Segment Trees in Algorithmic Problems

Piotr Szczepaniak

June 23, 2025

Contents

Abstract

This document provides an overview of segment trees. In the first place I will describe some basic implementation and use case for segment trees. Then we will try to understand the algebraic properties of segment trees for better understanding of how they work. Finally we will dive in some more complex and not so obvious use cases of segment trees.

1 Foundations of Segment Trees

Basic use case of segment tree is to store information about segments of an array. Consider an array A with some data stored in it. To be able to perform operations on segments of this array, like give the max value in $A[i, j]$ or give the sum of elements in $A[i, j]$, we can use a segment tree for faster access answers to these queries. Additionally segment tree allows us to update values in the array and still be able to perform queries on segments quickly. A segment tree is a binary tree used for storing information about segments with quick access to the data stored in it. To efficiently retrieve or update information about elements stored in segment tree we can perform various operations most common of which are query/update on point or range. One of the examples can be maximum value of elements in given range or sum of elements in given range.

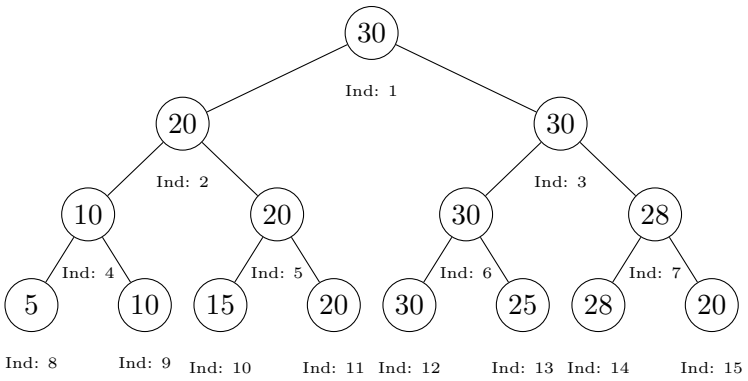

Figure 1: Example of a segment tree with maximum value of elements in given range.

1.1 Segment Tree with point update and range query

To illustrate use case of segment tree we will construct tree with max value on segment. Let's say we are given an array $A = [5, 10, 15, 20, 30, 25, 28, 20]$ of length $n = 8$. For now let's assume that the input array is of size $n = 2^k$ where k is integer (for different sizes of input we will fill input array with neutral elements (see section 2) to make it's length a power of 2). In the following implementation a segment tree is actually an array of size $2n - 1$ where we store the whole structure. The leaves of the tree will store the values of the array and the internal nodes will store the maximum value of the elements in their subtree. For every internal node i the children of this node will be $2i$ and $2i + 1$, and the root of the tree will be at index 1. We want to be able to perform the following operations:

- **Build structure**
  We will create a segment tree from an array. To build a segment tree, we need to create a binary tree where each node will store the maximum value of elements in its subtree. First we copy elements of array to the leaves of the tree (as $max(A[i, i]) = A[i]$), then we build the tree from bottom to the top

Algorithm 1 Build Segment Tree for Maximum on Segment (Iterative)

```
procedure BuildTree(arr, seg)
    for i = 0 to n − 1 do                          ▷ Fill leaves of the segment tree
        seg[n + i] ← A[i]
    end for
    for i = n − 1 downto 1 do                      ▷ Calculates nodes from bottom to top
        seg[i] ← max(seg[2 × i], seg[2 × i + 1])
    end for
end procedure
```

- **Point update**
  Now let's update single value in the array and update the tree. We will change the value of $A[1]$ from 10 to 35. To update the tree we need to change the value of the leaf node and then update all the parent nodes up to the root.

Algorithm 2 Point Update on Segment Tree

procedure UPDATE(seg, index, value)
  $index \leftarrow index + n$ ▷ Shift index to leaf
  $arr[index] \leftarrow value$ ▷ Update the value at the leaf
  while $index > 1$ **do** ▷ Update the parent nodes
    $index \leftarrow \lfloor index/2 \rfloor$
    $arr[index] \leftarrow \max(arr[2 \times index], arr[2 \times index + 1])$
  end while
end procedure

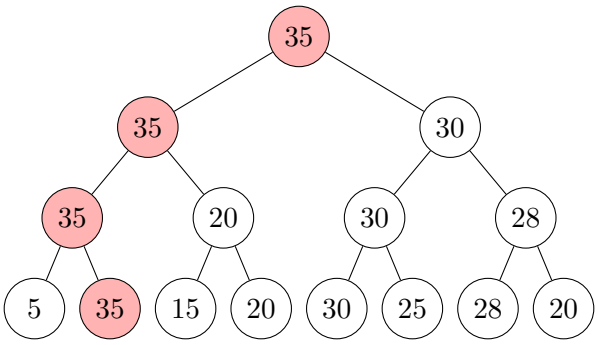

Figure 2: Example of point update.

- **Range query**
  Now let's say we want to find the maximum value in some range. To do this we need to traverse the tree from the root to the leaves and find nodes that are in the range (which means that every leaf in the subtree of this node is in the range of the query). Then we will get max the values of these nodes to get the final result.

Algorithm 3 Range Maximum Query on Segment Tree (Recursive, Close-Open Range)

procedure RANGEQUERY(seg, index, l, r, a, b) ▷ index: current node index in seg ▷ [l, r): segment represented by current node ▷ [a, b): query range
  if $b \leq l$ **or** $r \leq a$ **then**
    return $-\infty$ ▷ No overlap
  else if $a \leq l$ **and** $r \leq b$ **then**
    return $seg[index]$ ▷ Total overlap
  else
    $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$
    $left \leftarrow$ RANGEQUERY(seg, $2 \cdot index$, l, mid, a, b)
    $right \leftarrow$ RANGEQUERY(seg, $2 \cdot index + 1$, mid, r, a, b)
    return $\max(left, right)$
  end if
end procedure

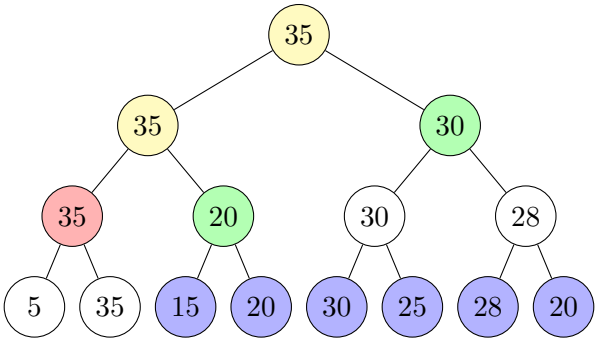

Figure 3: Example of range query. Blue leaves represents subarray $A[2:7]$. Green nodes are the nodes that totally overlap, red ones that don't overlap and yellow ones that partially overlap (are called in recursion).

1.2 Segment Tree with range update and point query

This type of segment tree is very similar to previous one but in this case we will be able to perform updates on given range and ask what is the value of an array at given index. In the update operation we will update all elements in the range by adding some value to them and in the query operation we will ask for the value of the array at given index. The difference from the previous example is that in the nodes of the tree don't keep the actual values of the array but some value that modifies the value from array. Let's take example where we add value to all elements of array on the given range. If the node has value x it means that we added x to all elements in the range of this node. So in order to get the value of the array at given index we need to traverse the tree from the leaf to the root and sum all values of the nodes we passed.

- **Build structure**
  To create a tree we only copy elements of the array to the leaves of the tree, because there are no updates yet.
- **Range update**
  Here we want to add some given value x to all elements in the range $A[a:b]$. This operation is similar to the range query from previous tree but here instead of returning the maximum value we will update the value of the node.

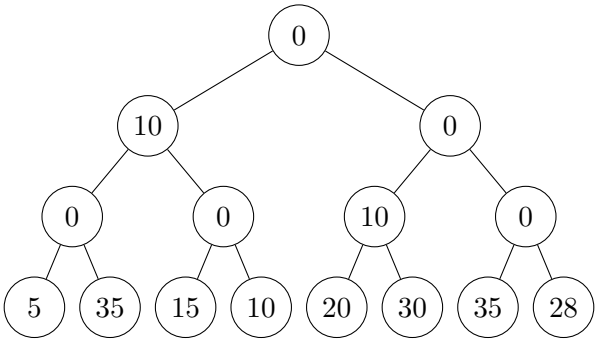

Figure 4: Example of range update. This is a state of the tree for initial array $A = [5, 35, 15, 10, 20, 30, 25, 28]$ after range update on range $A[0:6]$ by adding value 10.

Algorithm 4 Range Update on Segment Tree (Recursive, Close-Open Range)

```
1: procedure RangeUpdate(seg, index, l, r, a, b, x)
2:     if b ≤ l or r ≤ a then
3:         return
4:     else if a ≤ l and r ≤ b then
5:         seg[index] ← seg[index] + x
6:     else
7:         mid ← ⌊(l+r)/2⌋
8:         RangeUpdate(seg, 2 · index, l, mid, a, b, x)
9:         RangeUpdate(seg, 2 · index + 1, mid, r, a, b, x)
10:        return
11:    end if
12: end procedure
```

- **Point query**
  Let's consider what is stored in the nodes of the segment tree and how to retrieve the value of the array at a given index i. When we perform updates that affect the position i, we add values to the nodes along the path from the corresponding leaf to the root. Each leaf node represents a modifier applied to the neutral element 0. The result of a point query at index i is the sum of all the values stored in the nodes along the path from the leaf representing i up to the root. This sum represents the combined effect of all updates (modifiers) applied to the neutral element 0 along that path. We use an iterative approach to perform this query because a recursive solution would be more complex.

Algorithm 5 Point Query on Segment Tree

```
procedure query(seg, index)
    index ← index + n                          ▷ Shift index to leaf
    result ← 0                                 ▷ Initialize result with the neutral value
    while index ≥ 1 do
        result ← result + arr[index]
        index ← ⌊index/2⌋
    end while
end procedure
```

If we for example ask for the value at index 0 of the array after the update visualized in the tree above, we will get the value 15.

1.3 Segment Tree with Range Update and Range Maximum Query

The range update, range query tree is actually a combination of two previous types of tree. In this version we will be able to update a range of elements by adding a specified value to all elements within that range, and we can also retrieve the maximum value over a specified range. For this tree in nodes we need to store a value (like in the 1.1 section) and modifiers (like in the 1.2 section).

- **Build Structure**
  We need to initialize the values (max on subtree) and modifiers (neutral for now).

Algorithm 6 Build Segment Tree for Sum on Segment (Iterative)

```
procedure BuildTree(arr, seg)
    for i = 0 to n − 1 do                          ▷ Fill leaves of the segment tree
        seg[n + i] ← {A[i], 0}                     ▷ second value is for lazy propagation
    end for
    for i = n − 1 downto 1 do                      ▷ Calculates nodes from bottom to top
        seg[i] ← {max(seg[2 × i], seg[2 × i + 1]), 0}
    end for
end procedure
```

- **Range Update**
  When we find the node fully in given range we update the modifier and also add the value to the node's maximum value. Going back from the recursion we have to update the maximum value of the node based on its children, as the update may have changed the maximum value of the subtree.

Algorithm 7 Range Update on Segment Tree

```
procedure RangeUpdate(seg, index, l, r, a, b, value)
    if b ≤ l or r ≤ a then
        return
    else if a ≤ l and r ≤ b then
        seg[index].val ← seg[index].val + value
        seg[index].mod ← seg[index].mod + value
        return
    else
        mid ← ⌊(l+r)/2⌋
        RangeUpdate(seg, 2 · index, l, mid, a, b, value)
        RangeUpdate(seg, 2 · index + 1, mid, r, a, b, value)
        Pull(seg, index)
    end if
end procedure

procedure Pull(seg, index)
    seg[index].val ← max(seg[2 · index].val, seg[2 · index + 1].val) + seg[index].mod
end procedure
```

For example, consider updating the range $A[0:5]$ by adding 10 to each element, where the initial array is $A = [5, 10, 15, 20, 30, 25, 20, 20]$.

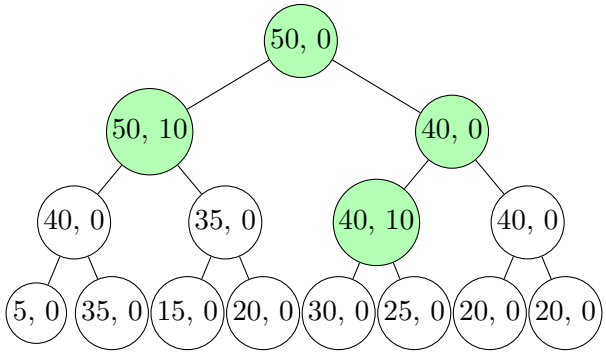

Figure 5: State of the tree after the range update. Green nodes indicate segments changed during the update.

- **Range Maximum Query**
  To retrieve the maximum value over a specified range, the query algorithm traverses tree the same way as the update algorithm. As we go deeper in recursion we have to consider all modifiers on the way. This is a similar situation like in point query in section 1.2 however this time we traverse the tree top to bottom.

Algorithm 8 Range Query on Segment Tree for Max with Modifiers

```
procedure RangeQuery(seg, index, l, r, a, b)
    if b ≤ l or r ≤ a then
        return −∞
    end if
    if a ≤ l and r ≤ b then
        return seg[index].val
    end if
    mid ← ⌊(l+r)/2⌋
    left ← RangeQuery(seg, 2 · index, l, mid, a, b)
    right ← RangeQuery(seg, 2 · index + 1, mid, r, a, b)
    return max(left, right) + seg[index].mod
end procedure
```

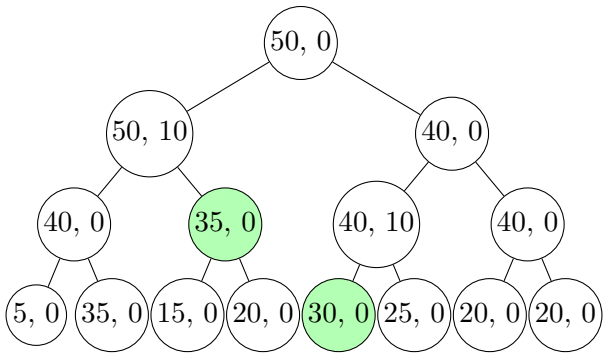

Figure 6: State of the tree after the range maximum query on the range [2:4]. Green nodes indicate those contributing to the query result. The result of the query is 45.

1.4 Lazy Propagation

The segment tree structure we have described so far works correctly for operations that are commutative and associative — for example, adding a value to all elements of a range or querying the maximum on a range. In these cases, the order in which we apply updates does not affect the final result.

However, this approach fails if we attempt to apply non-commutative or more complex operations (such as assignment of values). The main reason is that modifiers may no longer combine in a well-defined way, and the final value would depend on the order of updates.

To support such operations, we must introduce **lazy propagation**. In lazy propagation, we delay pushing updates to the children nodes until it is necessary — for example, when a query or further update requires the current node's children to be correct.

Below we provide a version of the `Push` procedure that applies the modifier from a parent node to its children:

Algorithm 9 Push Operation for Lazy Propagation (Range Max with Assignment as Example)

```
procedure PUSH(seg, index)
    if seg[index].mod ≠ 0 then
        seg[2 · index].val ← seg[2 · index].val + seg[index].mod
        seg[2 · index].mod ← seg[2 · index].mod + seg[index].mod
        seg[2 · index + 1].val ← seg[2 · index + 1].val + seg[index].mod
        seg[2 · index + 1].mod ← seg[2 · index + 1].mod + seg[index].mod
        seg[index].mod ← 0
    end if
end procedure
```

This `Push` method ensures that before accessing or updating children, any pending modification is first applied to them, and the current node's modifier is cleared. The actual implementation of `Push` will depend on the type of operation (e.g., addition, assignment, minimum constraint). This operation should be put right before the recursive calls in the range update, keep in mind that now in range update we don't apply the modifier when calling pull method.

2 A mathematical approach to segment trees

In this section I will present how segment trees work from a mathematical point of view. In order to do this we will need to understand some basic algebra that is used in segment trees.

2.1 Monoids

A monoid $(S, *, e)$ is a set equipped with an associative binary operation $S \times S \to S$ and an identity element e.

- **Associativity**
  For all $a, b, c \in S$, $(a * b) * c = a * (b * c)$.
- **Identity element**
  There exists an element $e \in S$ such that for all $a \in S$, $a * e = e * a = a$.
- **Commutativity**
  Monoid is called **commutative** if for all $a, b \in S$, $a * b = b * a$.

Examples of monoids:

- **Natural numbers under addition, ($\mathbb{N}$, +, 0)**
  In this simple example we can see that addition is associative and commutative. Adding 0 (neutral element) to any of the numbers will not change the result of the operation.
- **Strings over an alphabet, (Σ^*, $\cdot$, ϵ)**
  Σ^* - set of all strings over an alphabet Σ
  $\cdot$ - concatenation of strings
  ϵ - empty string
  This is a example of non-commutative monoid. Concatenation of strings is associative but not commutative as we can see in the following example.

$$\text{abc} \cdot \text{def} = \text{abcdef} \neq \text{defabc} = \text{def} \cdot \text{abc} \tag{1}$$

2.2 Application of Monoids to Segment Trees

How does this all relate to segment trees? First of all we can see that segment trees requires the properties of monoids to work. The associativity of the operation is required to be able to combine the results of the operations on the segments. The identity element is required when we ask for the result of the operation on an empty segment or when we want extend our initial array to a power of 2. Since all segment trees must preserve monoid properties, we can think more generic approach to the implementation. We will create a generic segment tree that can be used for any monoid for range queries and point updates. Let $*$ be a binary operation on a monoid S and let e be the identity element of the monoid.

Algorithm 10 Segment Tree over a Monoid $(S, *, e)$

```
1: procedure BuildTree(A, seg, e)
2:     n ← next power of two greater than or equal to |A|
3:     for i = 0 to |A| − 1 do
4:         seg[n + i] ← A[i]                                  ▷ Insert original values
5:     end for
6:     for i = |A| to n − 1 do
7:         seg[n + i] ← e                                     ▷ Pad remaining leaves with identity
8:     end for
9:     for i = n − 1 downto 1 do
10:        seg[i] ← seg[2 × i] ∗ seg[2 × i + 1]
11:    end for
12: end procedure
13:
14: procedure Update(arr, index, value)
15:    index ← index + n                                      ▷ Move to the leaf node
16:    arr[index] ← value                                     ▷ Set the new value at the leaf
17:    while index > 1 do
18:        index ← ⌊index/2⌋                                  ▷ Move to parent
19:        arr[index] ← arr[2 × index] ∗ arr[2 × index + 1]
20:    end while
21: end procedure
22:
23: procedure RangeQuery(seg, index, l, r, a, b, e)   ▷ index: current node in segment tree ▷
    [l, r]: range covered by node                                  ▷ [a, b]: query range
24:    if b ≤ l or r ≤ a then
25:        return e                                           ▷ No overlap
26:    else if a ≤ l and r ≤ b then
27:        return seg[index]                                  ▷ Total overlap
28:    else
29:        mid ← ⌊(l+r)/2⌋
30:        left ← RangeQuery(seg, 2 · index, l, mid, a, b, e)
31:        right ← RangeQuery(seg, 2 · index + 1, mid, r, a, b, e)
32:        return left ∗ right
33:    end if
34: end procedure
```

Now we can replace this generic monoid with some specific one. For example we can use a segment tree for sum of elements in given range. In this case we will use $*$ as addition and e as 0. Another example can be a segment tree for maximum value in given range. In this case we will use $*$ as maximum and e as $-\infty$. We can see that using this generic approach we can create a segment tree for any monoid.

2.3 Homomorphism

A homomorphism is a structure-preserving map between two algebraic structures. In this section we will look at homomorphisms between monoids. Let $(S, *, e)$ and $(T, \cdot, \theta)$ be two monoids. A function $f : S \to T$ is a homomorphism if:

- **Preserves the operation**
  For all $a, b \in S$, $f(a * b) = f(a) \cdot f(b)$.

Let's consider a simple example of homomorphism between two monoids. Let $S = (\mathbb{N}, +, 0)$ and $T = (\mathbb{N}, \cdot, 1)$. Let $f : S \to T$ be defined as $f(x) = 2^x$. We can see that:

- $f(a + b) = 2^{a+b} = 2^a \cdot 2^b = f(a) \cdot f(b)$
- $f(0) = 2^0 = 1$

2.4 Endomorphism

A special case of a homomorphism is an *endomorphism*. An endomorphism is a homomorphism from a monoid to itself.

Let us consider a simple example of an endomorphism. Let $S = (\mathbb{N}, +, 0)$ be the monoid of natural numbers under addition. Define $f : S \to S$ by $f(x) = 2x$.

We can verify that f is a homomorphism by checking:

$$f(a + b) = 2(a + b) = 2a + 2b = f(a) + f(b)$$

An interesting property of endomorphisms is that the set of all endomorphisms of a monoid forms a monoid itself.

Let E be the set of all endomorphisms of the monoid S. We define a binary operation $*$ on E by composition:

$$(f * g)(x) = f(g(x))$$

This operation is associative, and the identity element is the identity endomorphism e defined by $e(x) = x$. Hence, $(E, *, e)$ is a monoid.

2.5 Application of Endomorphisms to Segment Trees

In this section we will implement a generic segment tree for range update and range query. The operations in the segment tree are now defined over a monoid $(S, *, e)$ and we will use an endomorphism $f : S \to S$ to modify the values in the segment tree.

Algorithm 11 Range Update on Segment Tree over value Monoid and modifier Monoid

```
1: procedure RangeUpdate(seg, index, l, r, a, b, f)
2:     if b ≤ l or r ≤ a then
3:         return
4:     else if a ≤ l and r ≤ b then
5:         seg[index].val ← f(seg[index].val)
6:         seg[index].mod ← seg[index].mod ∗ f
7:         return
8:     else
9:         mid ← ⌊(l + r)/2⌋
10:        RangeUpdate(seg, 2 · index, l, mid, a, b, f)
11:        RangeUpdate(seg, 2 · index + 1, mid, r, a, b, f)
12:        Pull(seg, index)
13:    end if
14: end procedure
15:
16: procedure RangeQuery(seg, index, l, r, a, b)
17:    if b ≤ l or r ≤ a then
18:        return e
19:    else if a ≤ l and r ≤ b then
20:        return seg[index].val
21:    else
22:        mid ← ⌊(l + r)/2⌋
23:        left ← RangeQuery(seg, 2 · index, l, mid, a, b)
24:        right ← RangeQuery(seg, 2 · index + 1, mid, r, a, b)
25:        return f(left ⊕ right)
26:    end if
27: end procedure
28:
29: procedure Pull(seg, index)
30:    seg[index].val ← f(seg[2 · index].val ⊕ seg[2 · index + 1].val)
31: end procedure
```

Using this generic approach we can simply replace the monoid with some specific one and set endomorphism as some function that we want to apply to the range of elements.

2.6 Examples of Monoids and Operations for Segment Trees

Here are some practical examples of monoids and operations suitable for segment trees with range updates and queries:

1. **Change values on a range, get minimum on a range**

   - Value monoid: $(S, \oplus, e) = (\mathbb{Z}, \min, +\infty)$ — stores the minimum value on a segment.
   - Modifier monoid: $(M, \star, \mathrm{id}) = (\mathbb{Z} \cup \{\perp\}, \text{overwrite}, \perp)$ — assigns a value to the whole segment.

- Apply function:

$$f(x) = \begin{cases} f & \text{if } f \neq \bot \\ x & \text{if } f = \bot \end{cases}$$

2. **Add a string to all elements in the array, get the string with the most letters 'a' in a range**

   - Value monoid: $(S, \oplus, e) = (\text{Strings}, \max_a, "")$, where $\max_a$ returns the string with the highest count of letter 'a'.
   - Modifier monoid: $(M, \star, \text{id}) = (\text{Strings}, \text{concatenate}, "")$ — concatenates strings as updates.
   - Apply function:

$$f(x) = x \text{ concatenated with } f$$

3. **Range updates with addition, query maximum prefix sum on a range**

   - Value monoid: $S = \{\text{max prefix sum, total sum}\}$ with operation combining prefix sums properly.
   - Modifier monoid: $(M, \star, \text{id}) = (\mathbb{Z}, +, 0)$ — add values to elements.
   - Apply function:

$$f(x) = \text{update prefix sums and total sums by adding } f$$

Note: The exact structure and merge rules of complex monoids (like prefix sums or strings) need to be carefully defined to maintain associativity and identity elements.

3 Binsearch on Segment Tree

In this section we will see that we can use segment trees not only for queries and updates but also for binary search. Let's consider the following problem:

Given an array $a[1 \dots n]$, answer a query $Q(i, j, x)$: What is the smallest index k so that $a[k] \geq x$ in subarray $a[i \dots j]$?

We want to be able to answer this query in $O(\log n)$ time. We can create a segment tree with max value in given range but this is not enough. We need to upgrade our segment tree with binsearch operation, which goes as follows:

- Traverse recursively the tree to find the nodes that are in the range of the query (just like in range query). Always go to the left child first.
- For every nodes perform a binary search on the subtree.
- Binsearch
  - if right child's value is less than x, return -1 (no such index)
  - if left child's value is greater than x, go to left child
  - else go to right child

- Because we traverse the tree from left to right, we can be sure that the first binsearch that returns index different than -1 is the answer to the query.
- if no such index was found, return -1.

Algorithm 12 Find First Index With Value Greater Than x in Range $[a, b)$

```
1: procedure QUERY(segTree, index, l, r, a, b, x)  ▷ index: current node index  ▷ [l, r): segment represented by current node  ▷ [a, b): query range
2:     if b ≤ l or r ≤ a then
3:         return −1  ▷ No overlap
4:     else if a ≤ l and r ≤ b then
5:         return BINARYSEARCH(segTree, index, x)
6:     else
7:         mid ← ⌊(l+r)/2⌋
8:         left ← QUERY(segTree, 2 · index, l, mid, a, b, x)
9:         if left ≠ −1 then
10:            return left
11:        end if
12:        return QUERY(segTree, 2 · index + 1, mid, r, a, b, x)
13:    end if
14: end procedure
15:
16: procedure BINARYSEARCH(tree, index, x)
17:    while index ∗ 2 < tree.size do
18:        left ← tree[index ∗ 2]
19:        if left > x then
20:            index ← index ∗ 2
21:        else
22:            index ← index ∗ 2 + 1
23:        end if
24:    end while
25:    if tree[index] > x then
26:        return index
27:    else
28:        return −1  ▷ No index found
29:    end if
30: end procedure
```

Let's consider the following example:
We have an array $A = [5, 8, 2, 7, 1, 11, 13, 12, 19, 14, 15, 0, 15, 10, 15, 4]$ and we want to find the smallest index k such that $A[k] \geq 14$ in the range $A[5 : 12]$.

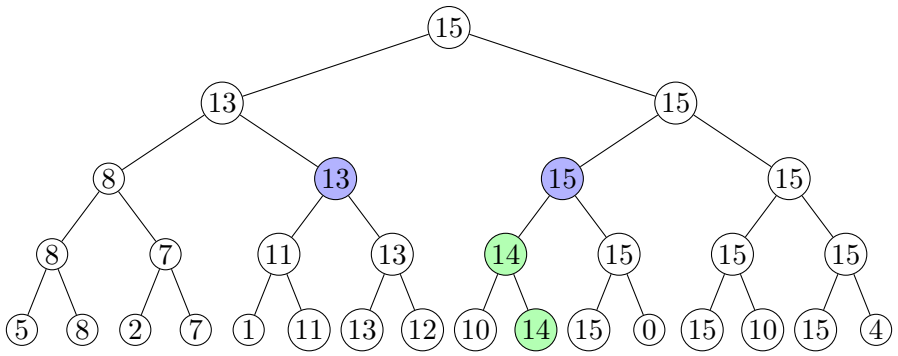

Figure 7: The blue nodes are the ones that we performed binsearch on. The green nodes are the ones that binsearch traversed. The result of the query is index 10. As we can see, the first binsearch (on node with value 13 returned -1)