Segment Trees

Justin Zhang[*]

December 8, 2017

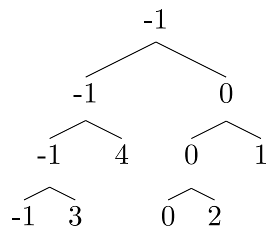
1 Introduction

A *segment tree* is a data structure for storing intervals, or *segments*. Segment trees can efficiently answer dynamic range queries. We will use a segment tree to solve the Range Minimum Query (RMQ) problem, which is the problem of finding the minimum element in an array within a given range i to j. Other range queries include range sum, minimum, GCD, or product. A naive solution to RMQ is to iterate from index i to j, which takes $O(n)$ per query. This is too slow if n is large or if there are many queries. Another solution is to build a 2D matrix containing every single RMQ, which would be able to answer queries in $O(1)$ time. However, it would take $O(n^2)$ time to build this matrix and $O(n^2)$ space to store the matrix. Therefore, neither of these solutions scales well. Segment trees solve the problems of both time and space.

A segment tree differs from a binary indexed tree (which we've talked about earlier in the year) in that it can solve a much larger range of range queries. The downside is that a segment tree takes more space and is harder to code.

2 Constructing the Tree

A segment tree is a balanced binary tree in which each leaf represents an element in the array. The root of the tree represents segment $[0, n-1]$, and for each segment$[l, r]$ represented by the node at index p, the left child represents the segment $[l, (l+r)/2]$ and the right child represents the segment $[(l+r)/2+1, r]$. In the case of RMQ, "represents" means the value of the node is the minimum of the segment it represents. For example, for the array $[-1, 3, 4, 0, 2, 1]$, the tree would look as follows:



Constructing this tree takes $O(n)$ time and $O(n)$ space. In the pseudocode below, we build the tree recursively. The tree is represented as an array *st* where index 1 is the root of the tree and the left and right children of index p are indices $2 \times p$ and $(2 \times p) + 1$, respectively. l and r are the left and right bounds of the current segment, respectively.

[*]Based on Kevin Geng and Charles Zhao's Segment Tree lectures

Algorithm 1 Segment Tree Construction

```
function BUILD($p$, $l$, $r$)
    if $l = r$ then
        $st[p] \leftarrow A[l]$
    else
        $pl \leftarrow 2 \times p$
        $pr \leftarrow 2 \times p + 1$
        BUILD($pl, l, (l + r)/2$)
        BUILD($pr, (l + r)/2, r$)
        return MIN($st[pl], st[pr]$)
```

Note, however, that we can construct this tree by updating length-one ranges n times. This gives us a runtime of $O(n \log n)$ for constuction, which is usually fast enough.

3 Solving Queries

There are three cases that we must consider when traversing a segment tree: when part of the segment represented by the node is within the query; when the segment is completely within the query; and when the segment is completely outside of the query. If part of the segment is within the query, we must check both of the node's children. If the segment is completely within the query, we return the node's value, which is the minimum of the segment it represents. If the segment is completely outside of the query, we return some very large number. In the pseudocode below, we traverse the tree recursively. With the segment tree built, solving an RMQ takes $O(\log n)$ time. This is because segment trees allow us to avoid traversing unrelated parts of the tree. In the worst case, in which only part of every segment we reach is within the query, we traverse two root-to-leaf paths, taking $O(2 \times \log n) = O(\log n)$ time.

Algorithm 2 Range Minimum Query Using a Segment Tree

```
function RMQ($p$, $l$, $r$, $i$, $j$)
    if $i > r$ or $j < l$ then
        return $\infty$
    if $l >= i$ and $r <= j$ then
        return $st[p]$
    $pl \leftarrow 2 \times p$
    $pr \leftarrow 2 \times p + 1$
    $minl \leftarrow$ RMQ($pl, l, (l + r)/2, i, j$)
    $minr \leftarrow$ RMQ($pr, (l + r)/2 + 1, r, i, j$)
    return MIN($minl, minr$)
```

4 Modifying the Tree

Remember that we said segment trees can efficiently answer *dynamic* range queries. This means that if the array on which we are performing RMQs changes, we can efficiently update the segment tree. If an element in the array changes, we start from the leaf node representing that element and move up the tree, updating nodes as we go. Thus, this takes $O(\log n)$ time.

5 Lazy propagation

With a segment tree, we can already handle the range min and sum queries with a complexity of $O(\log n)$ for both queries and *point updates*, or updates of individual elements. But what if we also wanted to perform *range updates*, or updates of a range of elements? If we want to increment the value of m contiguous elements, this will take $O(m \log n)$ time.

Fortunately, we can do better. With our current data structure, performing a range update will require modifying the values of m elements, so we need to change our data structure. To avoid recursing all the way to the bottom of the tree for all m elements, we can store some information in higher nodes instead.

Let's say we're performing a range update on the range u, and we're examining the segment s. The key insight is that if s is contained within u, then we can simply set a *lazy* value for s instead of recursing on its children. When a lazy value is set, it means that the values of **each of** the children of that node should be incremented by that value.

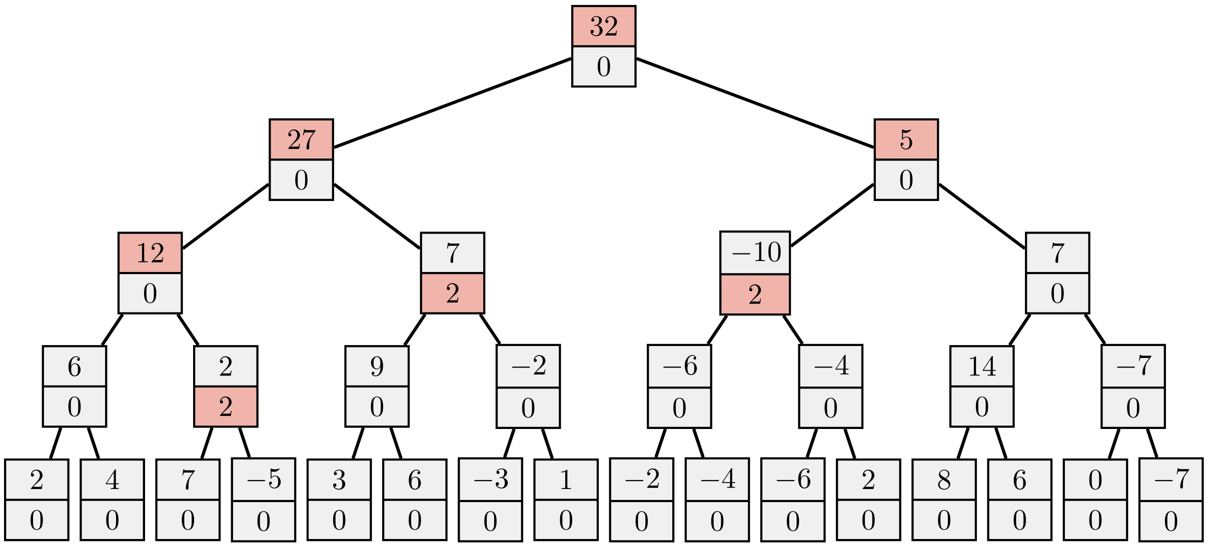


Figure 1: Example of a segment tree with lazy values stored underneath. The highlighted values reflect the values updated after calling *update*$(3, 12, 2)$. *Credit: Samuel Hsiang.*

However, if we encounter a lazy value while performing a query or an update, however, we need to push the value by moving the lazy value into its children, and updating the current node's sum correspondingly. But because we only push once per node, these operations still only take $O(\log n)$.

We can also use this technique to implement the range minimum query (RMQ) problem. A full implementation of both is provided by the solution to Counting Haybales (USACO December 2015, Platinum) at `http://www.usaco.org/current/data/sol_haybales_platinum_dec15.html`.

6 Problems

1. Counting Haybales (see link above)

2. You're given an array of length 10^5, consisting of 0's and 1's. Answer 10^5 operations, where each operation is either to invert the bits from a given i to j or to count the number of set bits from i to j.

3. You're given an array t of length 2^{17}, with non-negative integers less than 2^{30}. Let's define an OR(a) to be the result after bitwise-OR'ing pairs of elements in an array a (OR(a) := a[1] OR a[2], a[3] OR a[4], ...) and XOR(a) to be the result after bitwise-XOR'ing pairs of elements in an array a (XOR(a) := a[1] XOR a[2], a[3] XOR a[4], ...). V(a) is then defined to be the final value after alternating the OR and XOR operations (beginning with OR) on an array a until there is one value left: OR(XOR(OR(XOR(...OR(a)...)))). Answer 10^5 operations, where each operation sets the value of t at index i to a new value and then calculates V(t) (Codeforces 339D).

4. You're given an array of length 10^5. Answer 5×10^4 queries, where each query either asks to print the sum of all elements from a given index l to a given index r or to apply a bitwise-XOR to all elements between l and r with a value x (Codeforces 242E).