Segment Trees in Algorithmic Problems

Piotr Szczepaniak

May 11, 2025

Contents

Abstract

This document provides an overview of segment trees. In the first place I will describe some algebraic topics which are necessary for better understanding how and why segment trees works. This knowledge will be useful for reading the rest of the paper where We will dive into different kinds of trees. For each structure, I will explain how it work and how to apply it to problems. Then, I will look at each structure's time complexity and space complexity.

1 Foundations of Segment Trees

A segment tree is a binary tree used for storing information about segments. To efficently retrieve or update informations 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.

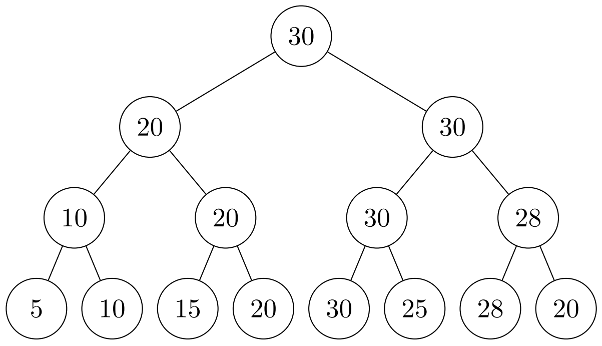

Figure 1: Example of a segment tree with maximum value of elements in given range.

1.1 Segment Tree with point update and range query

To ilustrate 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). The height of tree is $h = \log_2 n$. Let's define $dep(i)$ as depth of node i in our tree. We can see that $dep(root) = 0$ and $dep(leaf) = h$. 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.

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
        $seg[n + i] \leftarrow A[i]$
    end for
    for $i = n - 1$ downto 1 do          ▷ Calculates nodes from bottom to top
        $seg[i] \leftarrow \max(seg[2 \times i], seg[2 \times i + 1])$
    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
```

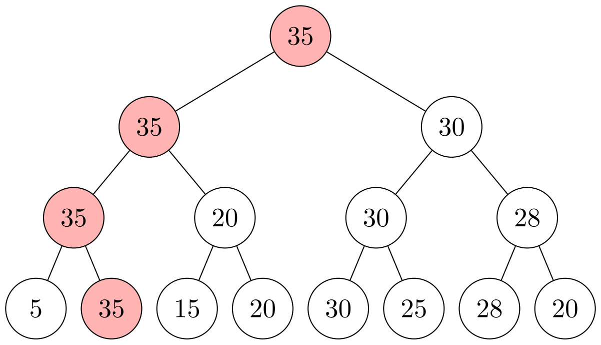

Figure 2: Example of point update.

- **Range query**
  Now let's say we want to find the maximum value in the range $A[2:7]$. To do this we need to traverse the tree from the root to the leaves and find nodes that are in the range. Then we will get max the values of these nodes to get the final result. To get the result for range $A[2:7]$ we call $RangeQuery(seg, 1, 1, 8, 2, 7)$.

Algorithm 3 Range Maximum Query on Segment Tree (Recursive)

```
1: 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
2:     if $b < l$ or $a > r$ then
3:         return $-\infty$                                ▷ No overlap
4:     else if $a \le l$ and $r \le b$ then
5:         return $seg[index]$                             ▷ Total overlap
6:     else
7:         $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$
8:         $left \leftarrow$ RangeQuery(seg, $2 \cdot index$, $l$, $mid$, $a$, $b$)
9:         $right \leftarrow$ RangeQuery(seg, $2 \cdot index + 1$, $mid + 1$, $r$, $a$, $b$)
10:        return max($left, right$)
11:    end if
12: end procedure
```

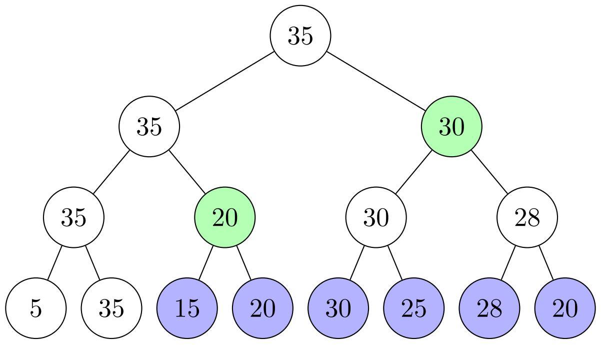

Figure 3: Example of range query. Blue leafs represents subarray $A[2 : 7]$. Green nodes represents nodes where ranges totally overlap and we can get max value. The result of the query is 30.

1.2 Segment Tree with range update and range query

A Little more complex version of segment tree is a segment tree with range update and range query which require **lazy propagation**. We will consider a segment tree with sum of elements in given range. Apart form values in tree we will also store lazy values in each node. The lazy value is used to delay the update of a node until it is needed.

- **Build structure** Only difference form previus example is initializig the lazy value for all nodes

Algorithm 4 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
        $\{seg[n + i] \leftarrow A[i], 0\}$        ▷ second value is for lazy propagation
    end for
    for $i = n - 1$ downto 1 do                    ▷ Calculates nodes from bottom to top
        $seg[i] \leftarrow \{seg[2 \times i] + seg[2 \times i + 1], 0\}$
    end for
end procedure
```

- **Range update**
  In this case lazy value is a value that we want to add to all elements in the range. Another way to think about this is that we want to add lazy value to all elenets in arrary which are represented by all leaves in subtree of given node.

Algorithm 5 Range Update on Segment Tree

```
procedure QueryUpdate(seg, lazy index, l, r, a, b, value)   ▷ Value: value
we wanted to add to each element in array
    $size = r - l + 1$
    if $b < l$ or $a > r$ then
        return
    else if $a \leq l$ and $r \leq b$ then
        $seg[index] \leftarrow seg[index] + value * size$
        $lazy[index] \leftarrow lazy[index] + value$
        return
    else
        $seg[index * 2] \leftarrow seg[index * 2] + value * size$              ▷ Update left child
        $lazy[index * 2] \leftarrow lazy[index * 2] + value$                   ▷ Push lazy to left child
        $seg[index * 2 + 1] \leftarrow seg[index * 2 + 1] + value * size$      ▷ Same to right
        $lazy[index * 2 + 1] \leftarrow lazy[index * 2 + 1] + value$
        $lazy[index] \leftarrow 0$                                             ▷ Reset lazy value for current node
        $mid \leftarrow \left\lfloor \frac{l+r}{2} \right\rfloor$
        $left \leftarrow$ QueryUpdate(seg, $2 \cdot index$, $l$, $mid$, $a$, $b$, $value$)
        $right \leftarrow$ QueryUpdate(seg, $2 \cdot index + 1$, $mid + 1$, $r$, $a$, $b$, $value$)
        $seg[index] \leftarrow seg[index * 2] + seg[index * 2 + 1] + lazy[index] * size$
▷ Update current node after lazy propagation
    end if
end procedure
```

How this algorithm works:

1. If the current node is fully in range update the sum of the node and set the lazy value. Sum - $value * size$ where $size$ is the number of

array elements (leafs) in the range of this node, Lazy is equal to *value*, which we will later push down to child nodes.

2. If not fully in range push the lazy value down to the child nodes and update their sum and also reset the lazy value for the current node.
3. Call the function recursively for the left and right child nodes.
4. Finally update the sum of the current node with the sum of its children.

On the following example we will update the range $A[0:5]$ by adding 10 to all elements in this range where initial array is $A = [5, 10, 15, 20, 30, 25, 20, 20]$.

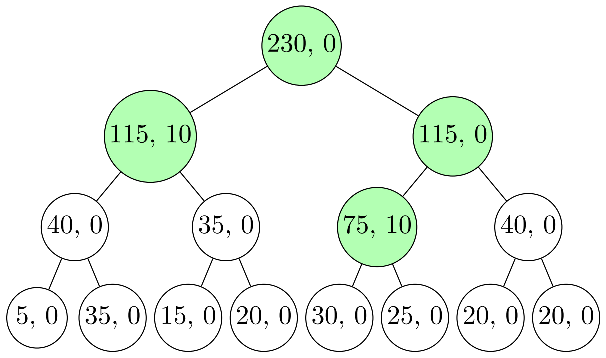

Figure 4: This is state of the tree after range update. Green nodes were changed during the update.

- **Range query** We will try to retrieve the sum of some given range. The algorithm for this will be simiar to the one for range update. First we will check if the current node is fully in range. If it is we add the sum of this node, propagating lazy values along the way.

Algorithm 6 Range Query on Segment Tree

```
procedure RangeQuery(seg, lazy, index, l, r, a, b, value)
    if $b < l$ or $a > r$ then
        return 0
    else if $a \leq l$ and $r \leq b$ then
        return $seg[index]$
    else
        $size = r - l + 1$
        $seg[index * 2] \leftarrow seg[index * 2] + value * size$          ▷ Update left child
        $lazy[index * 2] \leftarrow lazy[index * 2] + value$          ▷ Push lazy to left child
        $seg[index * 2 + 1] \leftarrow seg[index * 2 + 1] + value * size$ ▷ Same to right
        $lazy[index * 2 + 1] \leftarrow lazy[index * 2 + 1] + value$
        $lazy[index] \leftarrow 0$                              ▷ Reset lazy value for current node
        $result = 0$
        $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$
        $result += $ RangeQuery(seg, $2 \cdot index$, $l$, $mid$, $a$, $b$, $value$)
        $result += $ RangeQuery(seg, $2 \cdot index + 1$, $mid + 1$, $r$, $a$, $b$, $value$)
        $seg[index] \leftarrow seg[index * 2] + seg[index * 2 + 1] + lazy[index] * size$
        return $result$
    end if
end procedure
```

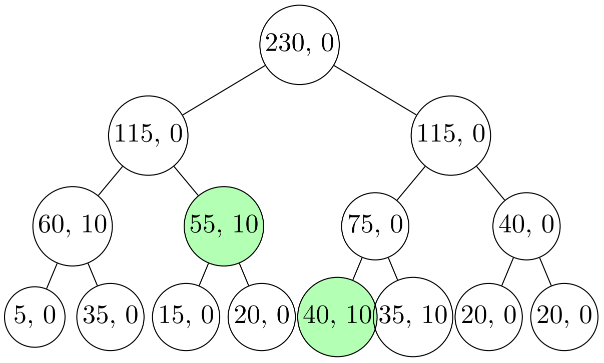

Figure 5: This is state of the tree after range query on range [2: 4]. Green nodes were we took values form to the sum. Result of query is 95.

2 A mathematical approach to segment trees

In this section we will try to understand how segment trees work and why they are so efficient. 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, $(\Sigma^*, \cdot, \epsilon)$**
  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 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 $a, b \in S$, $f(a * b) = f(a) \cdot f(b)$.
- **Preserves the identity element**
  $f(e) = \theta$.

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:

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

2.3 Endomorphism

A special case of homomorphism is an endomorphism. This is a homomorphism from a monoid to itself. Lets consider a simple example of endomorphism. Let $S = (\mathbb{N}, +, 0)$ and $f : S \to S$ be defined as $f(x) = x + 1$. We can see that:

- $f(a + b) = a + b + 1 = (a + 1) + (b + 1) = f(a) + f(b)$
- $f(0) = 0 + 1 = 1$

Interesing property of endomorphisms is that a Set of all endomorphisms of a monoid is also a monoid. Let's consider a set of all endomorphisms of a monoid S. Let $S = (\mathbb{N}, +, 0)$ and let E be a set of all endomorphisms of S. We can define a binary operation on E as follows:

$$f * g = h \text{ where } h(x) = f(g(x)) \tag{2}$$

We can see that this operation is associative and has an identity element $e(x) = x$.