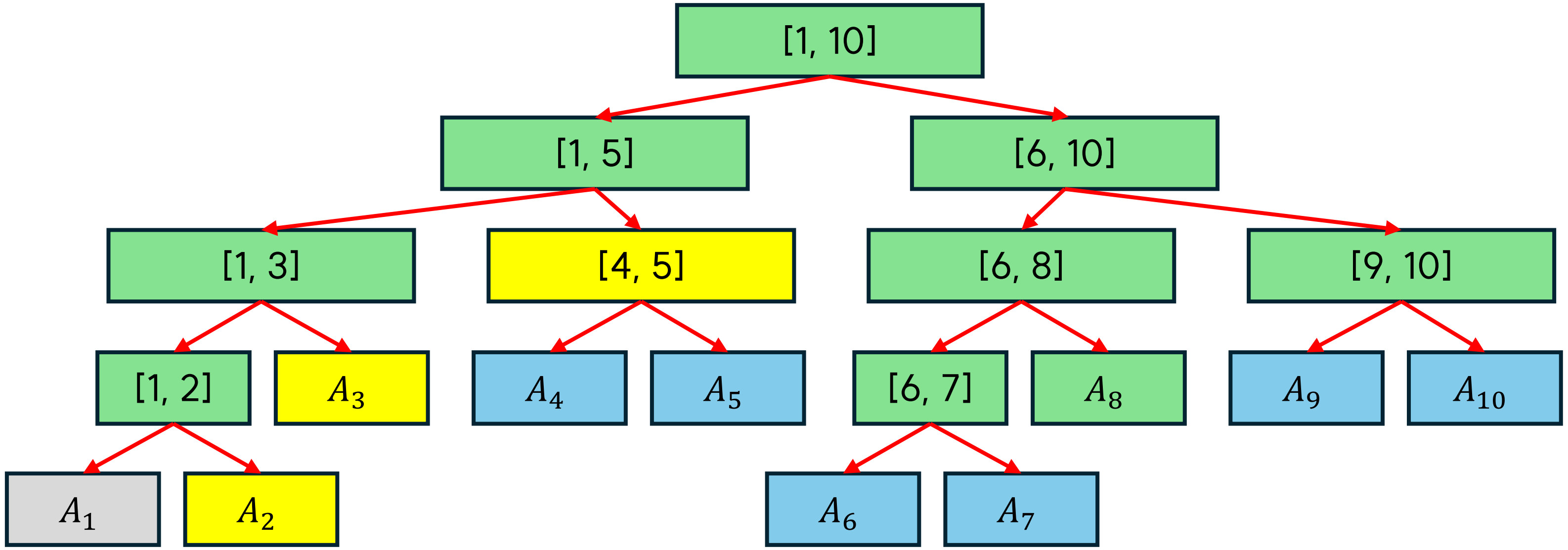
Data Structures

- 그 다음, $[l, r] = [6, 10]$과 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 일단, 겹치는 구간의 크기가 1 이상이지만, $[l, r]$이 $[L, R]$안에 속하지 않으므로 자식 노드로 한번 더 이동합니다.
  ($L \le l$ 이면서 $r \le R$)



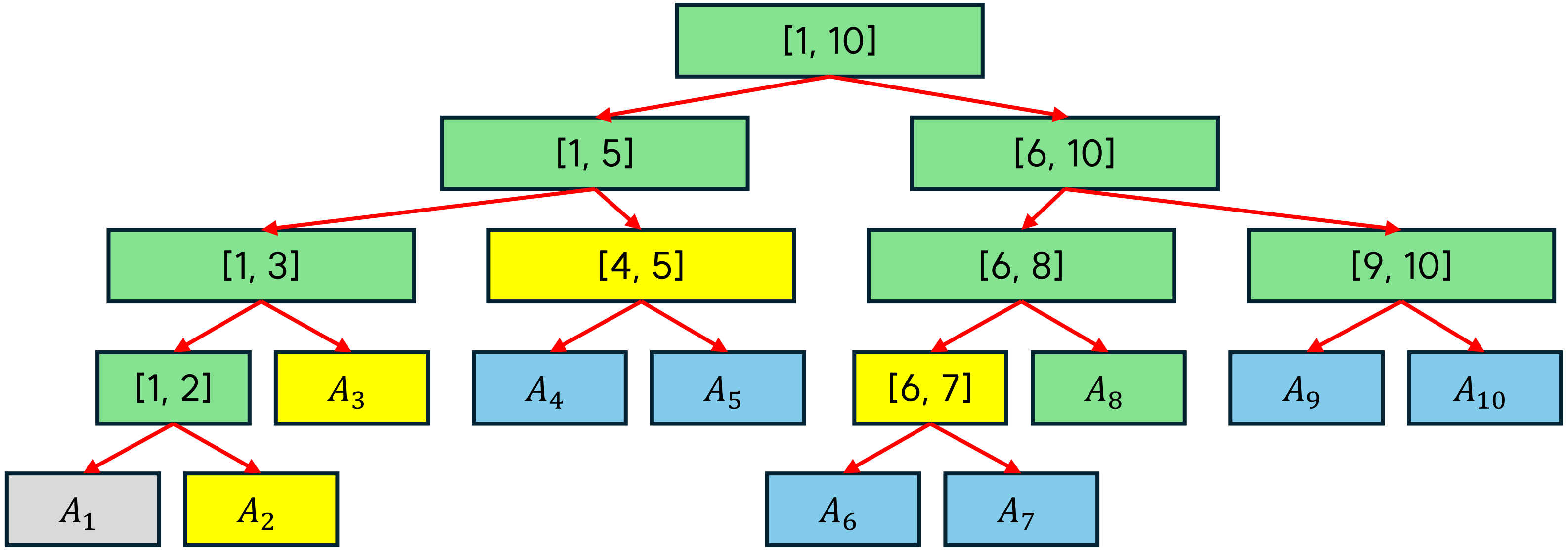
Data Structures

- 그 다음, $[l, r] = [6, 8]$과 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 일단, 겹치는 구간의 크기가 1 이상이지만, $[l, r]$이 $[L, R]$안에 속하지 않으므로 자식 노드로 한번 더 이동합니다.
  ($L \le l$ 이면서 $r \le R$)



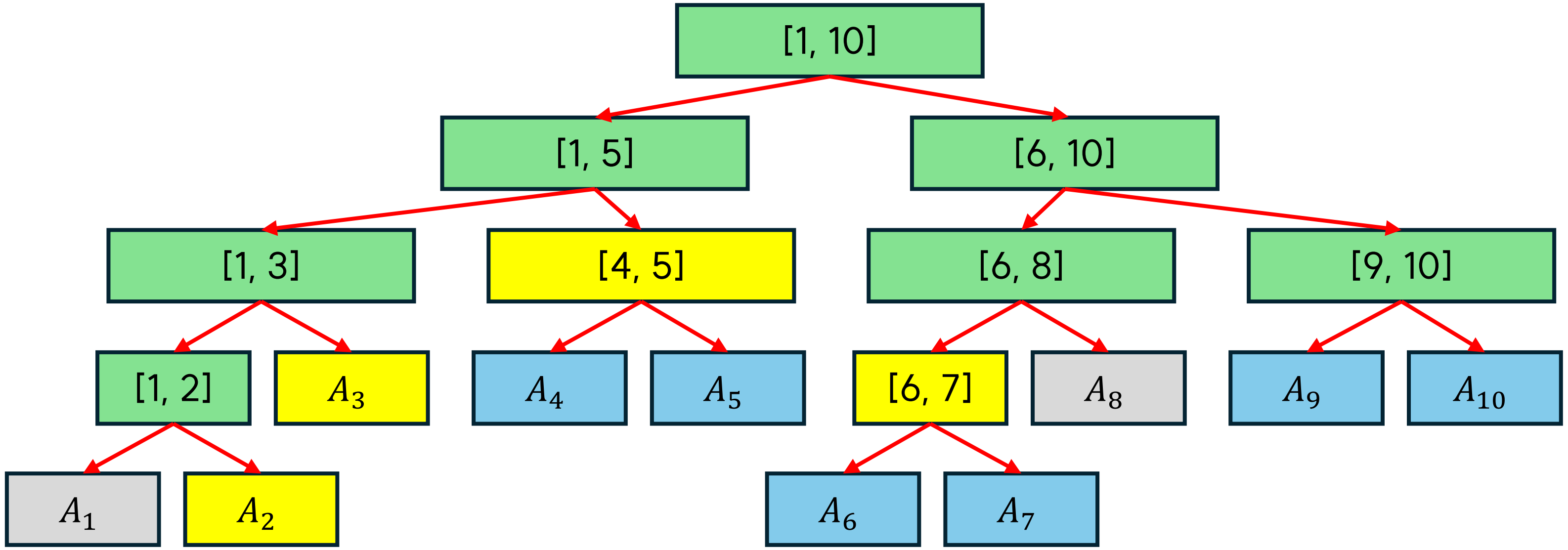
Data Structures

- 그 다음, $[l, r] = [6, 7]$과 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 겹치는 구간의 크기가 1 이상이고, $[l, r]$이 $[L, R]$에 속하므로 리턴값에 $[l, r]$노드의 값을 더합니다.



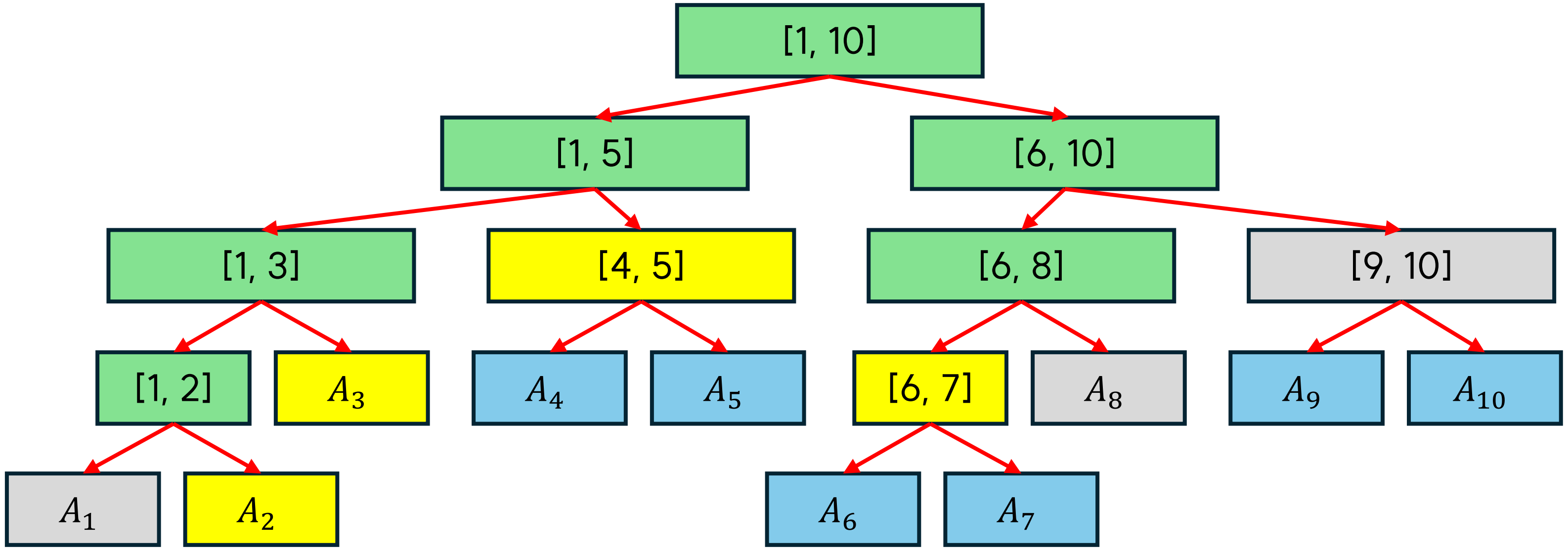
Data Structures

- 그 다음, $[l, r] = [8, 8] = A_8$과 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 겹치는 부분이 없습니다. 따라서 리턴값에 0을 더합니다.



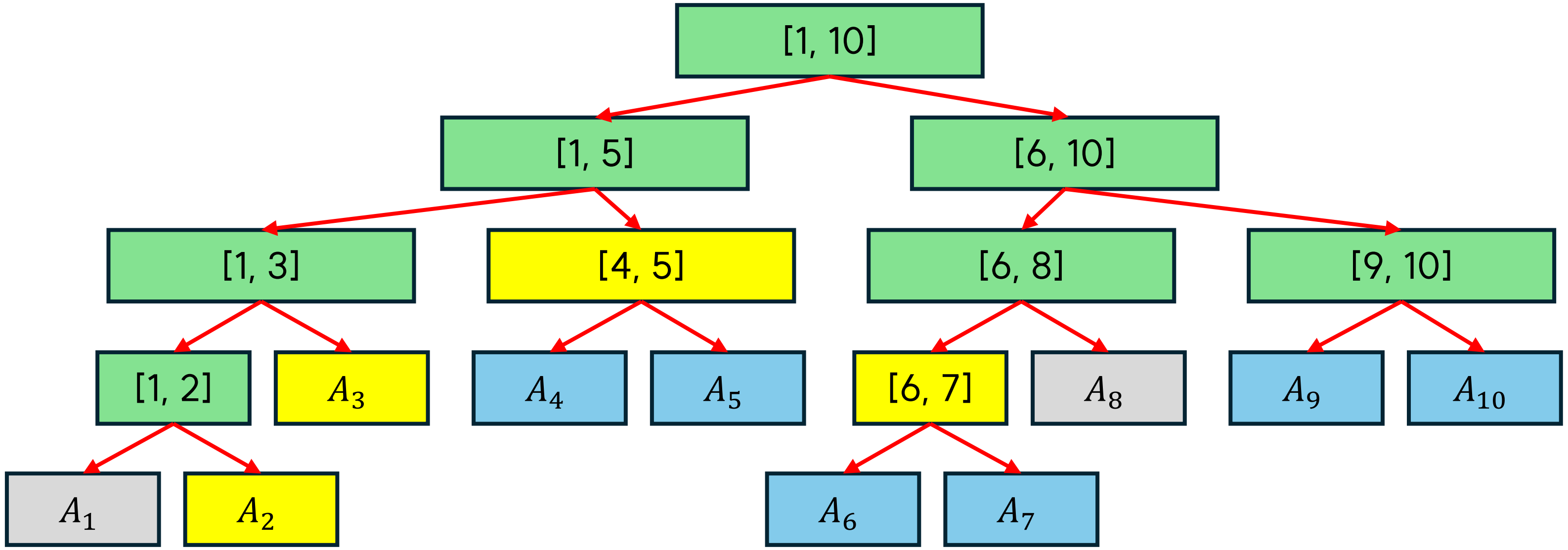
Data Structures

- 그 다음, $[l, r] = [9, 10]$과 $[L, R] = [2, 7]$이 겹치는 부분이 있는지 체크합니다. ($r < L$이거나 $R < l$이면 겹치지 않음)
- 겹치는 부분이 없습니다. 따라서 리턴값에 0을 더합니다.



Data Structures

- 이런 과정을 통해 쿼리 2도 쉽게 해결할 수 있게 됩니다.



Data Structures

- 세그먼트 트리를 구현할 때는 각 노드를 구조체로 만들지 않아도 됩니다.
- 단순히 부모 노드가 k번째 노드라고 하면, 왼쪽 자식 노드는 $2k$번째 노드, 오른쪽 자식 노드는 $2k+1$번째 노드가 됩니다.
- 따라서 최상위 노드가 1번째 노드라고 가정하고, 배열을 이용해서 만들어주면 됩니다.
- 쿼리마다 시간복잡도는 $O(\log N)$ 이므로 $O(Q \log N)$에 문제를 해결할 수 있습니다.
- 참고할 사항이지만, 트리 배열의 크기는 $4N$이상 정도면 된다고 합니다.

Data Structures

- 코드로 나타내면 다음과 같습니다. (범위 주의)

```
ll tree[500005];
ll arr[100005];
class SegmentTree {
public:
    void init(int node, int l, int r) {
        if(l==r) {
            tree[node] = arr[l];
        } else {
            int m = (l+r)/2;
            init(node*2, l, m);
            init(node*2+1, m+1, r);
            tree[node] = tree[node*2] + tree[node*2+1];
        }
    }

    void update(int node, int l, int r, int idx, int val) {
        if(idx < l || r < idx) {
            return;
        }
        if(l==r) {
            tree[node] = val;
        } else {
            int m = (l+r)/2;
            update(node*2, l, m, idx, val);
            update(node*2+1, m+1, r, idx, val);
            tree[node] = tree[node*2] + tree[node*2+1];
        }
    }

    ll query(int node, int l, int r, int L, int R) {
        if(L <= l && r <= R) {
            return tree[node];
        }
        if(l > R || r < L) {
            return 0;
        } else {
            int m = (l+r)/2;
            return query(node*2, l, m, L, R) + query(node*2+1, m+1, r, L, R);
        }
    }
};

int main() {
    fastio();
    int N, Q;
    cin >> N >> Q;
    SegmentTree myTree;
    while(Q--) {
        int q, x, y;
        cin >> q >> x >> y;
        if(q==1) {
            myTree.update(1, 1, N, x, y);
        } else {
            cout << myTree.query(1, 1, N, x, y-1) << '\n';
        }
    }
}
```

Data Structures - 2

끝.

다음 주에는 Graph Algorithms - 1로 찾아뵙겠습니다.