演算法與程式解題實務

Mong-Jen Kao (高孟駿)

Monday 18:30 – 21:20

Segment Tree

- Segment Tree is a <u>*data structure*</u> that can be used to ***answer queries*** that are ***related to "segments".***

- This data structure is <u>***applicable when***</u>
  - For any two "disjoint" segments I_1 and I_2,

    the answer for query$(I_1 \cup I_2)$ can be obtained from the answers for query(I_1) and query(I_2).

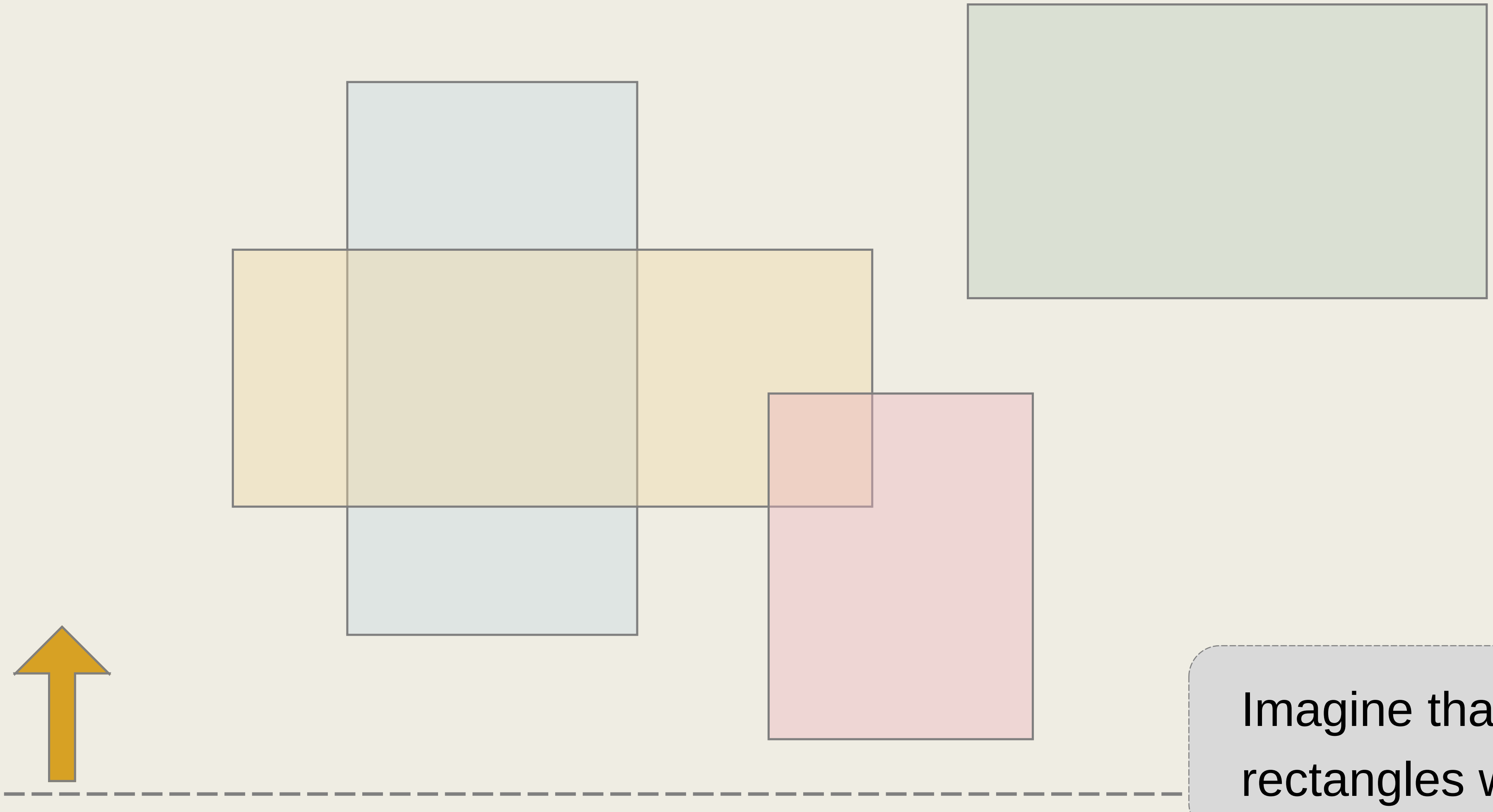
- In other words, segment tree can be used <u>*when the query can be solved by*</u> ***"divide-and-conquer"***.

Ex 1. Union of Segments

- Given $a_1 < a_2 < \cdots < a_n$ and an initial empty set $A \coloneqq \emptyset$,
  we want to process a sequence of queries of the following types.
  - **Insert**$(\boldsymbol{I})$ and **Delete**$(\boldsymbol{I})$ for some $I \coloneqq [a_i, a_j]$ with $i < j$.
    – to insert / delete the segment $I = [a_i, a_j]$ into A.
  - **Length**.
    – to report the length of $\bigcup_{I' \in A} I'$.

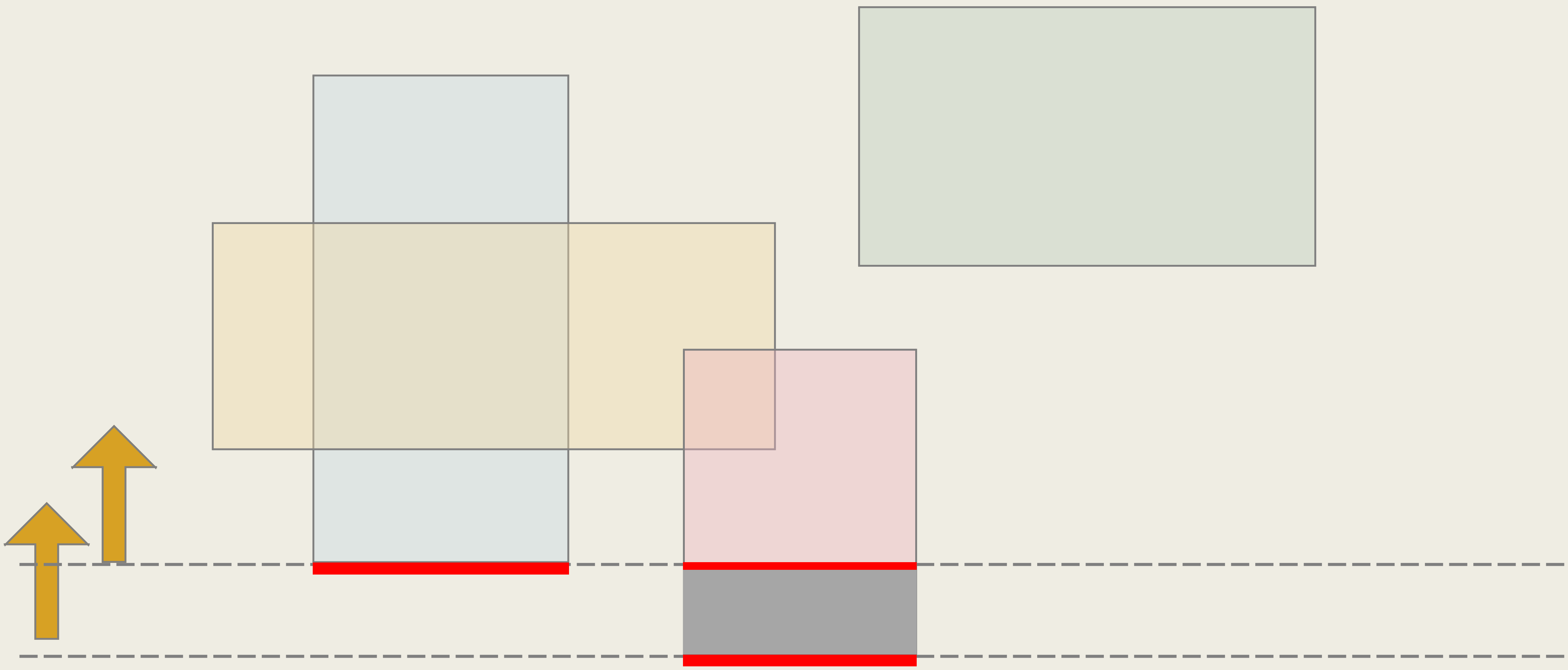
This is exactly the problem you have in ProgHW-III-D.

Application – Area of 2D-Rectangles

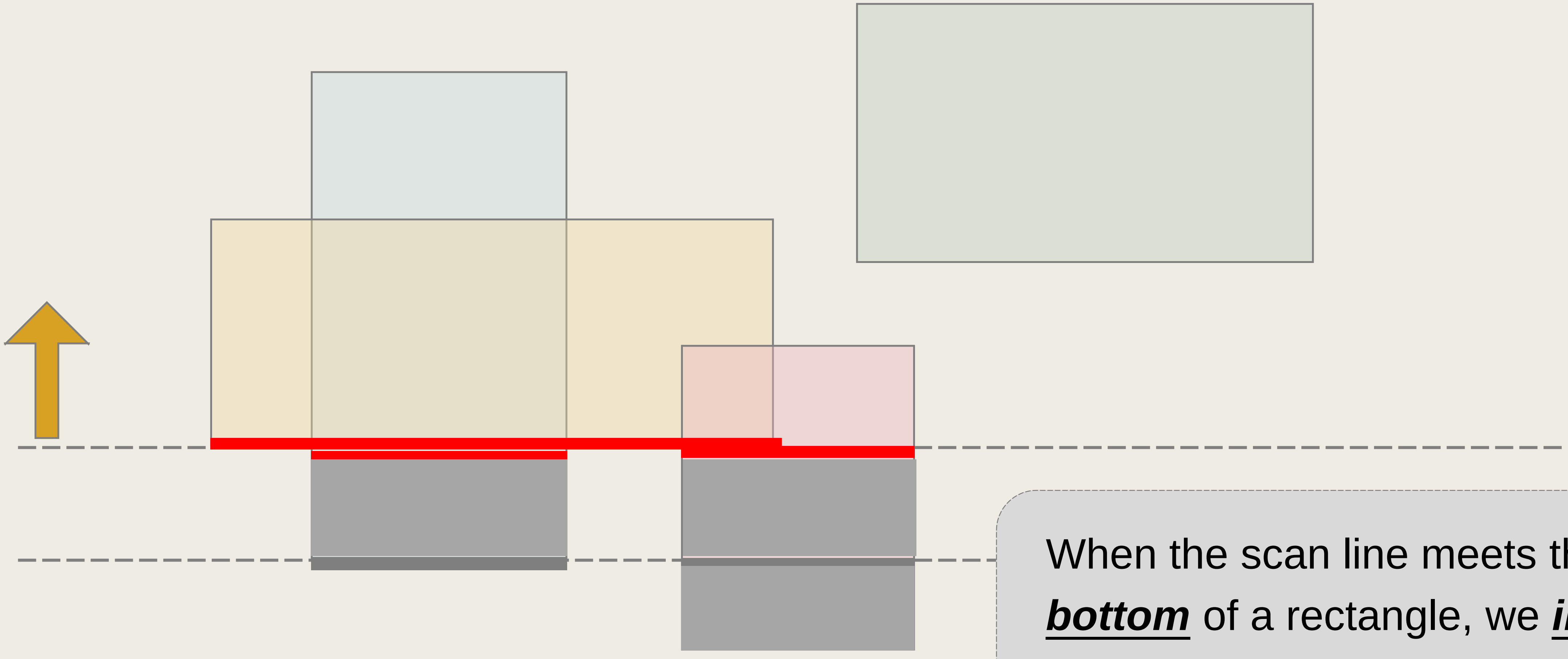


Imagine that we scan the rectangles with a sweep-line.

- Consider the intersection of the sweep-line with the rectangles.
  - As the sweep-line moves, the intersection *“integrates”* the area.

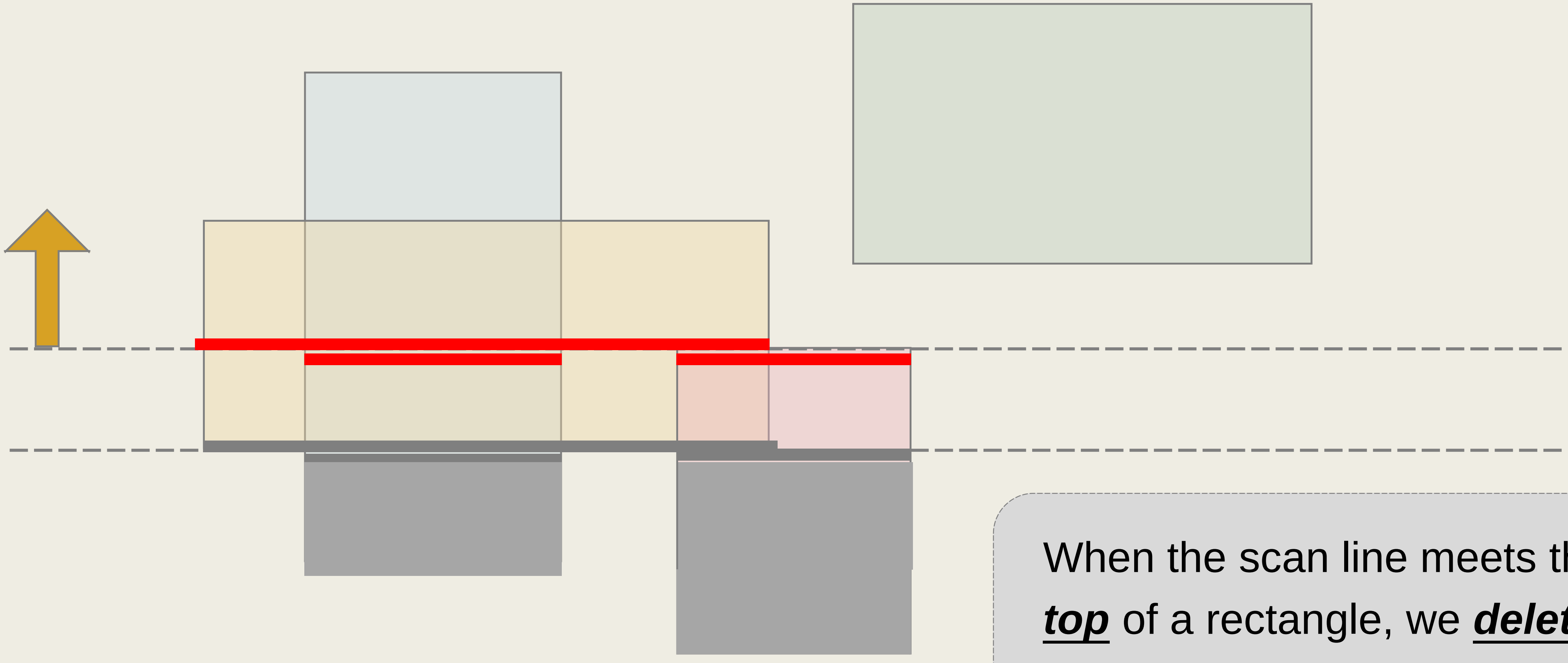


- Consider the intersection of the sweep-line with the rectangles.
  - As the sweep-line moves, the intersection <u>*"integrates"*</u> the area.



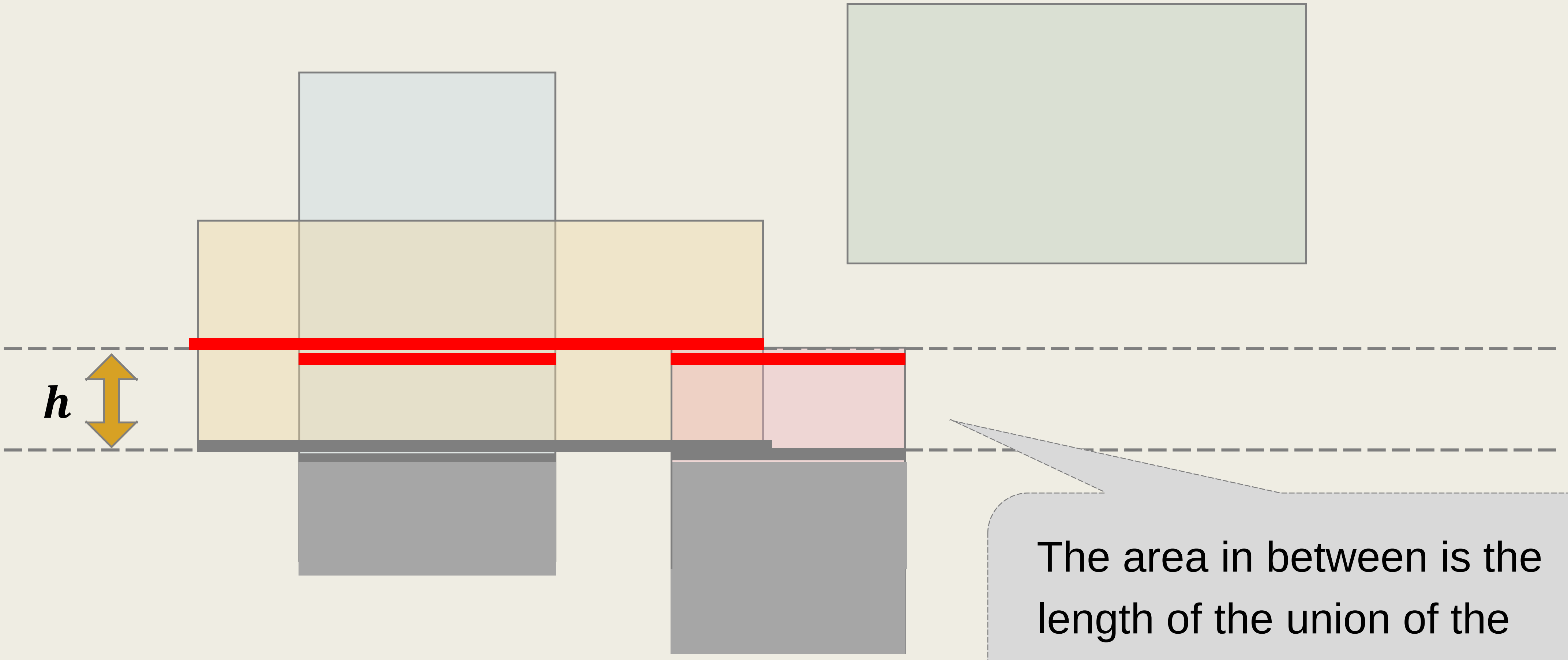
When the scan line meets the ***<u>bottom</u>*** of a rectangle, we ***<u>insert</u>*** the segment into A.

- Consider the intersection of the sweep-line with the rectangles.
  - As the sweep-line moves, the intersection *"integrates"* the area.



When the scan line meets the ***top*** of a rectangle, we ***delete*** the segment from A.

- Consider the intersection of the sweep-line with the rectangles.
  - As the sweep-line moves, the intersection *"integrates"* the area.



The area in between is the length of the union of the segments in A times the height.

Ex 2. Range Minimum Query

- Given $a_1, a_2, \dots, a_n$,

  we want to answer the following query.

  – **Minimum**(ℓ, r) for some $1 \le \ell \le r \le n$.

    – to report the minimum element between $a_\ell, \dots, a_r$.

  – **Update**(i, k) for some $1 \le i \le n$.

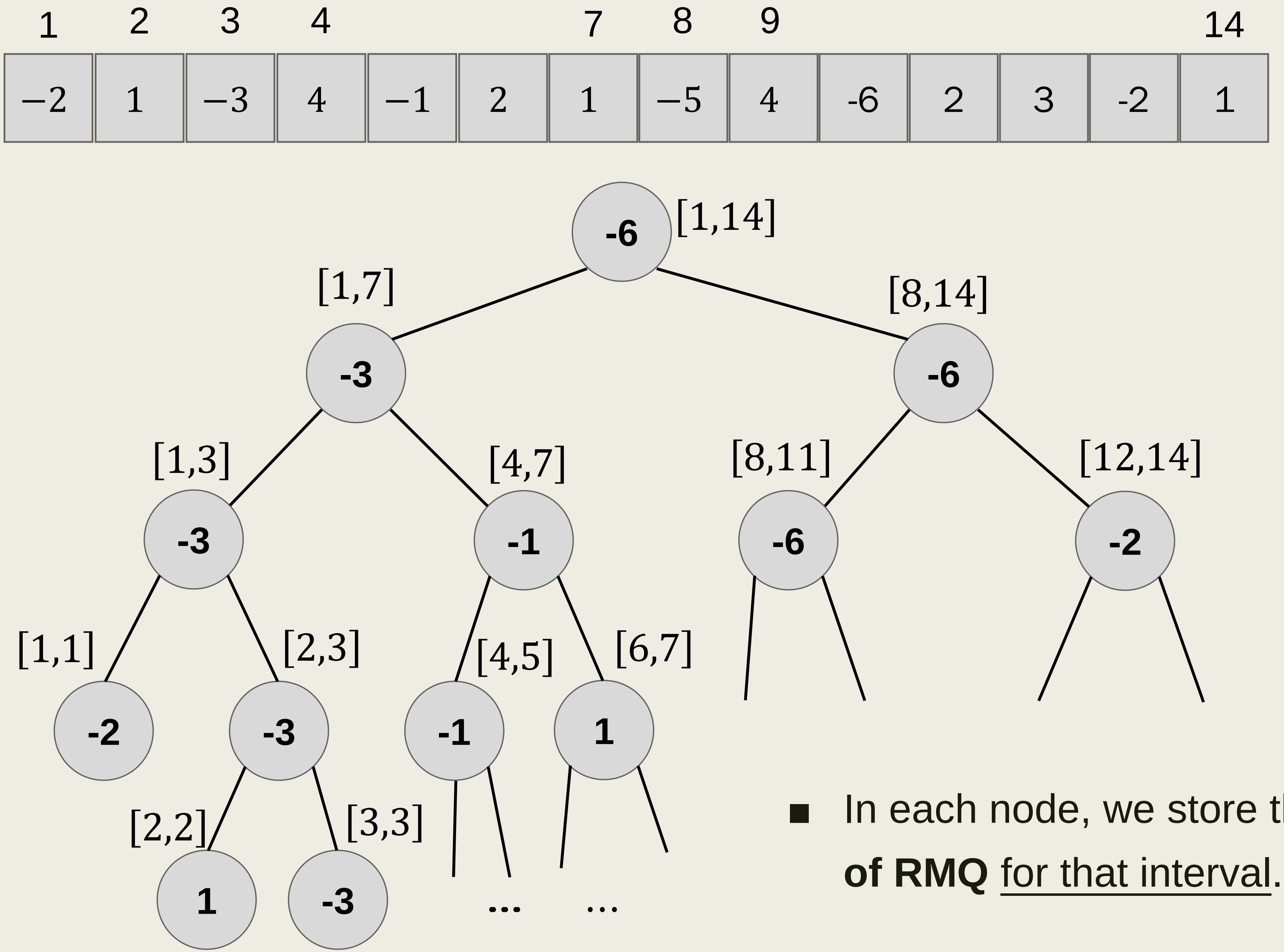
    – to change the value of a_i to k.

Has a minimum of -6.

-2	1	-3	4	-1	2	1	-5	4	-6	2	3	-2	1

Segment Tree for *<u>Range Minimum Query</u>*

- Let's examine how segment tree works for RMQ.
  - For any $1 \le \ell \le r \le n$, let $[\ell, r]$ denote the numbers $a_\ell, \ldots, a_r$.
- The segment tree is a complete binary tree with root $I_r \coloneqq [1, n]$, and each node $I_v \coloneqq [\ell, r]$ with $\ell < r$ has two children nodes
  - Left(v) for the segment $[\ell, \mathrm{mid}]$, where $\mathrm{mid} = \lfloor (\ell + r)/2 \rfloor$,
  - Right(v) for the segment $[\mathrm{mid} + 1, r]$.
  - In each node, we *store the* ***answer of RMQ*** *<u>for that interval</u>*.



- In each node, we store the **answer of RMQ** <u>for that interval</u>.

Segment Tree for *Range Minimum Query*

- We use the following structure to store the segment tree.

```
struct node {
        int left, right, mid;
        int rmq;
        node *lc, *rc;
} A[maxN*2];
```

where **`maxN`** is the maximum number of elements.

- ***Refer to the example code for the procedures***.

Building the Segment Tree for RMQ

- Building the tree is straightforward.

Simply follow the definition.

- Build-Tree(v, ℓ, r) -- to Build a segment tree for $[\ell, r]$ at node v.

A. Set $v.\mathrm{left} \leftarrow \ell$, $v.\mathrm{right} = r$, and $v.\mathrm{mid} \leftarrow (\ell + r)/2$.

B. if $\ell = r$, then // This is a leaf node
   set $v.\mathrm{rmq} = a_\ell$ and return.

C. Otherwise, create nodes y, z. Set $v.lc \leftarrow y$ and $v.rc \leftarrow z$.
   Call Build-Tree$(\, y,\ \ell,\ v.\mathrm{mid}\,)$ and Build-Tree$(\, z,\ v.\mathrm{mid} + 1,\ r\,)$.

D. Set $v.\mathrm{rmq} \leftarrow \min(\, v.lc.\mathrm{rmq},\ v.rc.\mathrm{rmq}\,)$.

Querying the Segment Tree for RMQ

- Let $I_v \coloneqq [\, v.\mathrm{left},\ v.\mathrm{right}\,]$ denote the segment stored in node v.

- Query-Tree(v, ℓ, r) -- to return the minimum within $[\ell, r] \cap I_v$.

A. // the node is completely contained within $[\ell, r]$.

If $\ell \leq v.\mathrm{left}$ and $r \geq v.\mathrm{right}$, then return $v.\mathrm{rmq}$.

B. If $v.\mathrm{mid} < \ell$, then return Query-Tree($v.rc,\ \ell,\ r$).

If $r \leq v.\mathrm{mid}$, then return Query-Tree($v.lc,\ \ell,\ r$).

C. Return

min(Query-Tree($v.lc, \ell, r$), Query-Tree($v.rc, \ell, r$)).

Make recursive calls according to the definition.

Analysis of the Procedure Query-Tree

- Let $I \coloneqq [\ell, r]$ denote the query interval and $I_v \coloneqq [\, v.\mathrm{left}, v.\mathrm{right}\,]$ be the segment stored in node v.

- The procedure starts from the root of the tree.
  - If the segment $I_v \subseteq I$, then $I \cap I_v = I_v$, and we already have the answer $v.\mathrm{rmq}$.

    $I \cap I_{v.lc} = \emptyset$ if $v.\mathrm{mid} < \ell$.

  - Otherwise, $$I \cap I_v = (I \cap I_{v.lc}) \cup (I \cap I_{v.rc}),$$

    and the answer is given by recursive calls to Query-Tree.

    $I \cap I_{v.rc} = \emptyset$ if $r \leq v.\mathrm{mid}$.

Analysis of the Procedure Query-Tree

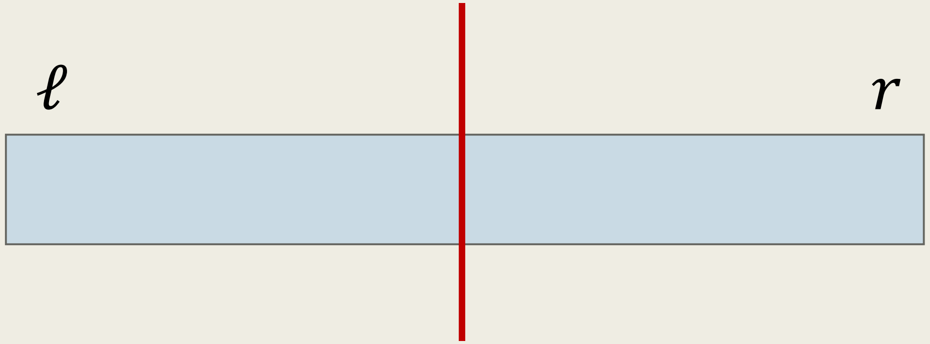
- For the time-complexity, consider the following cases.
  - If $I_v \subseteq I$, then the procedure *returns immediately*.
  - If $I \cap I_{v.lc} = \emptyset$ or $I \cap I_{v.rc} = \emptyset$,
    then the procedure makes *exactly one* recursive call.
  - Otherwise, *two recursive calls* are made.

Analysis of the Procedure Query-Tree

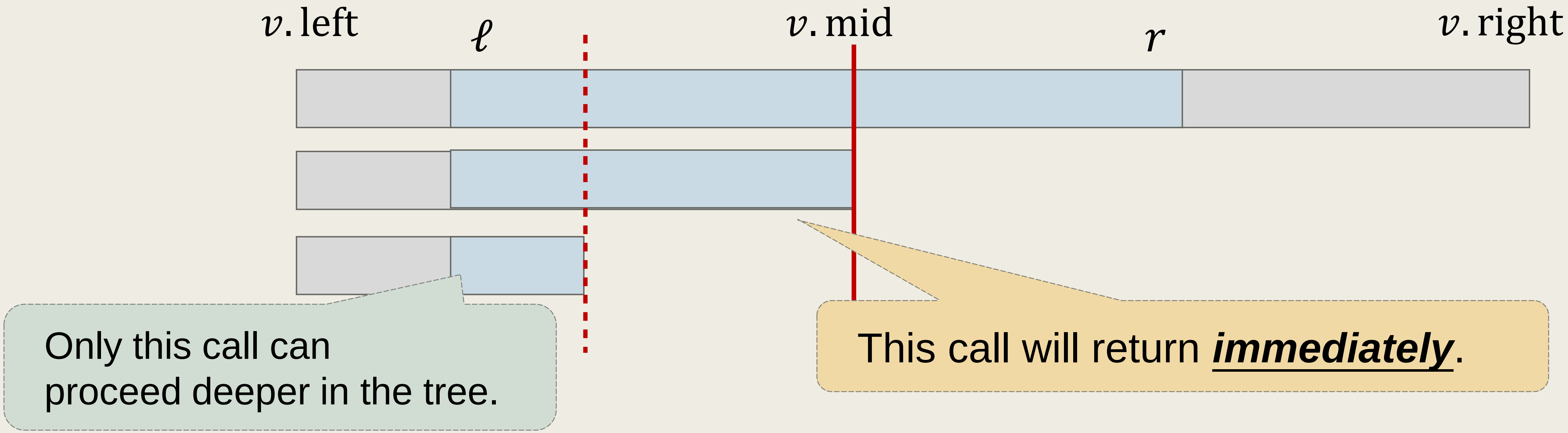
- The procedure starts from the root of the tree.
  - If <u>*at most one recursive call*</u> is made <u>*all the time*</u>, then the procedure runs in $O(\log n)$ time.
  - Otherwise, <u>*consider **the first time***</u> for which the procedure <u>***makes two recursive calls***</u>.
    - This happens when

      $$\ell \leq v.\mathrm{mid} < r$$

      holds <u>***for the first time***</u>.

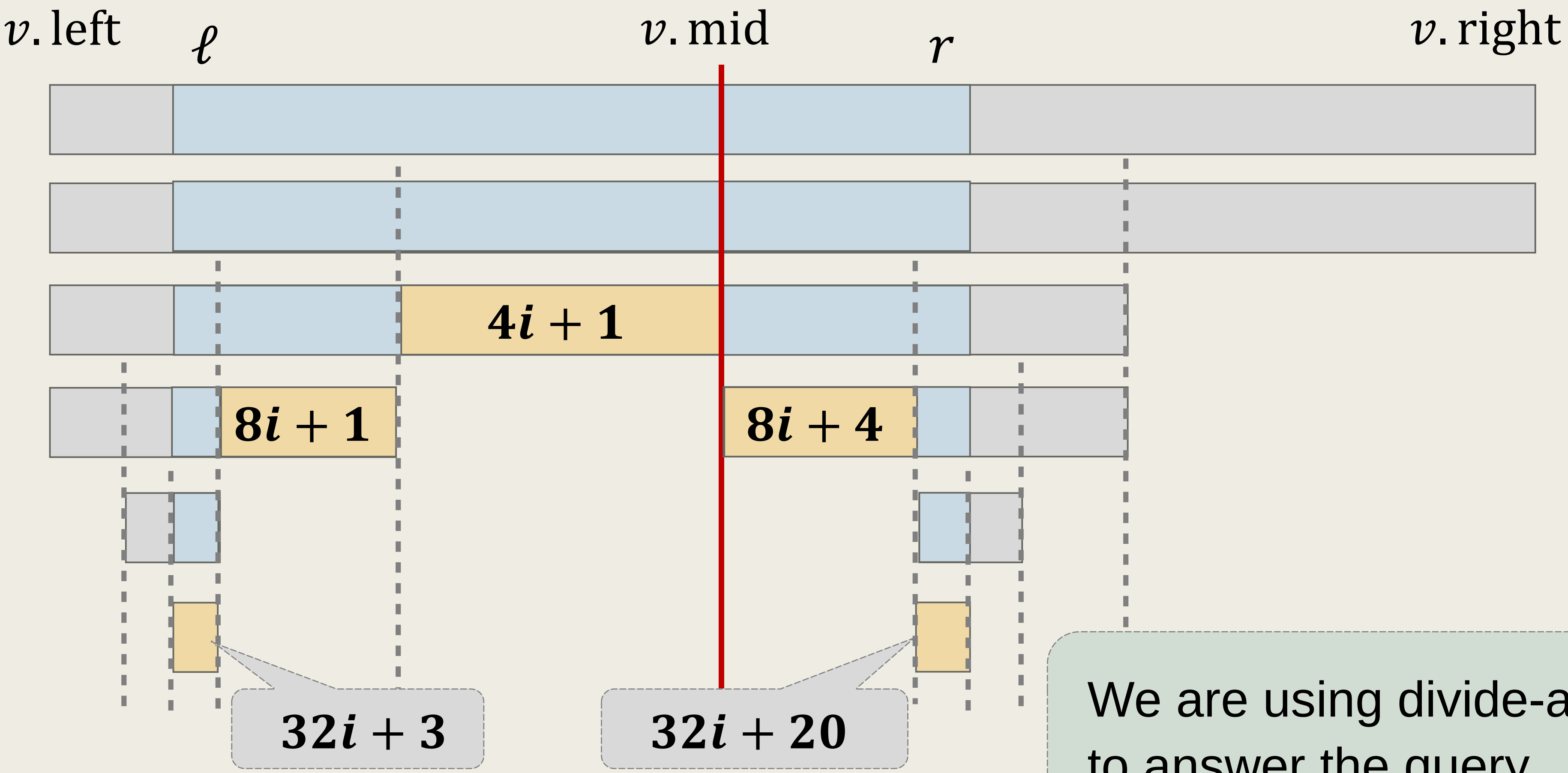


- Otherwise, consider **the first time** for which the procedure **makes two recursive calls**.
  - This happens when $\ell \leq v.\mathrm{mid} < r$ holds ***for the first time***.
  - After that, whenever the procedure makes two recursive calls, ***at most one of them*** can proceed deeper in the tree.



- Otherwise, consider **the first time** for which the procedure **makes two recursive calls**.
  - This happens when $\ell \leq v.\mathrm{mid} < r$ holds ***for the first time***.
  - After that, whenever the procedure makes two recursive calls, ***at most one of them*** can proceed deeper in the tree.
  - Hence, the query takes $O(\log n)$ time in this case.
- Equivalently, the query procedure ***divides*** *the query interval into* $O(log\, n)$ *pieces*, for ***which we already have the answer*** for.

- Equivalently, the query procedure ***divides*** *the query interval into* $O(\log n)$ *pieces*, for ***which we already have the answer*** for.



We are using divide-and-conquer to answer the query.

Updating the Segment Tree for RMQ

- Updating an element a_i is straightforward. It takes $O(\log n)$ time.

- Update-Tree(v, j) -- called after the value of a_j is updated.

A. If $v.\mathrm{left} = v.\mathrm{right}$ and $v.\mathrm{left} = j$, then
set $v.\mathrm{rmq} \leftarrow a_j$ and return.

B. If $v.\mathrm{mid} < j$, then call Update-Tree($v.rc,\ j$).
If $j \leq v.\mathrm{mid}$, then call Update-Tree($v.lc,\ j$).

C. Set $v.\mathrm{rmq} \leftarrow \min(v.lc.\mathrm{rmq},\ v.rc.\mathrm{rmq})$ and return.

Make recursive calls according to the definition.

Ex 2. Range Minimum Query

- Given $a_1, a_2, \dots, a_n$,

  we want to answer the following query.

  - **Minimum**(ℓ, r) for some $1 \le \ell \le r \le n$.

    – to report the minimum element between $a_\ell, \dots, a_r$.

  - **Update**(i, k) for some $1 \le i \le n$.

    – to change the value of a_i to k.

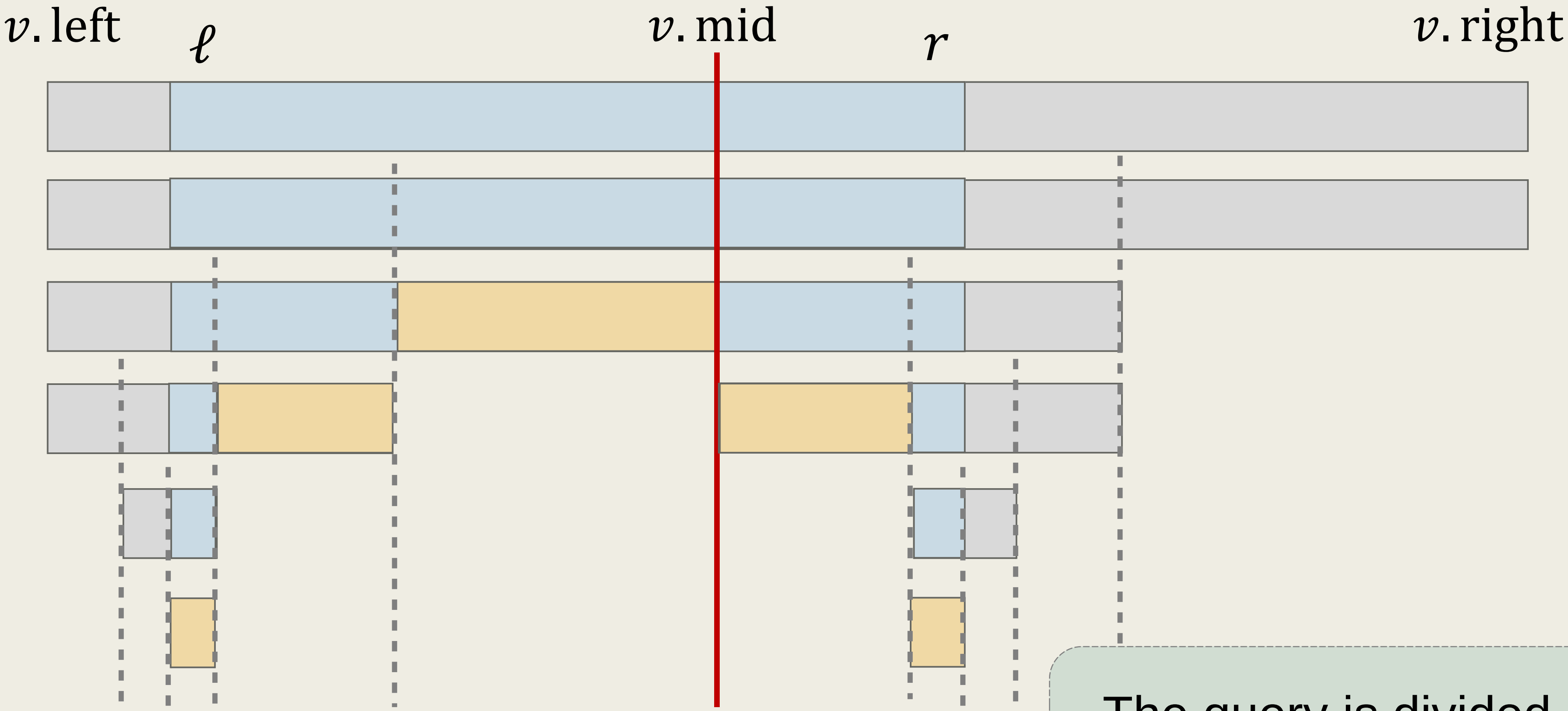
After that, each query can be done in $O(\log n)$ time.

Build the segment tree in $O(n)$ time.

Segment Tree for ***<u>Union of Segments</u>***

- For each query interval I to be inserted (or deleted),
  we ***<u>divide the interval</u>*** *into* $O(log\, n)$ *pieces* and
  store (or remove) them in (from) the segment tree.
  - We use the *<u>standard query procedure</u>* to store / remove the query interval.
  - For each node v,
    we need to store the following information.
    - <u>Number of times</u> I_v is <u>stored</u>.
    - **<u>Total length</u>** of the union of segments **<u>within</u>** I_v.

- The standard query procedure ***divides*** *the query interval into* $O(\log n)$ *pieces*, which can be stored in the tree.



The query is divided into $O(\log n)$ pieces and stored separately.

Segment Tree for ***<u>Union of Segments</u>***

- We use the following way to store the segment tree.

```
struct node {
    int left, right, mid;
    int cnt;  // number of times I_v is stored
    int len;
    node *lc, *rc;
} A[maxN*2];
```

where **`maxN`** is the maximum number of endpoints.

Area of 2-D Rectangles

- Given n rectangles $R_1, R_2, \ldots, R_n$,
  the are of their union can be computed in $O(n \log n)$ time.
  - Sorting takes $O(n \log n)$ time.
  - The segment tree can be built in $O(n)$ time.
  - There are $O(n)$ queries (insertion, deletion, length),
    each can be answered in $O(\log n)$ time.

Ex 3. Union of Segments (Adv. Version)

- Given $a_1 < a_2 < \cdots < a_n$ and an initial empty set $A \coloneqq \emptyset$, we want to process a sequence of queries of the following types.
  - **Insert**$(\boldsymbol{I})$ and **Delete**$(\boldsymbol{I})$ for some $I \coloneqq [a_i, a_j]$ with $i < j$.

    – to insert / delete the segment $I = [a_i, a_j]$ into A.

  - **Length** for some $I \coloneqq [a_i, a_j]$ with $i < j$.

    – to report the length of

$$I \cap \bigcup_{I' \in A} I' .$$

This is a bonus problem in ProgHW-III-D.