Introduction to **Algorithms**

Mong-Jen Kao (高孟駿)

Tuesday 10:10 – 12:00

Thursday 15:30 – 16:20

Program Assignment - III

# Segment Tree

■ Segment Tree is a _data structure_ that can be used to **answer queries** that are **related to "segments".**

■ This data structure is **applicable when**

  – 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 _when the query can be solved by_ **"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**(I) and **Delete**(I) for some $I \coloneqq \left[a_i, a_j\right]$ with $i < j$.

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

    - **Length**.

        – to report the length of $\bigcup_{I' \in A} I'$.
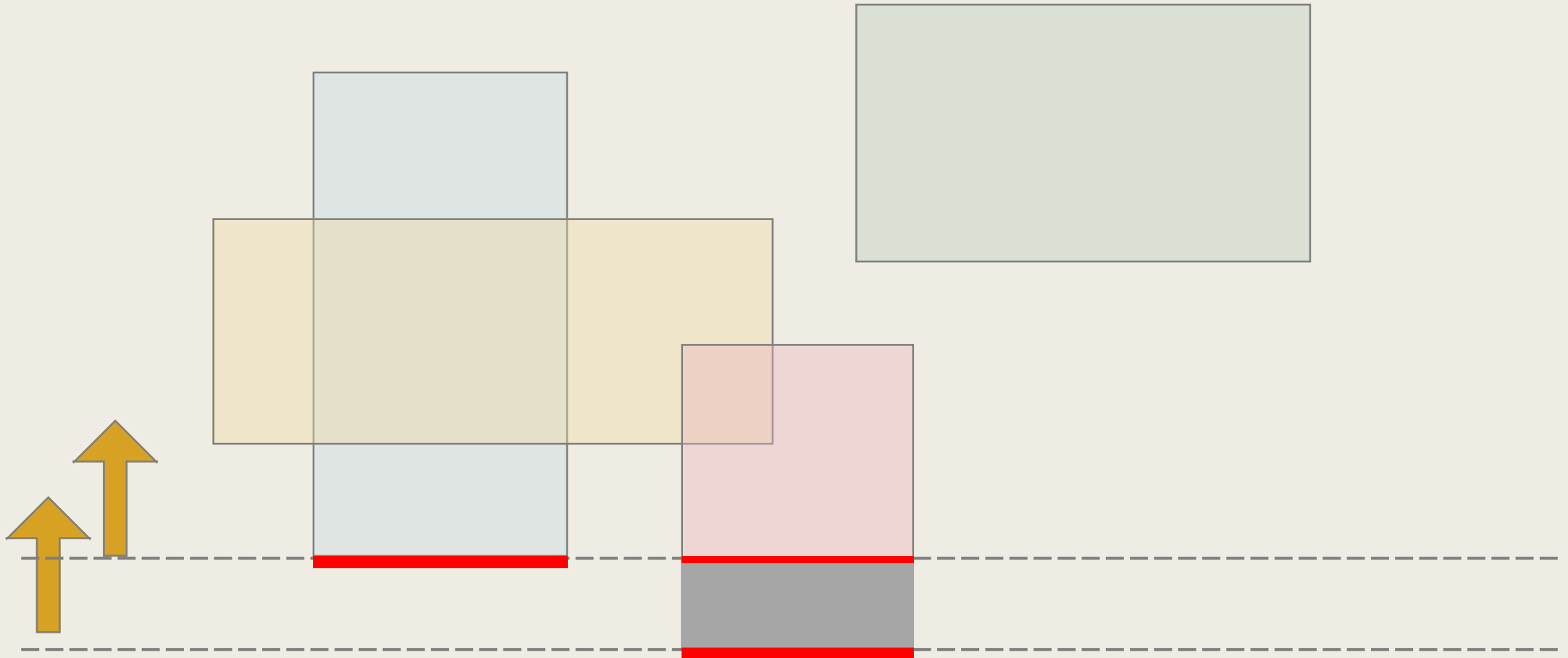
This is exactly the problem you have in ProgHW-III-D.
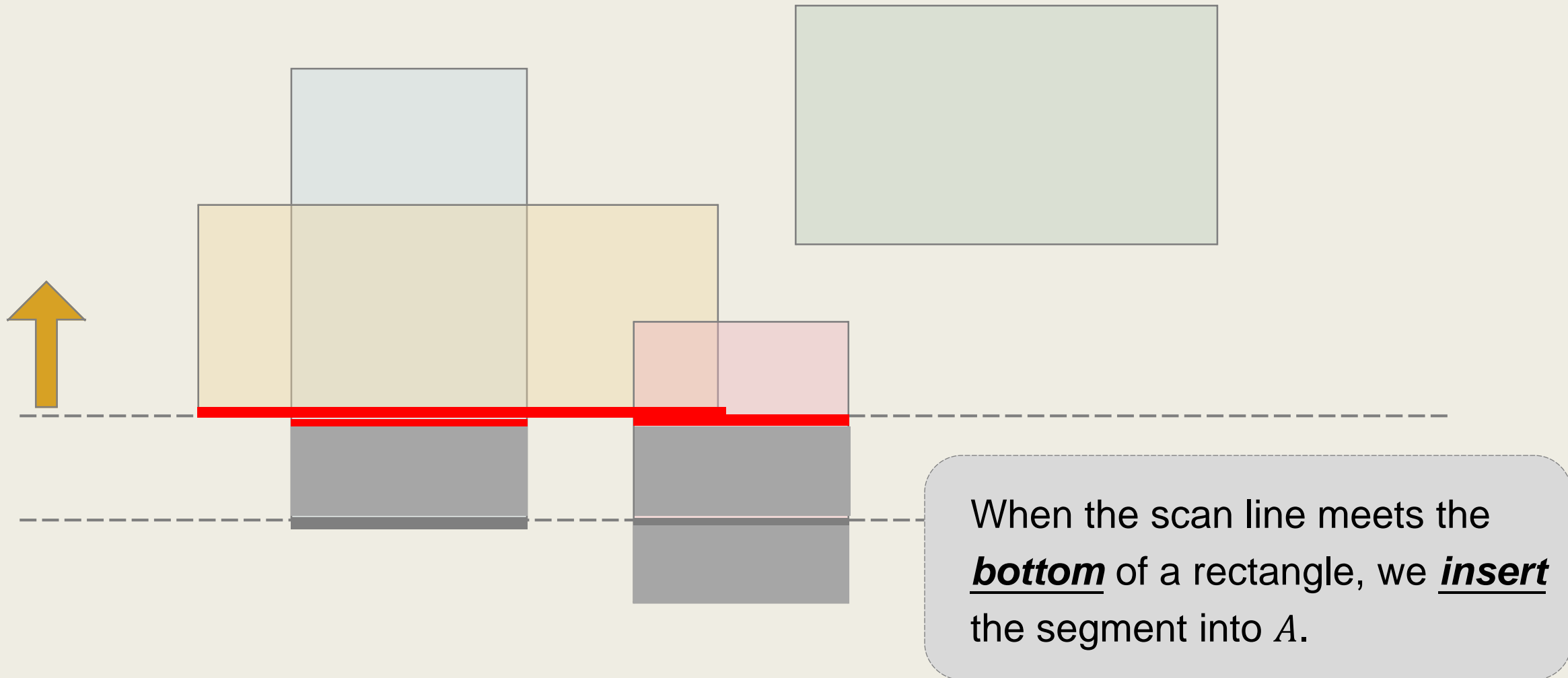
Application – Area of 2D-Rectangles
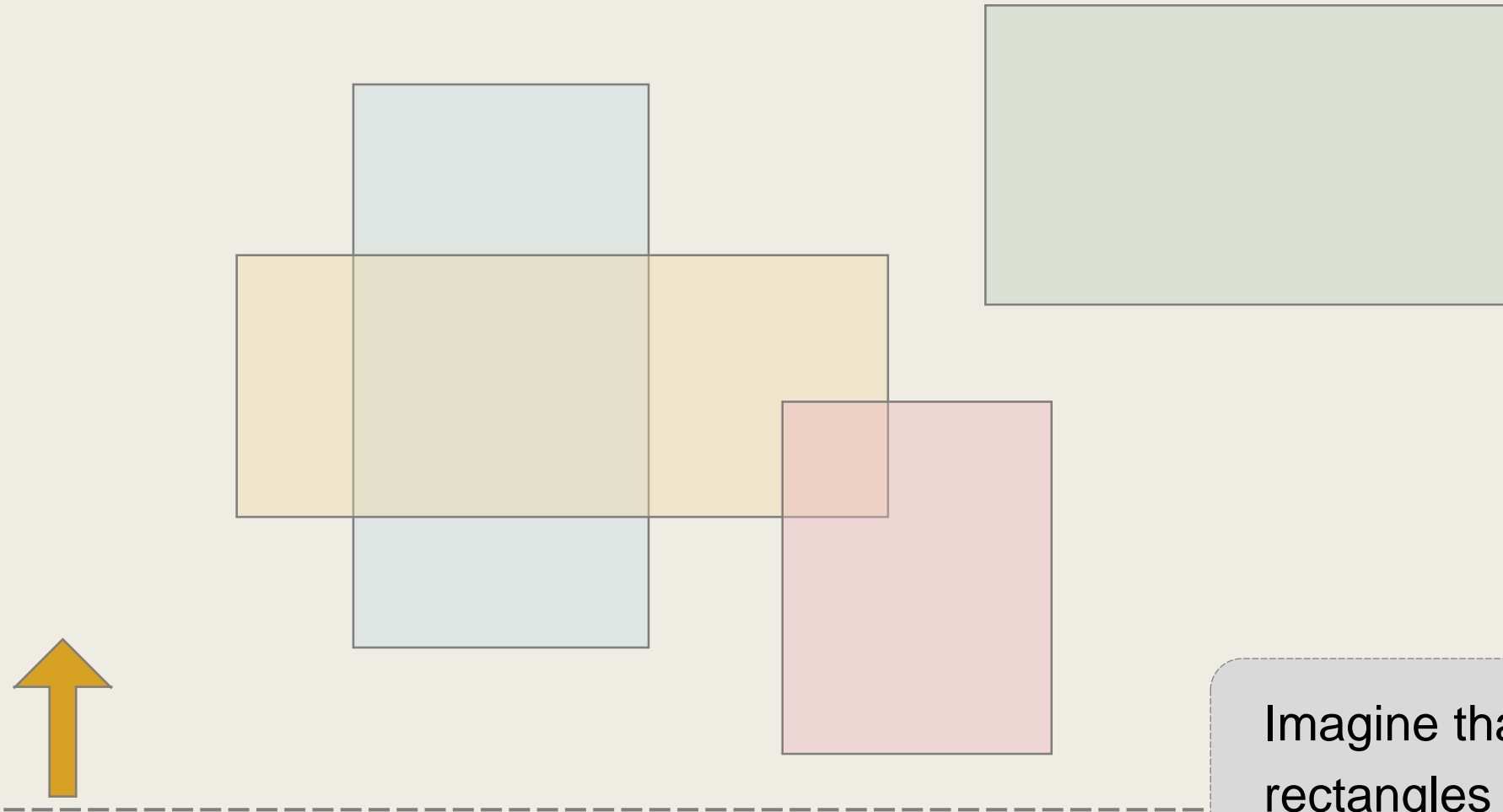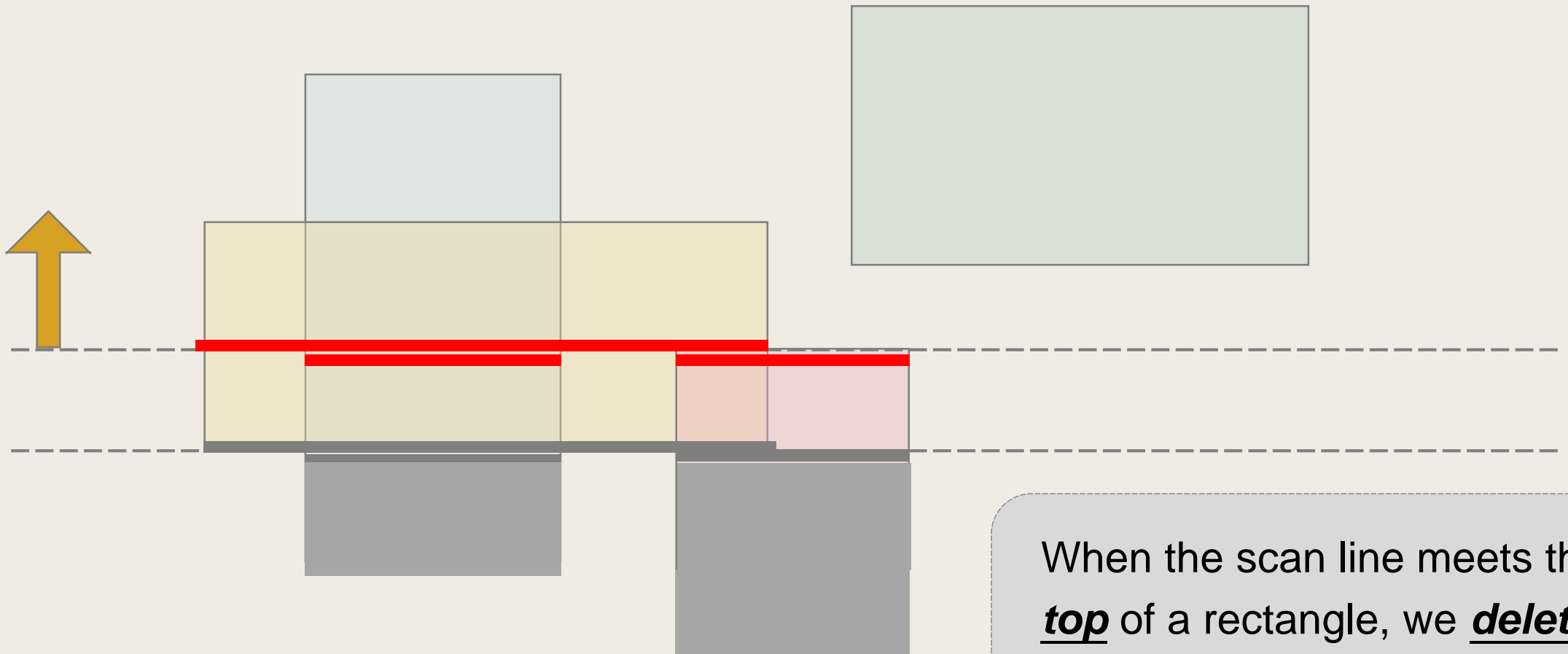
Imagine that we scan the rectangles with a sweep-line.

- 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 *"integrates"* the area.

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

■ Consider the intersection of the sweep-line with the rectangles.

  – As the sweep-line moves, the intersection *"integrates"* the area.

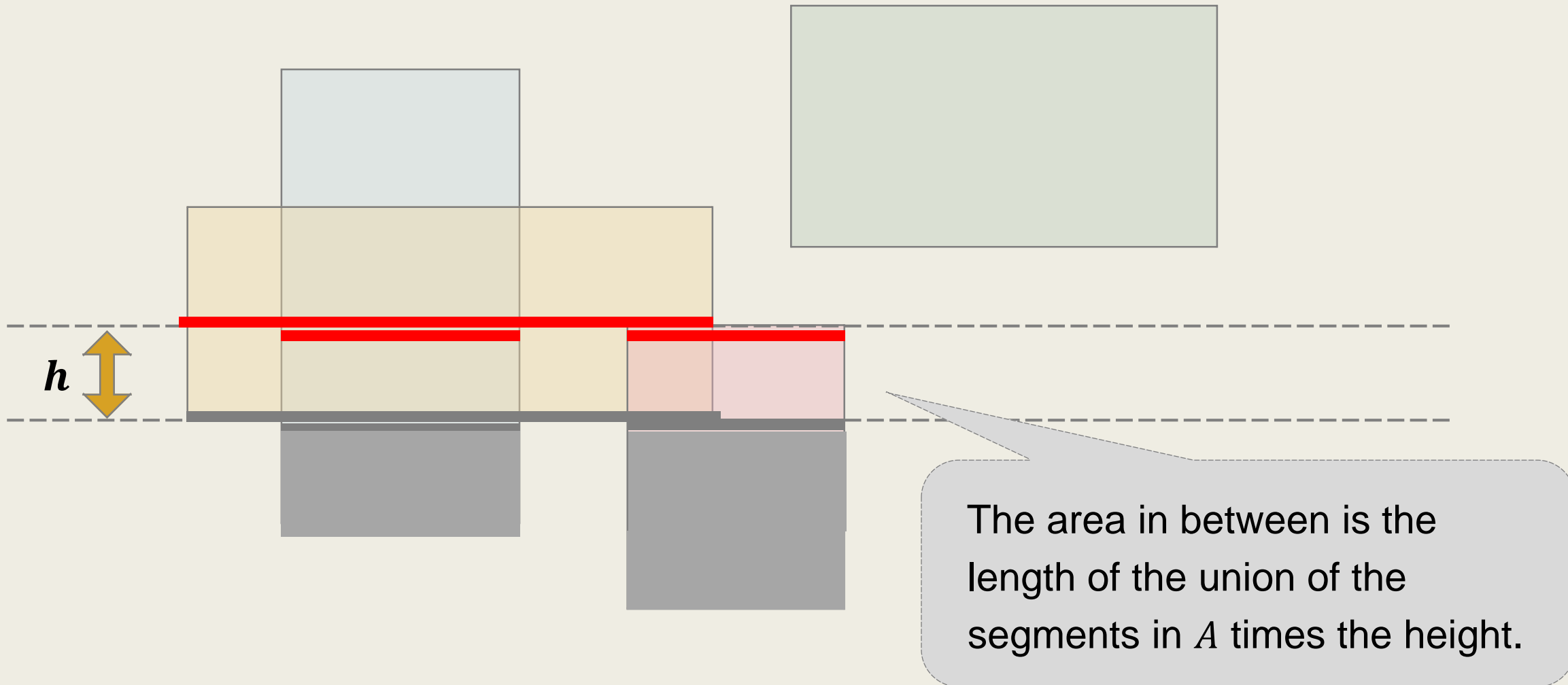When the scan line meets the ***top*** of a rectangle, we ***delete*** the segment from A.

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

h

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

Ex 2. Range Minimum Query

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

  we want to answer the following query.

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

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

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

   – 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 _Range Minimum Query_

■ Let's examine how segment tree works for RMQ.

  – For any $1 \leq \ell \leq r \leq n$, let $[\ell, r]$ denote the numbers $a_\ell, \ldots, a_r$.

■ The segment tree is a complete binary tree with root $I_r := [1, n]$, and each node $I_v := [\ell, r]$ with $\ell < r$ has two children nodes

  – Left(v) for the segment $[\ell, \text{mid}]$, where $\text{mid} = \lfloor (\ell + r)/2 \rfloor$,

  – Right(v) for the segment $[\text{mid} + 1, r]$.

  – In each node, we _store the **answer of RMQ** for that interval_.

1	2	3	4			7	8	9					14
−2	1	−3	4	−1	2	1	−5	4	-6	2	3	-2	1
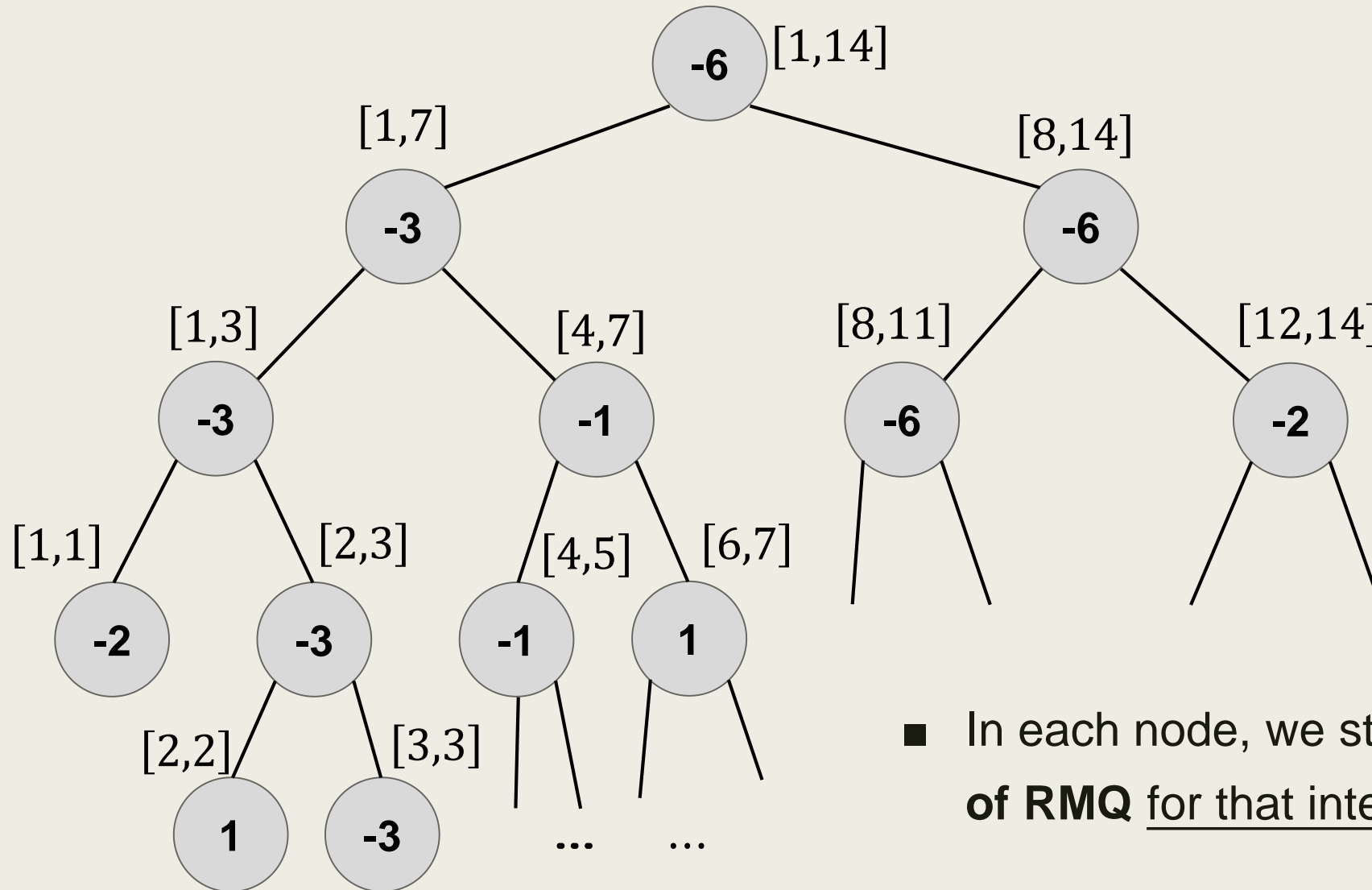
- In each node, we store the **answer of RMQ** <u>for that interval</u>.

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

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

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

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

Querying the Segment Tree for RMQ

- Let $I_v := [\, v.\text{left}, \ v.\text{right}\,]$ 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 $\ell \le v.\text{left}$ and $r \ge v.\text{right}$, then return $v.\text{rmq}$.

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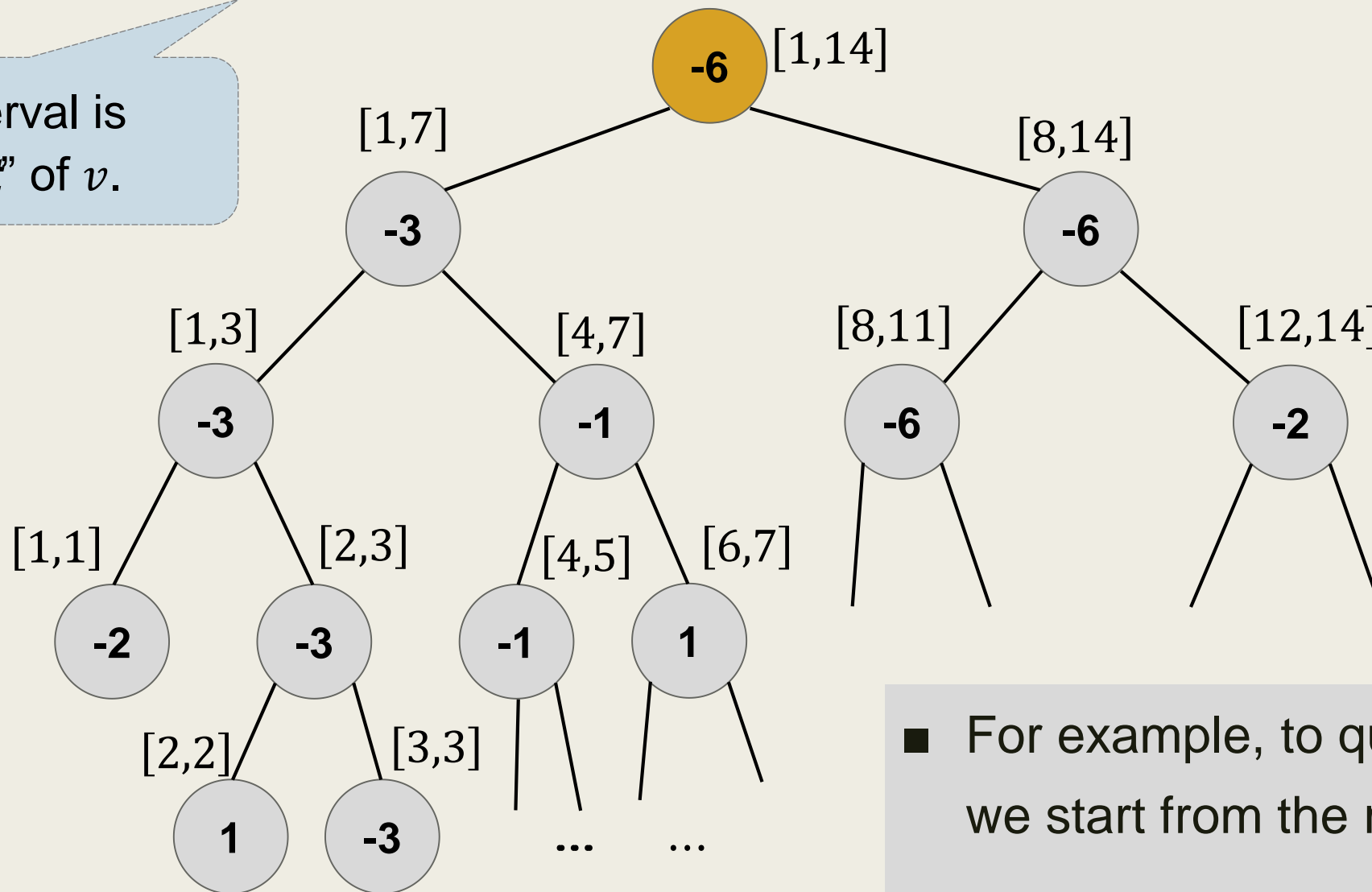
   If $r \le v.\text{mid}$, then return Query-Tree$(\, v.lc, \ \ell, \ r\,)$.

C. Return

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

> Make recursive calls according to the definition.

Query interval is in the "***right***" of v.
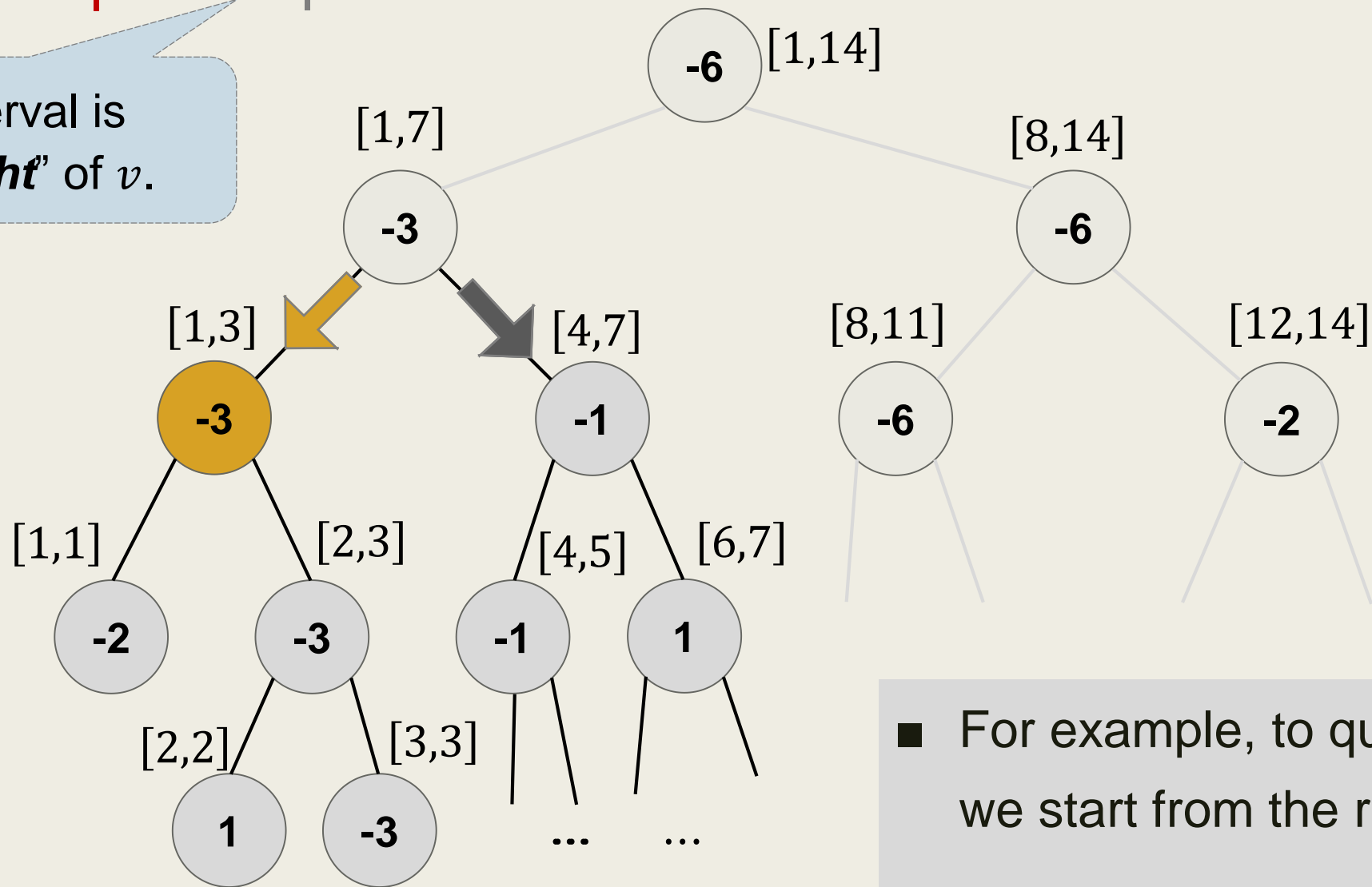
For example, to query [2,5], we start from the root node.

Analysis of the Procedure Query-Tree

- Let $I := [\ell, r]$ denote the query interval and
  $I_v := [\, v.\text{left}, v.\text{right}\,]$ 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 $v.\text{rmq}$.

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

  - 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.

    $$I \cap I_{v.rc} = \emptyset \;\; \text{if } r \le v.\text{mid}.$$

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
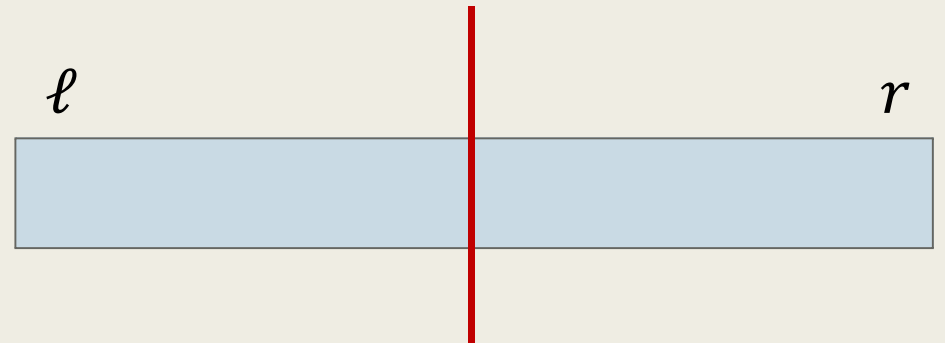
■ The procedure starts from the root of the tree.

- If _at most one recursive call_ is made _all the time_, then the procedure runs in $O(\log n)$ time.

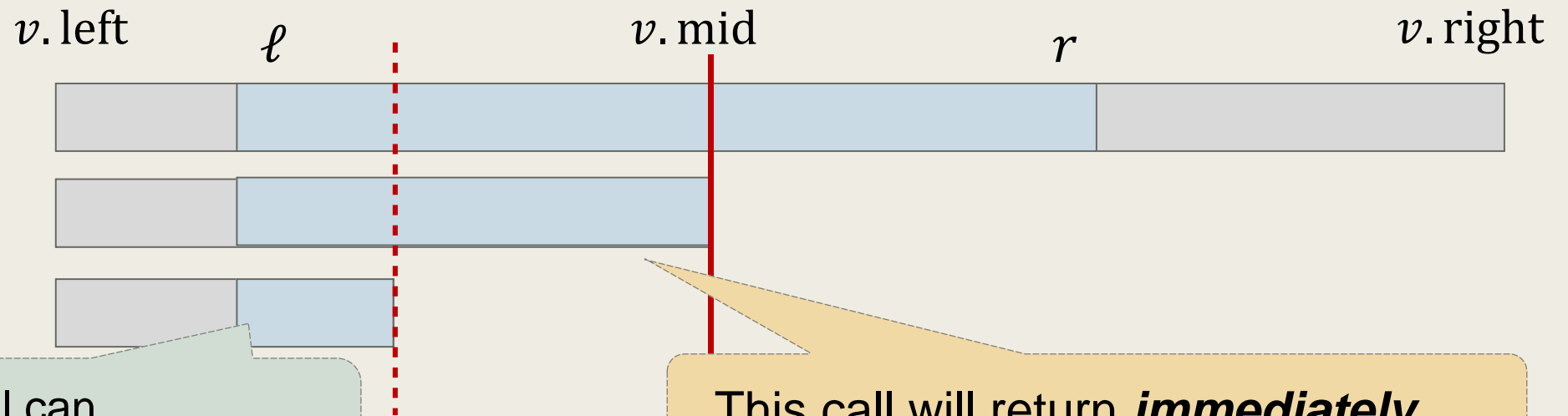- Otherwise, _consider **the first time**_ for which the procedure **_makes two recursive calls_**.

  ■ This happens when

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

  holds **_for the first time_**.

ℓ r

- Otherwise, <u>consider **the first time**</u> for which the procedure **makes two recursive calls**.

  - This happens when $\ell \leq v.\mathrm{mid} < r$ holds ***<u>for the first time</u>***.

  - After that, whenever the procedure makes two recursive calls, ***<u>at most one of them</u>*** can proceed deeper in the tree.

$v.\mathrm{left}$ ℓ $v.\mathrm{mid}$ r $v.\mathrm{right}$

Only this call can proceed deeper in the tree.

This call will return ***<u>immediately</u>***.

- Otherwise, consider <u>the first time</u> 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.
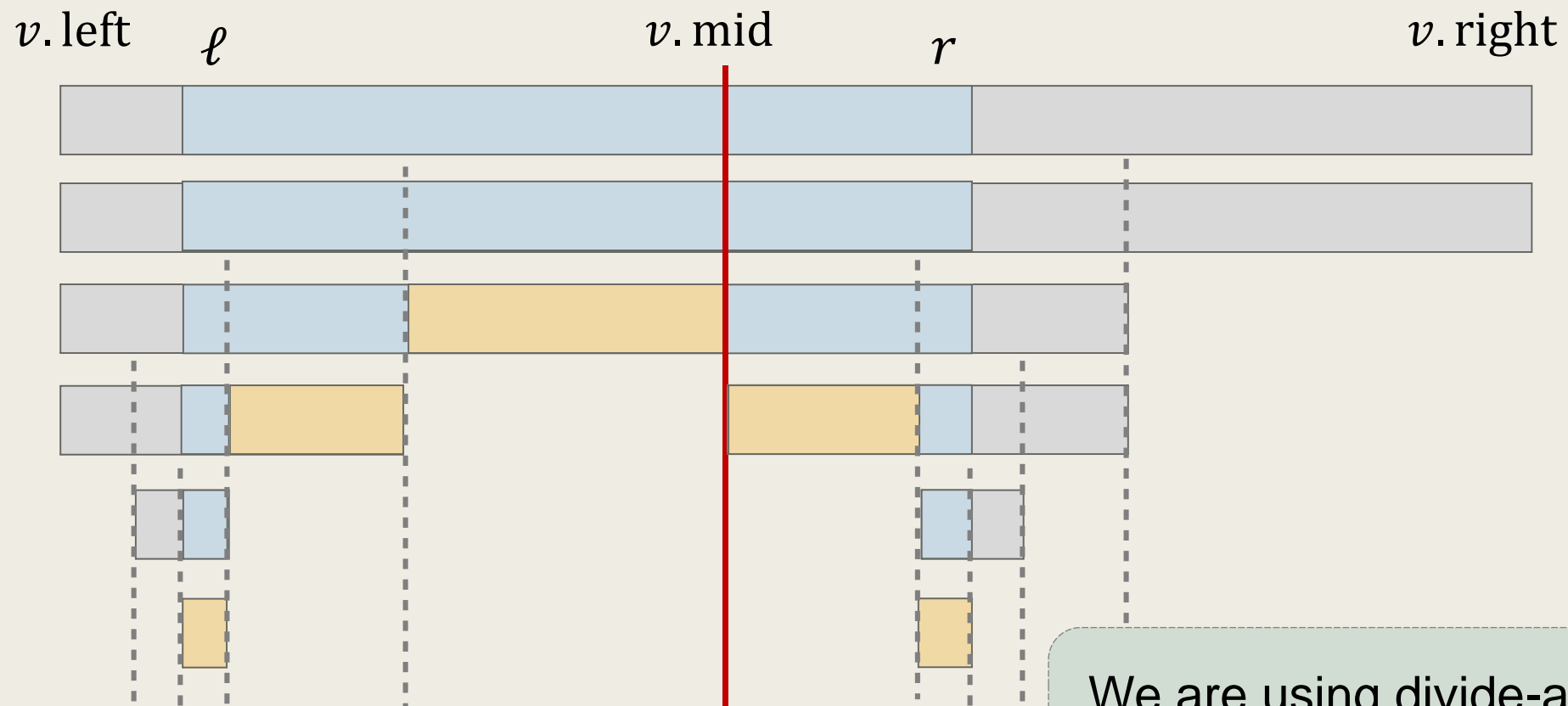
We are using divide-and-conquer to answer the query.

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 $1 \leq \ell \leq r \leq n$.

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

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

– 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 **_Union of Segments_**

- For each query interval I to be inserted (or deleted), we **_divide the interval_** into $O(\log n)$ *pieces* and store (or remove) them in (from) the segment tree.

  - We use the standard query procedure to store / remove the query interval.

  - For each node v, we need to store the following information.

    - Number of times I_v is stored.

    - **Total length** of the union of segments **within** I_v.

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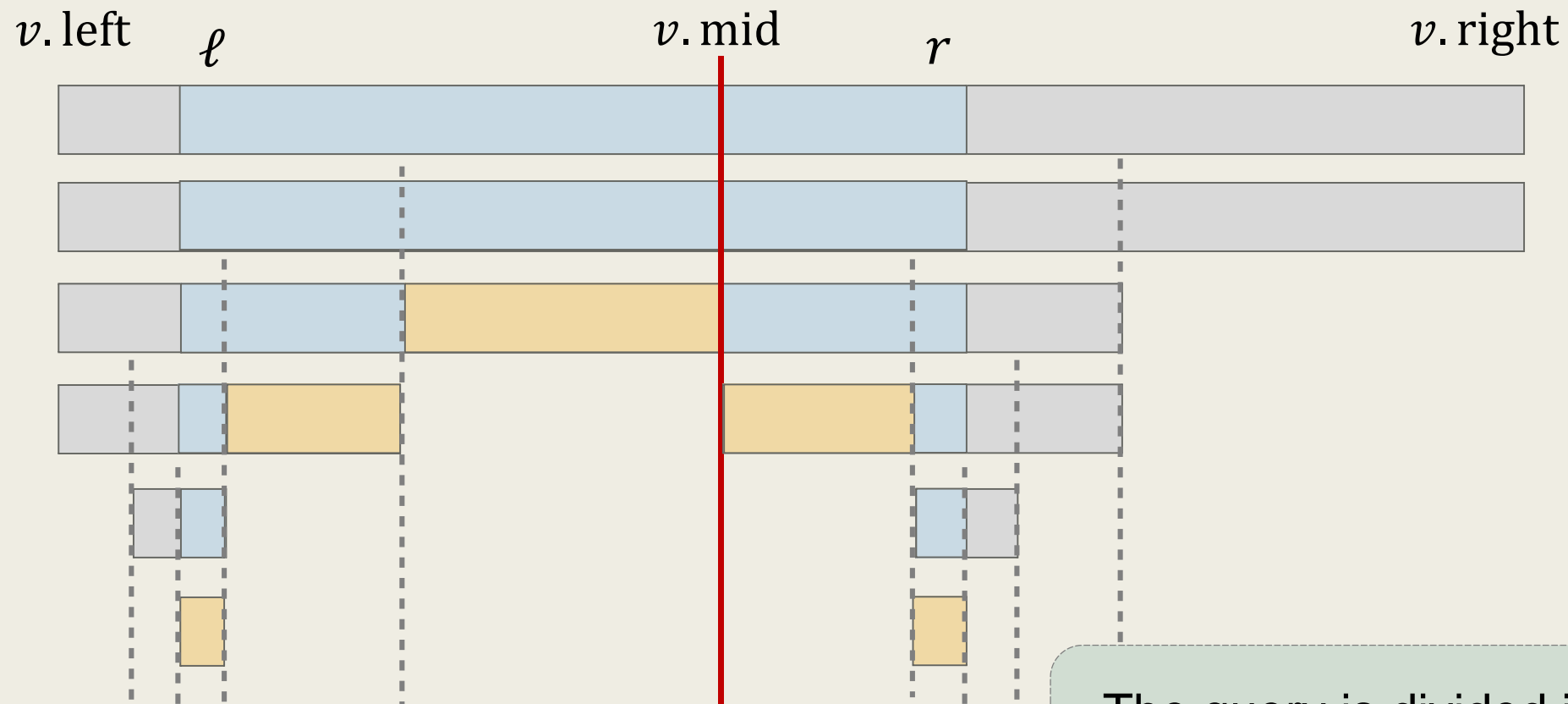
The query is divided into $O(\log n)$ pieces and stored separately.

Segment Tree for *Union of Segments*

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

```
struct node {

        int left, right, mid;
        int cnt;   // number of times l_v is stored
        int len;
        node *lc, *rc;

} A[maxN*2];
```

where `maxN` is the maximum number of endpoints.

Segment Tree for *Union of Segments*

- We have to think about the leaf nodes.

  - Recall that, we are using **divide-and-conquer** to solve the considered problem.

  - Hence, **the leaf nodes** correspond to **the smallest "atomic" segment** to be considered.

    - In the RMQ problem, the leaf nodes are associated with intervals $[i, i]$, which denotes **one single element a_i**.

Segment Tree for ***Union of Segments***

■ We have to think about the leaf nodes.

   – Hence, ***the leaf nodes*** correspond to ***the smallest "atomic" segment*** to be considered.

      ■ In the RMQ problem, the leaf nodes are associated with intervals $[i, i]$, which denotes one single element a_i.

      ■ In the Unions of Segments problem, the leaf nodes are with intervals $[i, i + 1]$, which denotes **the atomic interval** $[\boldsymbol{a_i, a_{i+1}}]$.

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 $A := \emptyset$,

  we want to process a sequence of queries of the following types.

  - **Insert**(I) and **Delete**(I) for some $I := [a_i, a_j]$ with $i < j$.

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

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

    – to report the length of

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

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