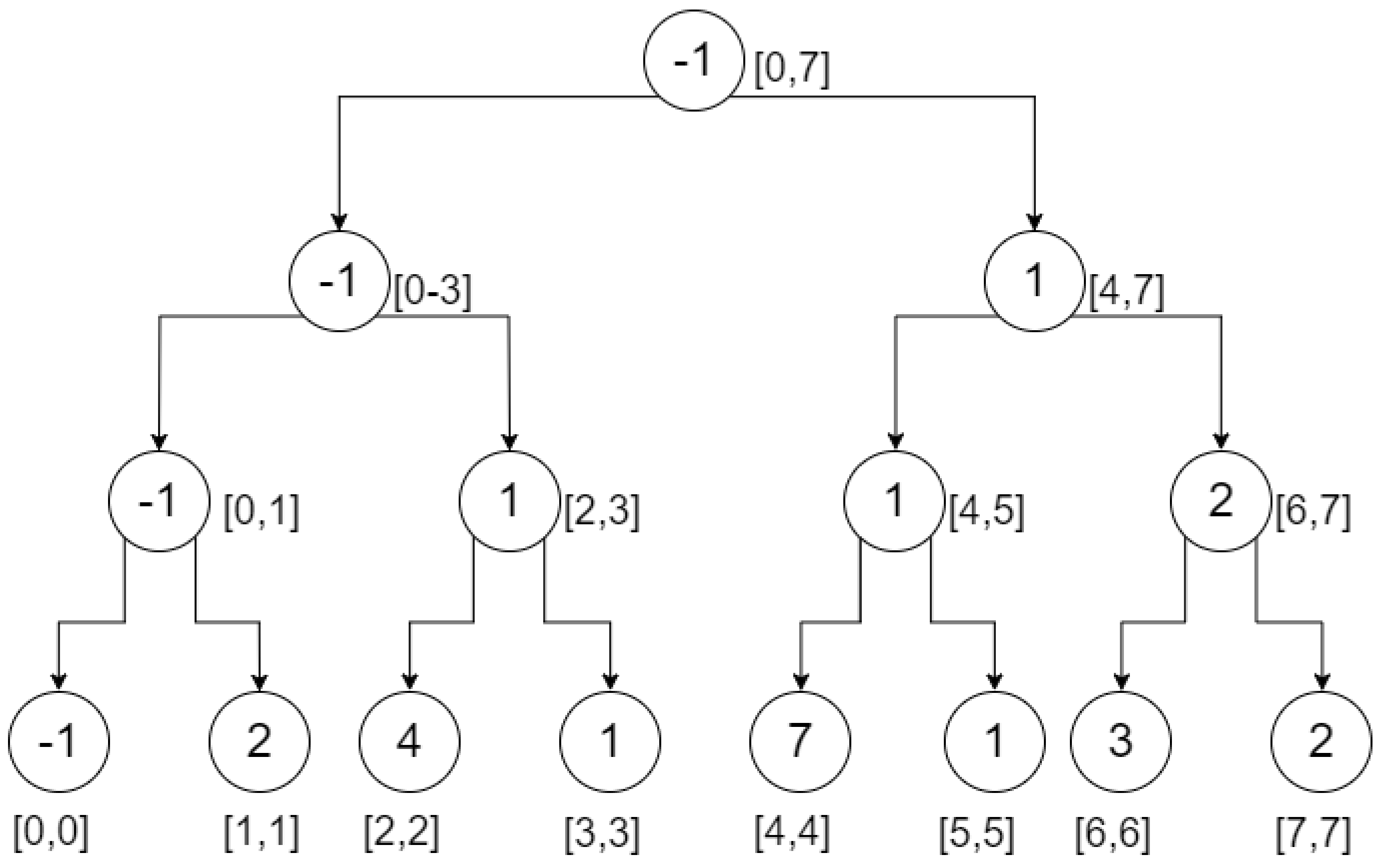
Lazy Propagation

Example

Let's say, you have already created a segment tree. You are required to update the values of the array, this will not only change the leaf nodes of your segment tree, but also the minimum/maximum in some nodes. Let's look at this with an example:



This is our minimum segment tree of **8** elements. To give you a quick reminder, each nodes represent the minimum value of the range mentioned beside them. Let's say, we want to increment the value of the first item of our array by **3**. How can we do that? We'll follow the way in which we conducted RMQ. The process would look like:

> At first, we traverse the root. **[0,0]** partially overlaps with **[0,7]**, we go to both directions.
>> At left subtree, **[0,0]** partially overlaps with **[0,3]**, we go to both directions.
>>> At left subtree, **[0,0]** partially overlaps with **[0,1]**, we go to both directions.
>>>> At left subtree, **[0,0]** totally overlaps with **[0,0]**, and since its the leaf node, we update the node by increasing it s value by **3**. And return the value **-1** + **3** = **2**.
>>>> At right subtree, **[1,1]** doesn't overlap with **[0,0]**, we return the value at the node(**2**).
>>>> The minimum of these two returned values(**2**, **2**) are **2**, so we update the value of the current node and return it.
>>>
>>> At right subtree **[2,3]** doesn't overlap with **[0,0]**, we return the value of the node. (**1**).
>>> Since the minimum of these two returned values (**2**, **1**) is **1**, we update the value of the current node and return it.
>>
>> At right subtree **[4,7]** doesn't overlap with **[0,0]**, we return the value of the node. (**1**).
>
> Finally the value of the root node is updated since the minimum of the two returned values **(1,1)** is **1**.

We can see that, updating a single node requires `O(logn)` time complexity, where **n** is the number of leaf nodes. So if we are asked to update some nodes from **i** to **j**, we'll require `O(nlogn)` time complexity. This becomes cumbersome for a large value of **n**. Let's be *Lazy* i. e., do work only when needed. How? When we need to update an interval, we will update a node and mark its child that it needs to be updated and update it when needed. For this we need an array **lazy** of the same size as that of a segment tree. Initially all the elements of the **lazy** array will be **0** representing that there is no pending update. If there is non-zero element in **lazy[i]**, then this element needs to update node **i** in the segment tree before making any query operations. How are we going to do that? Let's look at an example:

Let's say, for our example tree, we want to execute some queries. These are:

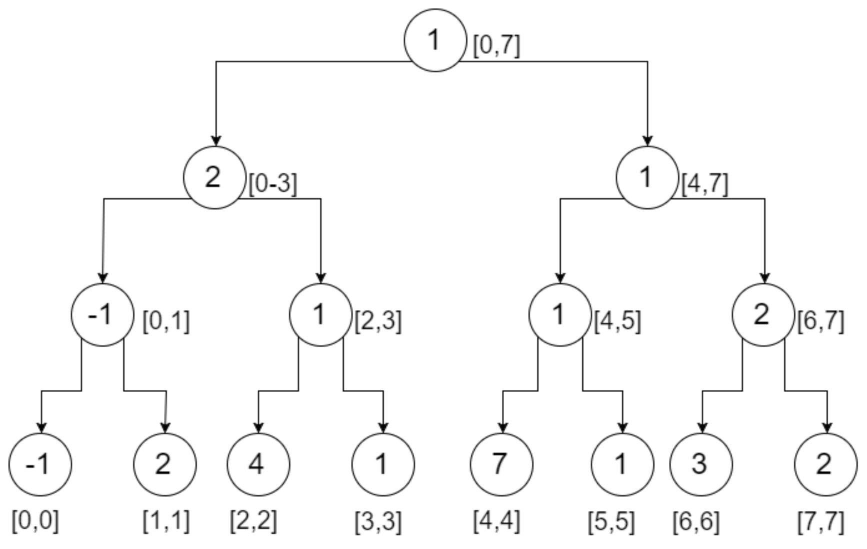
increment **[0,3]** by **3**.
increment **[0,3]** by **1**.
increment **[0,0]** by **2**.

increment [0,3] by 3:

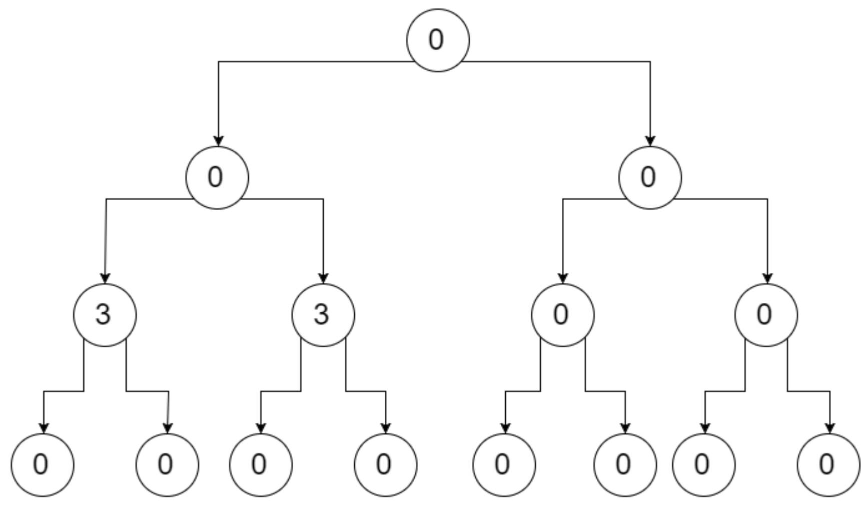
We start from the root node. At first, we check if this value is up-to-date. For this we check our **lazy** array which is **0**, that means the value is up-to-date. **[0,3]** partially overlaps **[0,7]**. So we go to both the directions.

At the left subtree, there's no pending update. **[0,3]** totally overlaps **[0,3]**. So we update the value of the node by **3**. So the value becomes **-1** + **3** = **2**. This time, we're not going to go all the way. Instead of going down, we update the corresponding child in the lazy tree of our current node and increment them by **3**. We also return the value of the current node.
At the right subtree, there's no pending update. **[0,3]** doesn't overlap **[4,7]**. So we return the value of the current node **(1)**.
The minimum of two returned values (**2**, **1**) is **1**. We update the root node to **1**.

Our Segment Tree and Lazy Tree would look like:



Segment Tree



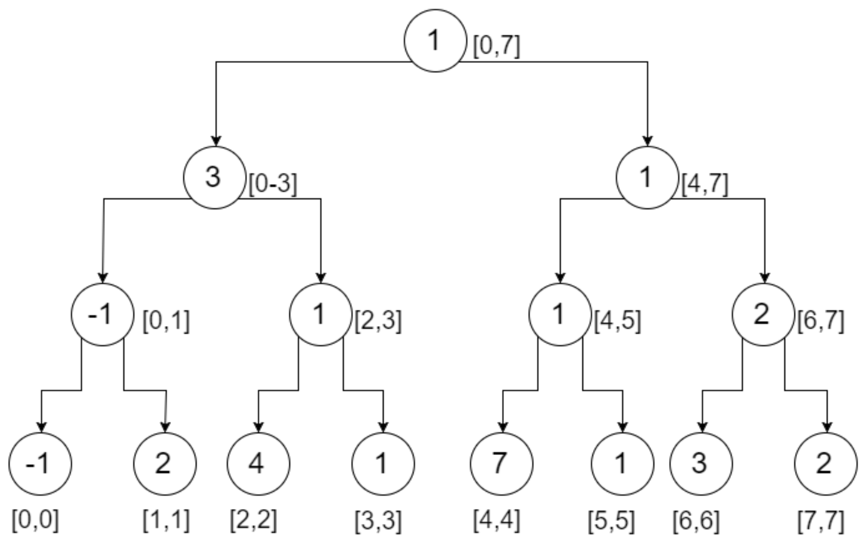
Lazy Tree

The non-zero values in nodes of our Lazy Tree represents, there are updates pending in those nodes and below. We'll update them if required. If we are asked, what is the minimum in range **[0,3]**, we'll come to the left subtree of the root node and since there's no pending updates, we'll return **2**, which is correct. So this process doesn't affect the correctness of our segment tree algorithm.

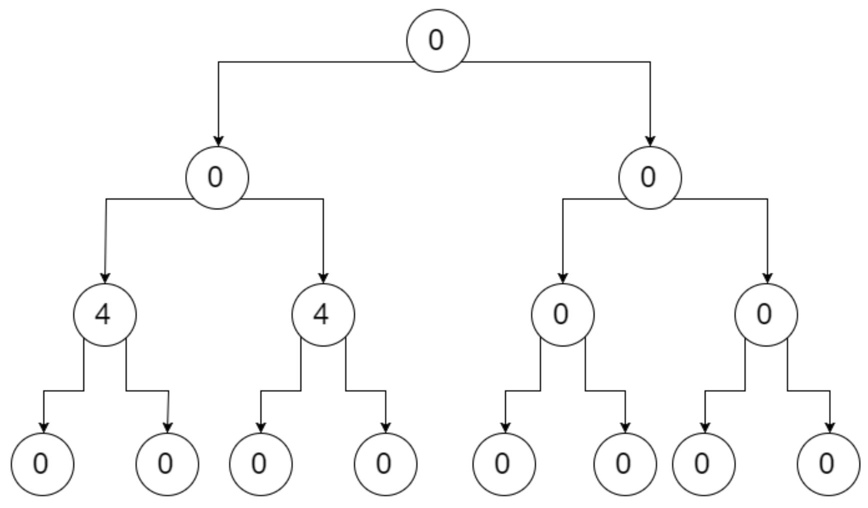
increment [0,3] by 1:

We start from the root node. There's no pending update. **[0,3]** partially overlaps **[0,7]**. So we go to both directions.
In the left subtree, there's no pending update. **[0,3]** completely overlaps **[0,3]**. We update the current node: **2** + **1** = **3**. Since this is an internal node, we update its children in the Lazy Tree to be incremented by **1**. We'll also return the value of the current node (**3**).
In the right subtree, there's no pending update. **[0,3]** doesn't overlap **[4,7]**. We return the value of the current node (**1**).
We update the root node by taking the minimum of two returned values(**3**, **1**).

Our Segment Tree and Lazy Tree will look like:



Segment Tree



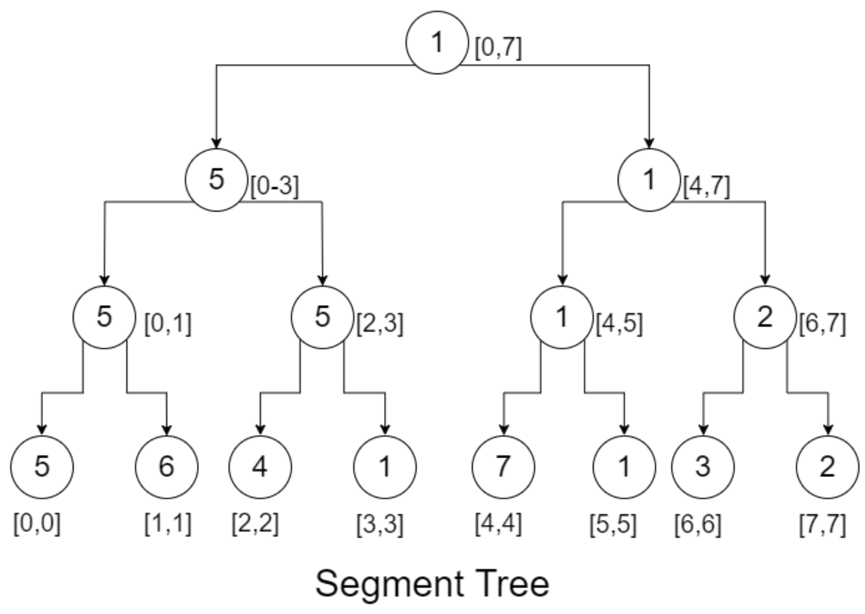
Lazy Tree

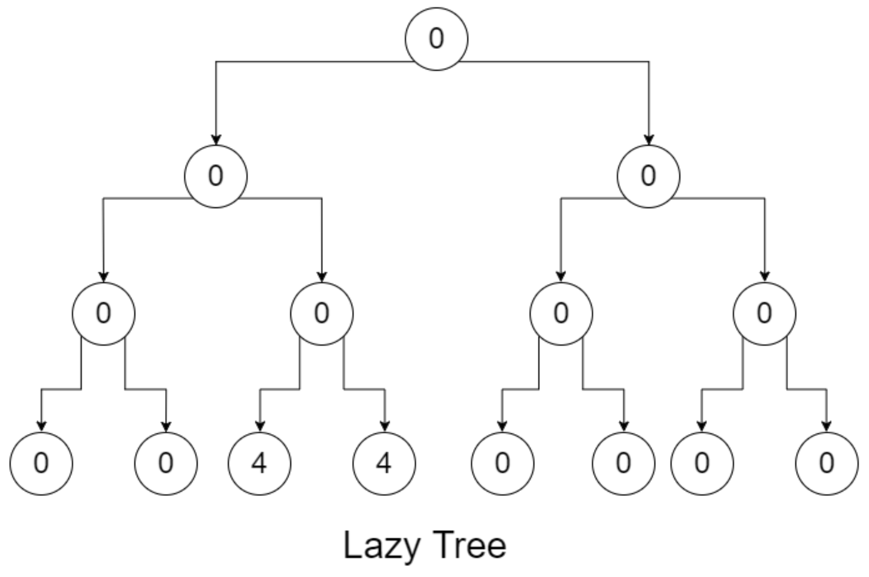
As you can see, we're accumulating the updates at Lazy Tree but not pushing it down. This is what Lazy Propagation means. If we hadn't used it, we had to push the values down to the leaves, which would cost us more unnecessary time complexity.

increment [0,0] by 2:

- We start from the root node. Since root is up-to-date and **[0,0]** partially overlaps **[0,7]**, we go to both directions.
  - At the left subtree, the current node is up-to-date and **[0,0]** partially overlaps **[0,3]**, we go to both directions.
    - At the left subtree, the current node in Lazy Tree has a non-zero value. So there is an update which has not been propagated yet to this node. We're going to first update this node in our Segment Tree. So this becomes **-1** + **4** = **3**. Then we're going to propagate this **4** to its children in the Lazy Tree. As we have already updated the current node, we'll change the value of current node in Lazy Tree to **0**. Then **[0,0]** partially overlaps **[0,1]**, so we go to both directions.
      - At the left node, the value needs to be updated since there is a non-zero value in the current node of Lazy Tree. So we update the value to **-1** + **4** = **3**. Now, since **[0,0]** totally overlaps **[0,0]**, we update the value of the current node to **3** + **2** = **5**. This is a leaf node, so we need not to propagate the value anymore. We update the corresponding node at the Lazy Tree to **0** since all the values have been propagated up till this node. We return the value of the current node(**5**).
      - At the right node, the value needs to be updated. So the value becomes: **4** + **2** = **6**. Since **[0,0]** doesn't overlap **[1,1]**, we return the value of the current node(**6**). We also update the value in Lazy Tree to **0**. No propagation is needed since this is a leaf node.
      - We update the current node using the minimum of two returned values(**5**,**6**). We return the value of the current node(**5**).
    - At the right subtree, there's a pending update. We update the value of the node to **1** + **4** = **5**. Since this is not a leaf node, we propagate the value to its children in our Lazy Tree and update the current node to **0**. Since **[0,0]** doesn't overlap with **[2,3]**, we return the value of our current node(**5**).
    - We update the current node using the minimum of the returned values(**5**, **5**) and return the value(**5**).
  - At the right subtree, there's no pending update and since **[0,0]** doesn't overlap **[4,7]**, we return the value of the current node(**1**).
- We update the root node using the minimum of the two returned values(**5**,**1**).

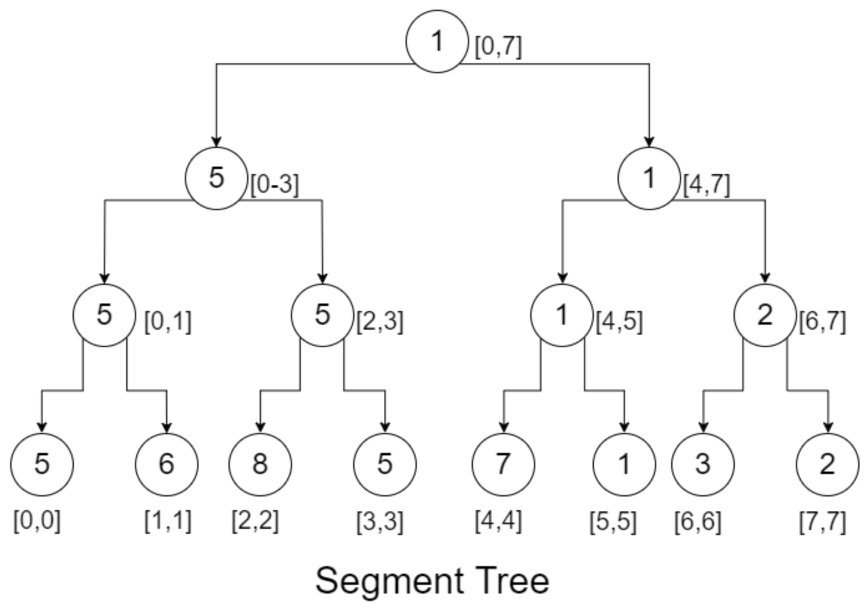
Our Segment Tree and Lazy Tree will look like:

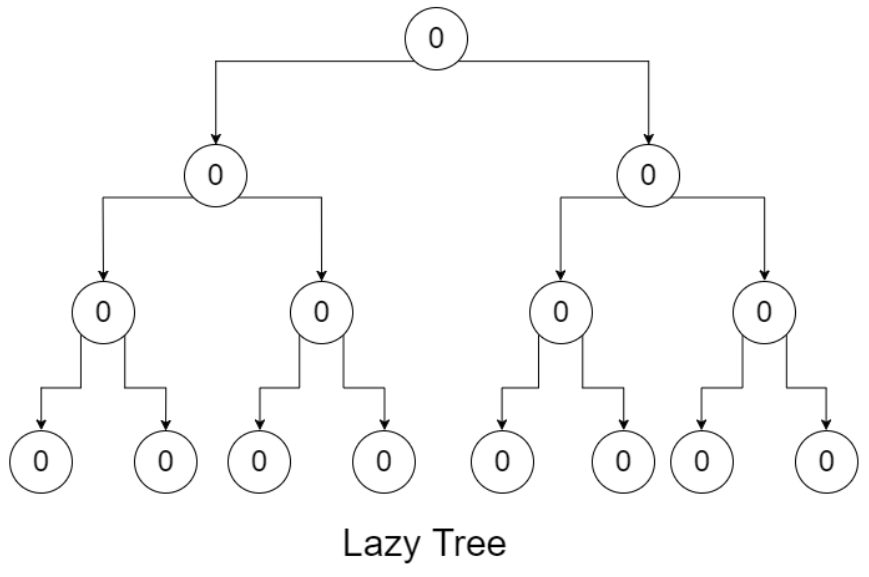




We can notice that, the value at **[0,0]**, when needed, got all the increment.

Now if you are asked to find the minimum in range **[3,5]**, if you have understood up to this point, you can easily figure out how the query would go and the returned value will be **1**. Our segment Tree and Lazy Tree would look like:





We have simply followed the same process we followed in finding RMQ with added constraints of checking the Lazy Tree for pending updates.

Another thing we can do is instead of returning values from each node, since we know what will be the child node of our current node, we can simply check them to find the minimum of these two.

The pseudo-code for updating in Lazy Propagation would be:

```
Procedure UpdateSegmentTreeLazy(segmentTree, LazyTree, startRange,
                                endRange, delta, low, high, position):
if low > high                                                //out of bounds
    Return
end if
if LazyTree[position] is not equal to 0                      //update needed
    segmentTree[position] := segmentTree[position] + LazyTree[position]
    if low is not equal to high                              //non-leaf node
        LazyTree[2 * position + 1] := LazyTree[2 * position + 1] + delta
        LazyTree[2 * position + 2] := LazyTree[2 * position + 2] + delta
    end if
    LazyTree[position] := 0
end if
if startRange > low or endRange < high                       //doesn't overlap
    Return
end if
if startRange <= low && endRange >= high                     //total overlap
    segmentTree[position] := segmentTree[position] + delta
    if low is not equal to high
        LazyTree[2 * position + 1] := LazyTree[2 * position + 1] + delta
        LazyTree[2 * position + 2] := LazyTree[2 * position + 2] + delta
    end if
    Return
end if
//if we reach this portion, this means there's a partial overlap
mid := (low + high) / 2
UpdateSegmentTreeLazy(segmentTree, LazyTree, startRange,
                                endRange, delta, low, mid, 2 * position + 1)
UpdateSegmentTreeLazy(segmentTree, LazyTree, startRange,
                                endRange, delta, mid + 1, high, 2 * position + 2)
segmentTree[position] := min(segmentTree[2 * position + 1],
                                                    segmentTree[2 * position + 2])
```

And the pseudo-code for RMQ in Lazy Propagation will be:

```
Procedure RangeMinimumQueryLazy(segmentTree, LazyTree, qLow, qHigh, low, high, position):
if low > high
    Return infinity
end if
if LazyTree[position] is not equal to 0                      //update needed
    segmentTree[position] := segmentTree[position] + LazyTree[position]
    if low is not equal to high
        segmentTree[position] := segmentTree[position] + LazyTree[position]
    if low is not equal to high                              //non-leaf node
        LazyTree[2 * position + 1] := LazyTree[2 * position + 1] + LazyTree[position]
        LazyTree[2 * position + 2] := LazyTree[2 * position + 2] + LazyTree[position]
    end if
    LazyTree[position] := 0
end if
if qLow > high and qHigh < low                                //no overlap
    Return infinity
end if
if qLow <= low and qHigh >= high                              //total overlap
    Return segmentTree[position]
end if
//if we reach this portion, this means there's a partial overlap
mid := (low + high) / 2
Return min(RangeMinimumQueryLazy(segmentTree, LazyTree, qLow, qHigh,
                                        low, mid, 2 * position + 1),
           RangeMinimumQueryLazy(segmentTree, LazyTree, qLow, qHigh,
                                        mid + 1, high, 2 * position + 1)
```

‹ Previous Next ›

PDF - Download **data-structures** for free