Push-Free Segment Tree

Nikita Gaevoy

April 16, 2024

Figure 1: Tiffany

1 Prerequisites

First of all, who this article is aimed at. This article assumes you already know what a segment tree is. Have you never heard about the segment tree, the Fenwick tree (aka binary indexed tree, BIT) or the RMQ problem, you should better read about it somewhere else and then come back here right after to learn a bunch of cool stuff. The model target audience is the people able to solve this problem at least in theory. But even if you can't, this article could still be helpful. However, I think this article contains some ideas that were never published before, thus making it interesting even for people who are closely familiar with different variants of the segment tree. And if you are somewhere in the middle, like you know about segment trees, but find stuff like historical sums too complicated, this article is still for you, because it will make those super-easy to understand. Also, there will be a link to the code at the end, which you may take and use.

On a side note, if you are a huge fan of Fenwick tree or using some trees for solving RMQ, the vast majority of this article will still be applicable to your favorite data structure;

I just decided to focus on the data structure I find the simplest and, probably, the most feature-rich.

So, let's start!

2 What Segment Trees Can Actually Do

Here we discuss some algebra behind the operations on a segment tree. But don't be afraid! There's not any hardcore algebra here, we just give everything a name, so that it is easier for us to move on. So, what can segment trees actually do? A bunch of stuff! We have a lot of variants: computing range-sum, range-max, adding in a range, remaxing in a range, etc. We can also mix them as well. In order to make things a little more structured, I call an operation that computes something on a range without modification an addition and range-modification a multiplication.

A tropical example. Let our addition be the operation of maximum and multiplication be the ordinary addition. And let's consider an array $[1, 2, 3]$. After multiplication on a range 1..3 (0-indexed, semi-open, as always) by 4 and summing the whole array we will be computing $1 + 2 \cdot 4 + 3 \cdot 4$. Or, in ordinary notation it would be $\max\{1, 2 + 4, 3 + 4\}$.

Yes, I also assume that we always multiply from the right-hand side, because most of the languages have an operator $*=$, but don't have $=*$. Here algebra-fans could argue that multiplying from the left-hand side would have been more canonical, but as an average theoretical computer science enjoyer, I do not care.

What do we need from our operations of addition and multiplication in order for our segment tree to work? First of all, we clearly need associativity of both addition and multiplication just to be able to chain them. Second, a segment tree has two specific functions that we also need to work, one for updating a value in a node from the values of its children (we call it `update`, sometimes it is also called relaxing), and the second one is pushing the update from a node to its children (we call it `push`).

The `update` works always, there is nothing specific attached to it, but `push` requires us to be able to transfer the "modification" to the children, i. e. the following formula holds $(x + y) \cdot a = x \cdot a + y \cdot a$. This is the distributive property! If we examine the segment tree a little more closely, we will also notice that we require having identity elements for both addition (i. e. zero) and multiplication (i. e. one). To sum up, we need the following properties.

1. Associativity of addition: $x + (y + z) = (x + y) + z$.
2. Associativity of multiplication: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
3. Distributivity of multiplication over addition: $(x + y) \cdot a = x \cdot a + y \cdot a$.
4. Having zero: $x + 0 = 0 + x = x$. Sometimes also $0 \cdot a = 0$, depending on the implementation.
5. Having one: $x \cdot 1 = x$.

This is actually almost a semiring, we only miss the requirement of the addition to be commutative. Unfortunately, there is no algebraic name for such a structure, so I will stick to almost-a-semiring.

An important note: In fact, multipliers and summands don't have to belong to the same set or, from the programming point of view, type; moreover, implementation-wise it makes much more sense to have two different template types and, in fact, I use two different sets of

symbols for summands $(x, y, z, \ldots)$ and multipliers $(a, b, c, \ldots)$, but for the sake of simplicity, I will keep pretending this opportunity does not exist.

At this moment, it may look like we did a lot of boring algebraic work to gain nothing. At least, it seems like a segment tree can do all of that and even much more beyond. But, in fact, it does not, and almost all operations a segment tree can perform fit in this exact framework.

Examples:

- Let's start with the simplest one: $\mathbb{Z}$. Clearly, works.
- max and addition, tropical semiring. Again, fulfills all the requirements, zero and one are infinity and zero, respectively.
- Range-sum and range-addition. At first glance does not seem to fulfil the distributivity, while definitely being viable for a segment tree. However, it would work if we add the size of a subsegment to the value in the node and use the following rules:

$$\langle x, y\rangle + \langle z, t\rangle = \langle x + z, y + t\rangle$$
$$\langle x, y\rangle \cdot a = \langle x + a \cdot y, y\rangle.$$

  Then all our leaves are of the form $\langle x, 1\rangle$. Obviously, we don't have to store the size explicitly, but it is important to notice that it still participates in resulting formulas.
- Range-sum and range-assign. Here we have to add 1 manually, and multiplication follows the rule $a \cdot b = b$, except for 1, where $1 \cdot a = a \cdot 1 = a$. Again, the same trick with the size.
- Matrices, linear functions, polynomials, nimbers. Nothing tricky with them, but not excessive to mention as well.
- Historical sums! If you are unfamiliar with this idea, it is a trick over segment trees that allows you to answer the following types of queries on two "arrays" A and B:

  1. Add on a range of the array A.
  2. Perform element-wise addition $B \mathrel{+}= A$.
  3. Compute range-sums on B.

  And actually, those are much easier at this point, because for them, you are just adding piecewise linear functions representing how values in the array B change over time with the last segment (that represents the future) having the slope of A. The only important thing is to notice that you can store only the last segment of each function, because the time always moves forward. And that is it! Look how much simpler than tutorials on historical sums that are full of formulas over multiple segment trees!
- Almost any other variant of the segment tree named after some red Chinese participant. The only variant that stands out a bit is the segment tree beats, because unlike other variants of the segment tree its running time is amortized. Maybe a name like "amortized segment tree" would have been more sound than the current one. At least, it would reflect the main difference and be less animeish effectively attracting more problemsetters to the topic.

3 Multiple Dimensions

We have developed a language in the previous section, and now we can move on to discuss new things. You may say that the multidimensional segment tree is not anything new: you just store a segment tree inside a segment tree, do the same operations as in one-dimensional case, and you are fine. And I almost agree, but what do you do when you want to add on a rectangle and then also compute a sum in a rectangle? Well, you just store a segment tree inside a segment tree and then do the same operations as in one-dimensional case, you just need to check that `push` works... Oh, wait! `push`! You can't push the whole segment tree!

And it indeed looks like a problem. Moreover, there were a lot of discussions for similar multidimensional cases. First, there was an article by Laakeri who pointed out that in the most general case of arbitrary dimension, such data structure cannot exist unless ETH falls. Then, there was the problem D in Yuhao Du Contest 7 (all links could be found here) claiming that an offline variant of the two-dimensional case is possible. And now, I am going to present another trick to solve some other variants of this problem. What is even more important, this trick would eventually help us to cut in half both our time and memory usage in the one-dimensional case under some not so restricting conditions.

So, the trick. Let's focus exactly on the problem of an addition on a rectangle and computing the sum on a rectangle. Yes, I know that we can do prefix-sums and probably be fine without range modifications, but that is not the point. Our problem is that we can't push. And we can't make our inner segment trees be pushable, thus, we have to invent a segment tree without pushes. And it is possible!

Actually, I was definitely not the first one to come up with a similar data structure, even though I never saw any information on it posted publicly. But at least, Alex_2oo8 told me that he sometimes implements a similar data structure in some simple cases when I told him about it. Here I will describe the data structure in its most general form and with a couple of important optimizations that, to my knowledge, were never known before.

Let's firstly focus on a one-dimensional case where we want to make range-additions and compute range-sums. Similarly to the general case, we can store two values in a node: the sum on a subrange corresponding to that node and the value of our "laziness" that is what we have to add (in a sense of multiplication) to all our descendants. The difference is that we would never have to push our second value, "laziness". Instead of that, we will apply it each time we exit the node.

Here roughly how we computed range-sum before (in the code, `T` is the type in which we do all the calculation):

```
T range_sum(size_t l, size_t r,
            size_t cl, size_t cr, size_t v)
{
    if (r <= cl || cr <= l)
        return 0;
    push(v);
    if (l <= cl && cr <= r)
        return value[v];

    auto ct = midpoint(cl, cr);

    auto ans = range_sum(l, r, cl, ct, 2 * v) +
               range_sum(l, r, ct, cr, 2 * v + 1);

    update(v);
```

```
16
17      return ans;
18  }
```

And here roughly how we do it now:

```
1   T range_sum(size_t l, size_t r,
2               size_t cl, size_t cr, size_t v) const
3   {
4       if (r <= cl || cr <= l)
5           return 0;
6       if (l <= cl && cr <= r)
7           return value[v] + laziness[v] * (cr - cl);
8
9       auto ct = midpoint(cl, cr);
10
11      auto ans = range_sum(l, r, cl, ct, 2 * v) +
12                 range_sum(l, r, ct, cr, 2 * v + 1);
13
14      return ans + laziness[v] * (min(cr, r) - max(cl, l));
15  }
```

And in order to solve the problem with rectangular addition and rectangular sum, the only thing we need to change is to additionally perform queries to the inner tree in computing the sum:

```
1   T rectangle_sum(size_t l, size_t r,
2                   size_t bot, size_t top,
3                   size_t cl, size_t cr, size_t v)
4   {
5       if (r <= cl || cr <= l)
6           return 0;
7       if (l <= cl && cr <= r)
8           return value[v].range_sum(bot, top) +
9                  laziness[v].range_sum(bot, top) * (cr - cl);
10
11      auto ct = midpoint(cl, cr);
12
13      auto ans = rectangle_sum(l, r, bot, top, cl, ct, 2 * v) +
14                 rectangle_sum(l, r, bot, top, ct, cr, 2 * v + 1);
15
16      return ans + laziness[v].range_sum(bot, top) * (min(cr, r) - max(cl, l));
17  }
```

The addition part does not change too much comparing to ordinary two-dimensional data structures, but updating is a little tricky:

```
1   T rectangle_add(size_t l, size_t r,
2                   size_t bot, size_t top,
3                   size_t cl, size_t cr, size_t v,
4                   const auto &x)
```

```
{
    if (r <= cl || cr <= l)
        return 0;
    if (l <= cl && cr <= r)
    {
        laziness[v].range_add(bot, top, x);

        return x * (cr - cl);
    }

    auto ct = midpoint(cl, cr);

    auto add = rectangle_add(l, r, bot, top, cl, ct, 2 * v, x) +
               rectangle_add(l, r, bot, top, ct, cr, 2 * v + 1, x);

    value[v].range_add(bot, top, add);

    return add;
}
```

But what do we actually need from the segment tree for this trick to work? Commutative property of multiplication only: $a \cdot b = b \cdot a$. However, in the multidimensional case, this property turns out to be quite restrictive (remember that in that case our elements are stripes, not the values, and see how it changed the updating part in `rectangle_add`) making it hard to come up with almost any other natural example except for rectangular addition and rectangular sum.

But in one-dimensional case, the push-free segment tree (this is how I named it about 3.5 years ago when it came up during some training contest) allows us to write the code without pushes giving the possibility to mark sum queries `const` and improving the const-safety of your code. But it does not stop here!

4 Push-Free Segment Tree Upwards

As you probably know, there are two main ways to implement a segment tree. Downwards: recursive, we start at the top and go towards the bottom. And upwards: a simple loop, we start at the bottom of the tree and go towards the top.

You probably also know that the upwards implementation is about two times faster than the downwards. On top of that, it also doesn't require extending the length of the underlying vector to be a power of 2. Note that in this case you have to be extra careful if your addition is not commutative, because, for example, the root can store the sum of the whole array in a wrong order. But if you don't cut any corners and just invoke range-sum on every sum query, it is completely fine.

The main problem of the upwards implementation is that it can't do pushes and therefore is unable to deal with both range sum and range multiplication queries at the same time. But our brand new push-free segment tree does not require them! Even though the idea is not really too smart, it took me more than three years to come up with it.

Still, the most effective implementation is not exactly easy, so I dedicate the rest of this section to the tricks that help make it easier and faster.

The first important moment is that when you multiply, there can be up to four nodes on each layer that are to be updated somehow (see Figure 2). Two could have been multiplied

<table>
<tr><td colspan="16">0..16</td></tr>
<tr><td colspan="8">0..8</td><td colspan="8">8..16</td></tr>
<tr><td colspan="4">0..4</td><td colspan="4">4..8</td><td colspan="4">8..12</td><td colspan="4">12..16</td></tr>
<tr><td colspan="2">0..2</td><td colspan="2">2..4</td><td colspan="2">4..6</td><td colspan="2">6..8</td><td colspan="2">8..10</td><td colspan="2">10..12</td><td colspan="2">12..14</td><td colspan="2">14..16</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
</table>

Figure 2: Example of multiplication on range 1..11. Cyan cells correspond to multiplied nodes and lime cells correspond to not directly multiplied but with an updated sum.

themselves, and on two (possibly, distinct) you probably have to run an `update`. The latter two (when exist) are placed right outside the range we are maintaining when we are running our loop for multiplication. In order not to accidentally access outside the memory (and not to do range-checks), it is wise to have the underlying vector of size at least $2n+2$ while having the root at the index 1 (which is actually the best index for the root in all implementations of the segment tree). The second thing we have to remember is that we always have to go the full path to the root, because there could be some nontrivial multipliers or nodes you still have to update. And the last optimization is to change the convention of storing our values a little. If we store not the pair of the sum of the children and the lazied operation to the subtree, but rather the sum with this operation already applied and the operation itself, we would save one additional multiplication per `update`, which gives us a couple of tens of percents of speedup, depending on the operations themselves. Always nice to have.

And overall, we built a data structure that for any commutative almost-a-semiring is able to do whatever the regular recursive segment tree with lazy propagation can do, but in time and space of the segment tree upwards, which is practically roughly two times better in both parameters! Also, if you think that commutativity is a very restrictive property, remember that sometimes you may slightly change your operations to fulfill it. For example, very non-commutative assignment can become commutative `remax`, if you add the time of assignment, which makes range-max and range-assignment a viable pair of operations for a push-free segment tree.

Here is the implementation I made with help of Burunduk1 for our team reference document for the World Finals that implements all the tricks above. The repository (here is the link to its root) contains a lot of cool stuff, so if you find something else in it useful, feel free to use it as well.

5 Off Topic

Having such an opportunity, I also want to promote our (made jointly with tranquility and Kaban-5) new contest that was named SPb SU LOUD Enough Contest 2 in Petrozavodsk Winter Camp 2023 and the Northern stage of the second Universal Cup. We think that its problems are full of great ideas and we are willing to share them with the community to commemorate our retirement from the ICPC career. And if you are reading it somewhere in the future, it is still probably a great contest to train on!

Acknowledgements

Many thanks to Boris, Liana, Vanya and Vlad who read this text prior to publication. Also, special thanks to Katya for the picture of Tiffany that should attract two times more readers to this text than the sacral knowledge it contains.