Segment Trees

Rares Cristian

December 5, 2017

1 Range Queries

Let's first look at some example problem. Let's consider the following: given some array of numbers, and several queries consisting of continuous ranges of the array, determine the sum of the elements in each range.

For example, take the array:

3 6 2 9 8 3 2 10

And a possible query could ask for the sum of elements from indices 3 to 7 (using 1-based indexing). In this case, this would be $2 + 9 + 8 + 3 + 2$.

1.1 Prefix Sums

Say that the elements of the array are $a_1, a_2, ..., a_n$ for some integer n. Let's create an array $prefix[]$ defined as $prefix[i] = \sum_{i=1}^{i} s_i$. Additinally, we can compute this array iteratively as follows:

$$prefix[0] = 0$$

$$prefix[i] = prefix[i-1] + a_i, i \geq 1$$

We can answer a query asking for the sum of elements between indices j and k as follows:

$$\sum_{i=j}^{k} a_i = prefix[k] - prefix[j-1]$$

Thus, we can answer each query in $O(1)$ time with $O(n)$ preprocessing to create the $prefix$ array. Here's a nice problem utilizing this on SPOJ.

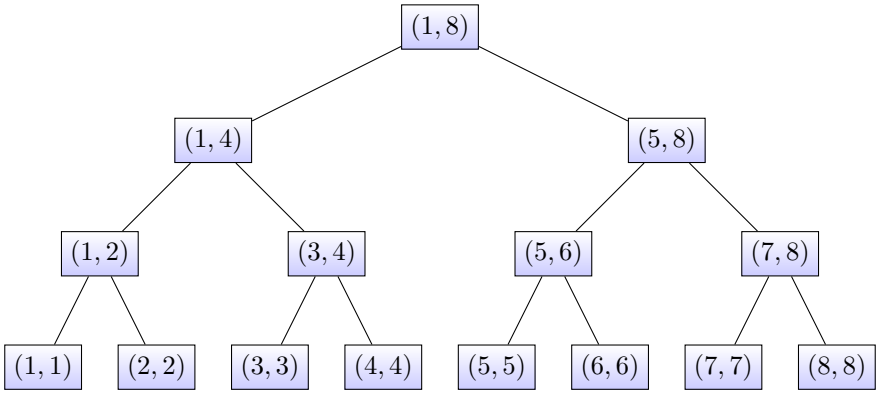
1.2 Segment Trees

Now, let's say we have another type of operation that can be interleaved with the interval sum queries. We also want to change an element at some given index to some new value. Formally, we now have an update query (i, v) where we change the value of a_i to v.

For the moment, let's consider solving this again using prefix sums. Answering queries would be the same as before, however, when updating, we need

to change all indices from i onwards in our $prefix$ array. Thus, we have $O(1)$ queries and $O(n)$ updates.

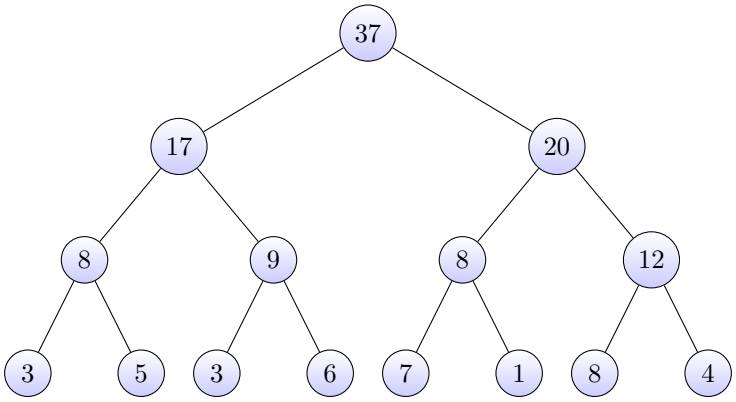
We need a different approach. This is where segment trees come in. Each node in the tree will represent some range (i, j) in our array and will store the sum of the elements in that range. The root node of the tree will hold the sum of all elements in the array. If a node represents the range (i, j). The left child of that node will hold the sum of the elements of the left half of the parent's interval, that is the range $(i, \lfloor \frac{i+j}{2} \rfloor)$. The right child will hold the right half $(\lfloor \frac{i+j}{2} \rfloor + 1, j)$. For example, say we have an array of 8 elements. Here is the ranges each node in the segment tree will represent



If we have the array,

3 5 3 6 7 1 8 4

Our segment tree would be:



Note that the leaves will all represent intervals of length 1, and thus the elements of the array itself. Let's take a look at some code to build this tree. Let's denote our array to be A and we will represent our tree with an array. The root appears at index 1. Since this is a binary tree, the left child of i will be at index $2 \cdot i$ and the right child is $2 \cdot i + 1$.

```
void build(int node, int left, int right) {
    if (left == right) {
        tree[node] = A[left];
        return;
    }

    int mid = (left + right) / 2;
    build(node * 2, left, mid);
    build(node * 2 + 1, mid + 1, right);
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
```

Let's take a moment to see how feasible it is to construct such a tree. First, the height of the tree is on the order of $log_2(n)$. The number of nodes at depth $i + 1$ is twice the number of nodes at depth i since we have a binary tree. So, the number of nodes at depth i in the tree is 2^i. We have n leaves, so $2^{height} = n$ and so $height = log_2(n)$.

Since there are 2^i nodes at depth i and the height of the tree is $log_2(n)$, it follows that there are

$$\sum_{i=0}^{log_2(n)} 2^i =$$

$$2^{log_2(n)+1} - 1 =$$

$$2n - 1$$

nodes in the tree. [1] These figures we have discussed are simply those of a binary tree.

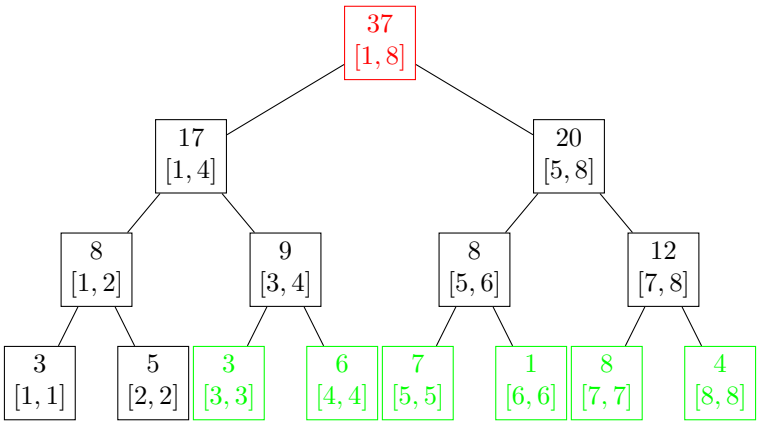
Now, we will first discuss how this tree can be used to solve our range queries and then how to handle updates.

Say that we want the answer for the range (a, b). We will construct the interval (a, b) from several segments that are represented within our segment tree, for which we already have computed the sum. Call S the smallest set of ranges in our segment tree whose union creates (a, b) exactly. When traversing the tree, we always start at the root. Say that we are currently at node i which

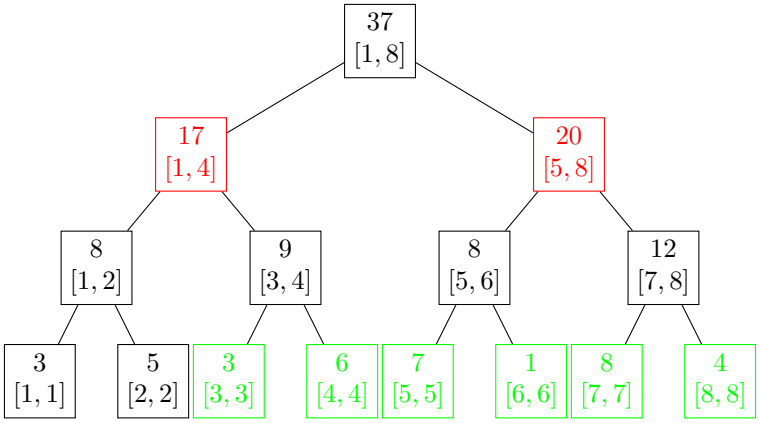
[1] Note that this is exact only if n is a power of two. Otherwise, this figure serves as an upper bound.

represents the range (l, r). If the interval (l, r) is completely contained within (a, b), then $(l, r) \in S$. If the range (l, r) does not intersect (a, b) at all, then there is no point in traversing further down the tree. Otherwise, we have that (l, r) intersects (a, b), so we must traverse down to both of its children.

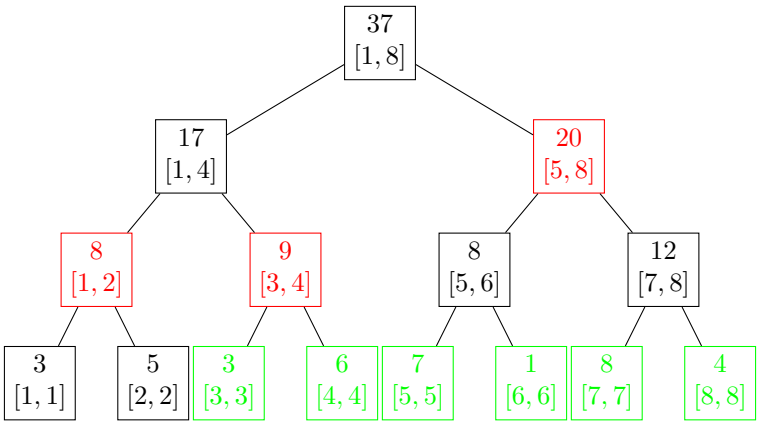
Let's go over an example using the array for the above-drawn segment tree. Say we want to find the sum of elements at indices from 3 to 8. We will use red to show the nodes we are currently traversed to, and green to be our query. Blue will represent our solution, the nodes in S.



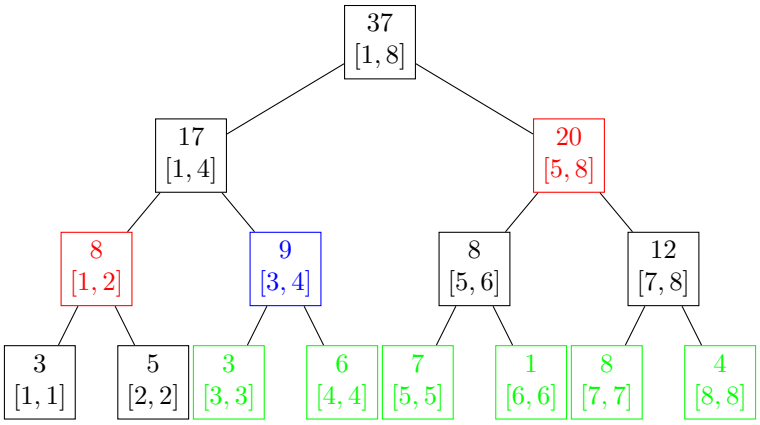
We start at the root, so the current interval is $(1, 8)$. This intersects $(3, 8)$, so we travel down to both children. We do this recursively, so here we solve the problem in the range $(1, 4)$ first, then in the range $(5, 8)$.



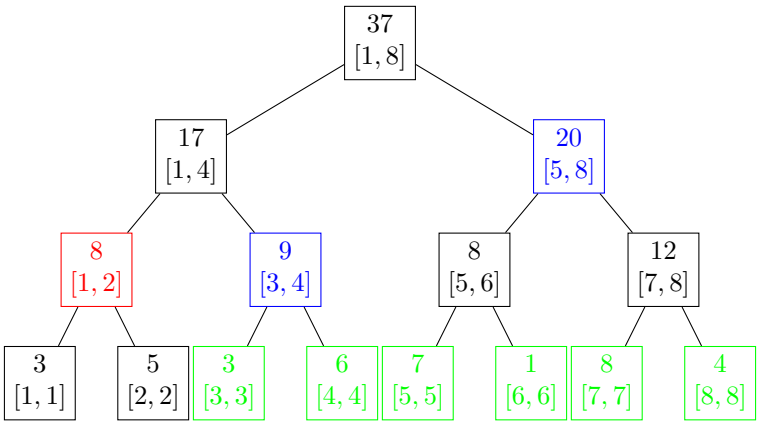
Again, we have that $(1, 4)$ intersects our query, so we must traverse to both children: $(1, 2)$ and $(3, 4)$.



Now, we see that the left child, $(1, 2)$ is completely outside of our query range, so we can stop there altogether, and we will not include the sum in our answer. We now recursively backtrack to the right range, $(2, 4)$. This is completely contained within the query, so we include it in our answer.



Now, we recursively backtrack to the range $(5, 8)$. Here we see that this is also completely within our query range, so we include it in our answer. We do not traverse the tree downwards from here, and there are no more nodes to visit in our recursion. Thus, we are done.



Our answer is simply what appears in the blue nodes. So we have the sum is $20 + 9 = 29$ in the range of elements from 3 to 8.

Here is some $c++$ code that answers queries:

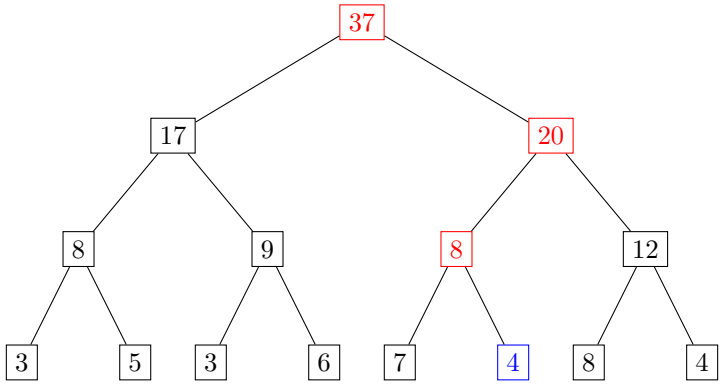
```
int query(int node, int l, int r, int a, int b) {
    // current interval is completely outside of query range
    if (b < l || a > r)
        return 0;

    // completely within range
    if (a <= l && b >= r)
        return tree[node];

    //partially in range, partially out (intersection)
    int mid = (start + end) / 2;
    int left_child = query(node * 2, l, mid, a, b, default_val);
    int right_child = query(node * 2 + 1, mid + 1, r, a, b, default_val);
    return right_child + left_child;
}
```

Let's analyze the time complexity of answering queries before moving forward. This clearly is not $O(1)$ as it was for the prefix sums, however, each query can be answered in $O(log(n))$ time using a segment tree. We can prove this by showing that no more than two nodes will be expanded at each level. We say that to expand a node is to recurse down to its children. If this is true, we have a constant number of operations at each level. Since there are $log(n)$ levels, we indeed have complexity $O(log(n))$. We prove that at most two nodes are expanded at each level by contradiction:

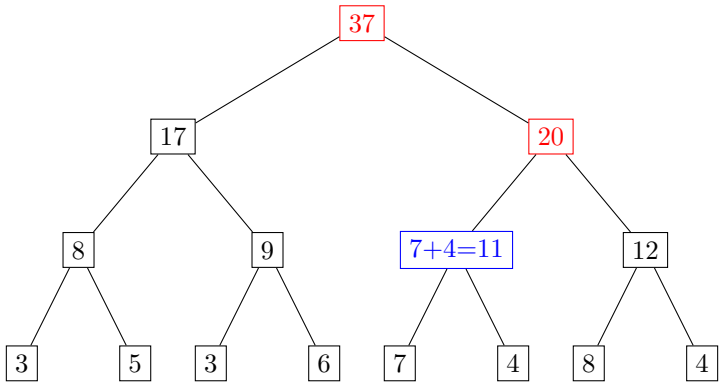
Assume that there are three nodes being expanded. Our query range must extend all the way from the leftmost expanded node, to the rightmost one.

This, however, indicates that the middle node is completely within the query range. When this happens, the algorithm stops searching the children and simply returns the sum. Thus the middle node would never be expanded. So, at most two nodes are expanded.

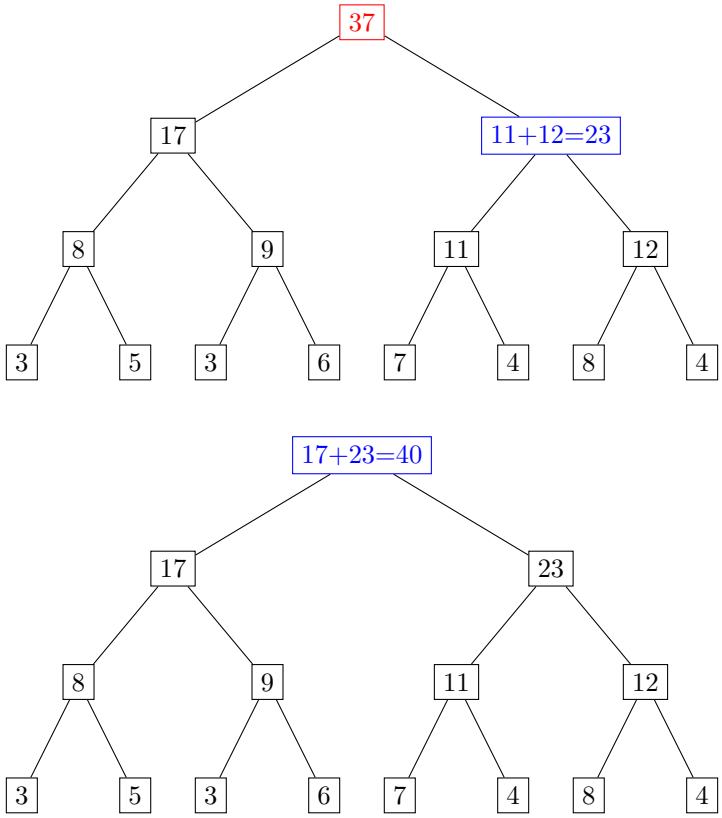
Now comes the question of updating values. This is done in a similar traversal fashion as answering queries. We need to find the path leading from the root down to the leaf node that represents the element we want to update. As we backtrack to the root, we also update the values of the nodes in the tree. These nodes on the path from root to leaf are the only ones that will be affected by changing a single value. Note that updating the nodes on the path is really the same as when we're building the tree in the first place (except we limit ourselves to a single path). Going back to our example, let's say that we want to update the value of the 6^{th} element from a 1 to a 4. Below is the path from root to leaf highlighted in red, with the blue node as the current vertex we are updating.



Note that we already updated the leaf node. Now we backtrack. The sum in the current node is simply the sum of its left and right children.



Performing the next iterations up to the root we have:



Essentially, all we have done is add 3, which is the difference between our original value (1) and the updated value (4), to each node along the path from the root to the leaf corresponding to the updated value.

The time complexity is much easier to see than for queries. We traverse along a path from the root to a leaf node, and since the height is $O(log(n))$ then so is our time complexity.

With that, we have a data structure that can answer queries that can find the sum of elements in a contiguous range, as well as update a single element to another value. These both are done in $O(log(n))$ time, which is an improvement over the prefix sums solution if we have to support a lot of updates.

Here is some code that performs updates.

```
// currently inspecting range (l, r). Want to update element at index
    'index' with value 'val'
void update(int node, int l, int r, int index, int val) {
    if (l == r) {
        tree[node] = val;
        return;
    }

    int mid = (l + r) / 2;
    if (index <= mid)
        update(node * 2, l, mid, index, val);
    else
        update(node * 2 + 1, mid + 1, r, index, val);
}
```

What we have now is pretty nice. But what if we wanted something other than the sum over an interval? What if we wanted to find the maximum element in some range? This very clearly cannot be solved with prefix sums in any way, but our segment tree only needs a small tweak.