Augmented Static BBST (Segment Tree)

Samuel Hsiang

July 2015

Suppose we want to support two operations:

- *update*(i, x) – increment the value of a_i by x
- *query*(i, j) – return $\sum_{k=i}^{j} a_k$.

Suppose we simply stored the sequence in an array. *update* then becomes an $O(1)$ operation, but *query* is $O(n)$.

2	4	7	-5	3	6	-3	1	-2	-4	-6	2	8	6	0	-7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Another natural approach would be to store in a separate array the sum of the first i terms in the sequence for every index i, or store the *prefix sums*.

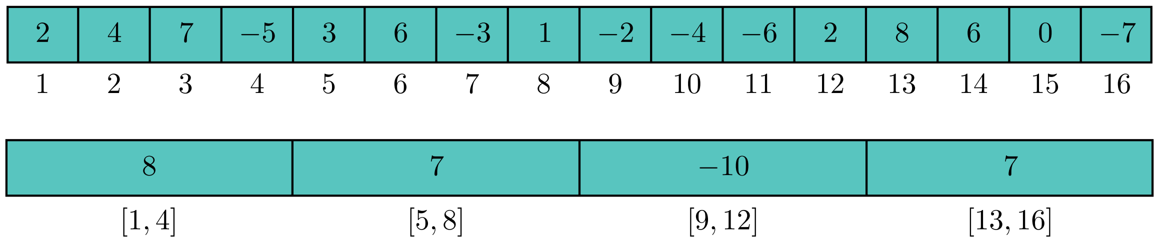
0	2	6	13	8	11	17	14	15	13	9	3	5	13	19	19	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Now *query* becomes an $O(1)$ operation, as we can simply subtract two elements in the array to answer a query. Unfortunately, *update* becomes $O(n)$, as changing the value of an element in the beginning of the sequence forces us to change almost all the values in the prefix sum array.

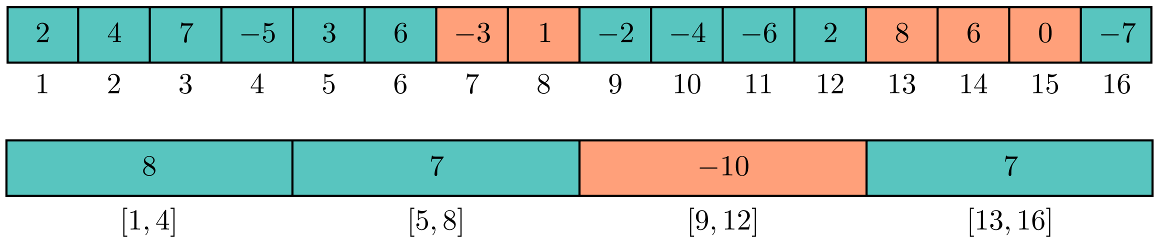
1 Some Motivation: $\sqrt{n}$ Bucketing

We can still use this idea, though... what we are looking for is some way to group values into sums such that we only need to change a small number of the sums to *update* and only require a small number of them to *query*.

This leads us directly to a $\sqrt{n}$ bucketing solution. Let's group the 16 elements into 4 groups.



We'll keep track of the total sum of each group. Now, if we want to update a value, we need to change only two values – the value of that element in the original array and the total sum of the bucket it is in. When we query a range, we'll take advantage of the sum of the bucket when we can. Highlighted are the numbers we'll need for $query(7, 15)$.



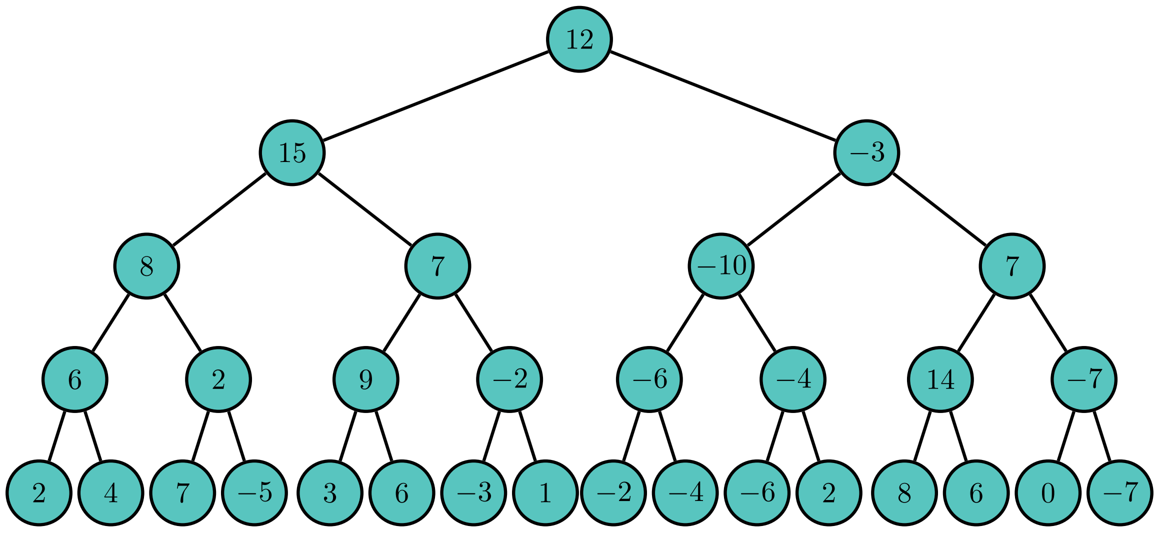
Querying requires access to at most $\sqrt{n}$ bucket sums and $2(\sqrt{n}-1)$ individual values. Therefore we have $O(\sqrt{n})$ query and $O(1)$ update. We are able to improve $O(\sqrt{n})$ update to $O(1)$ because of nice properties of the $+$ operator. This is not always the case for range queries: suppose, for instance, we needed to find the minimum element on a range.

It is often the case that $O(\sqrt{n})$ bounds can be improved to $O(\log n)$ using more complex data structures like segment trees and more complex ideas like 2^n jump pointers, both of which are covered in this chapter. These are, however, more complicated to implement and as such are often comparable in runtime in the contest environment. Steven Hao is notorious for using crude $\sqrt{n}$ bucketing algorithms to solve problems that should have required tighter algorithm complexities. $\sqrt{n}$ bucketing is a crude yet powerful idea; always keep it in the back of your mind.

2 Segment Tree

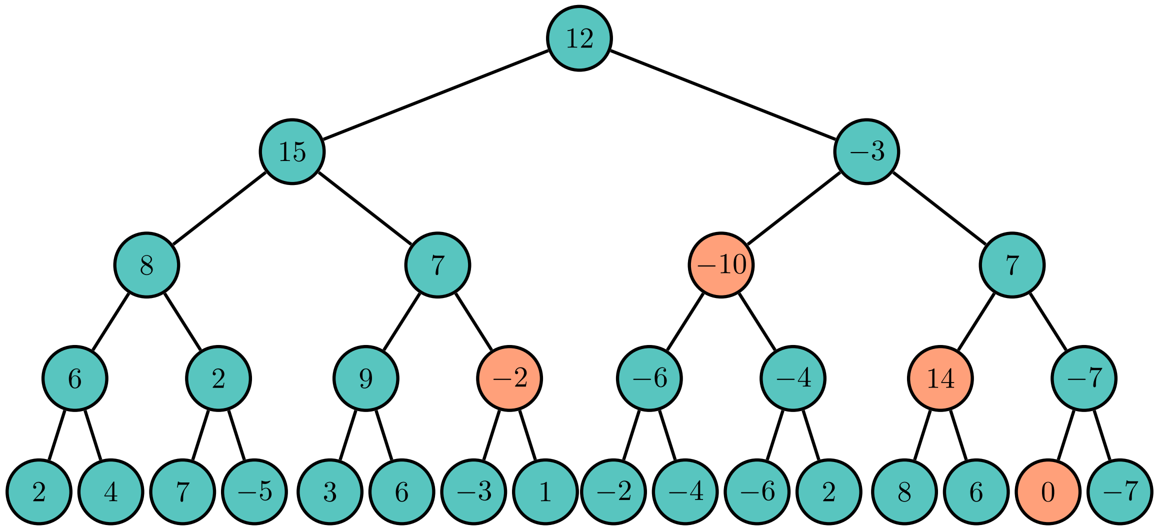
It turns out for our specific sum problem that we can do as good as $O(\log n)$ with a *segment tree*, or *range tree*, or *augmented static BBST*. The essential idea is still the same – we want to group elements in some way that allows us to update and query efficiently.

As the name "tree" suggests, we draw inspiration from a binary structure. Let's build a tree on top of the array, where each node keeps track of the sum of the numbers associated with its children.



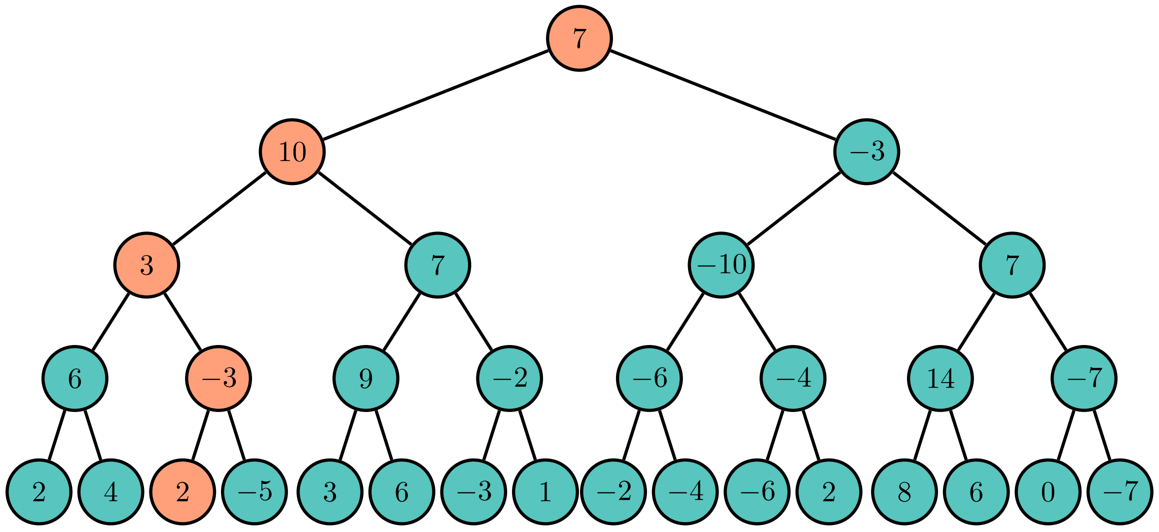
2	4	7	−5	3	6	−3	1	−2	−4	−6	2	8	6	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Highlighted are the nodes we'll need to access for $query(7, 15)$. Notice how the subtrees associated with each of these nodes neatly covers the entire range $[7, 15]$.



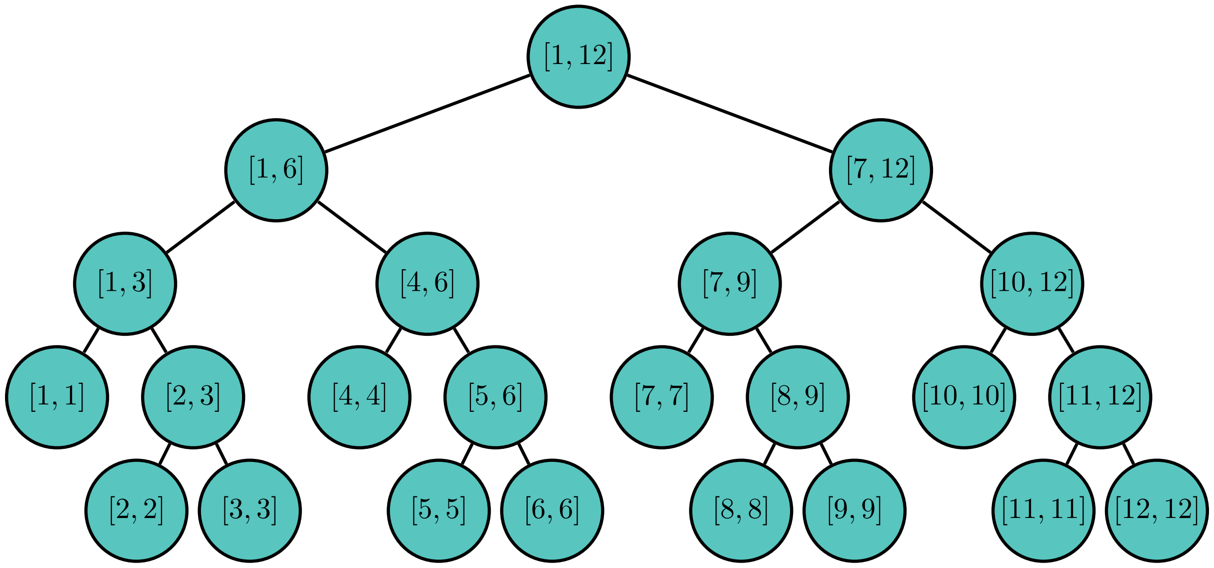
Again, the key idea is if the the interval associated with a node is completely contained within the interval we are querying, we simply return the sum associated with that node. Otherwise, we recurse on the two children. This process is $O(\log n)$ because each level in the tree can have at most two highlighted nodes.

If we wanted to change the third element to 2, we would have to update the highlighted nodes in the following diagram. This process is more straightforward than the querying.



Updating is also $O(\log n)$ as we need to change the values of at most one node in each level in the tree.

I cheated with my example by using a nice power of two, $n = 16$, as the number of elements in the sequence. Of course, the size is not always this nice. One solution is to simply pad the back of the sequence with enough 0s and pretend that the number of elements in the sequence is actually a power of two. For the segment tree, this is not necessary – if a vertex is associated with the range $[a, b]$, we can simply split this range into two, $\left[a, \left\lfloor \frac{a+b-1}{2} \right\rfloor\right]$ and $\left[\left\lfloor \frac{a+b-1}{2} \right\rfloor + 1, b\right]$. We then recursively apply this pattern except for when the lower bound of the range is equal to the upper bound. If $n = 12$, the resulting tree would have the following structure.



For this reason, while I used the concept of "building up" on top of our array to introduce the segment tree, any operation we implement will start at the root and recursively trickle down the tree. We see that the segment tree structure does not have to resemble a complete tree at all.

However, with this approach, it is still quite balanced, so we can store a segment tree within an array as we would a heap.

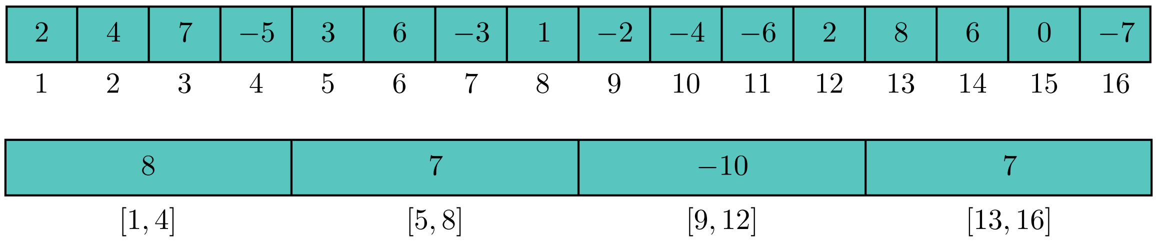
3 Lazy Propagation

It is often the case that in addition to performing range queries, we need to be able to perform *range updates*. (Before, we only had to implement point updates.) One extension of our sum problem would require the following two functions:

- $update(i, j, x)$ – increment the value of a_k by x for all $k \in [i, j]$
- $query(i, j)$ – return $\sum_{k=i}^{j} a_k$.

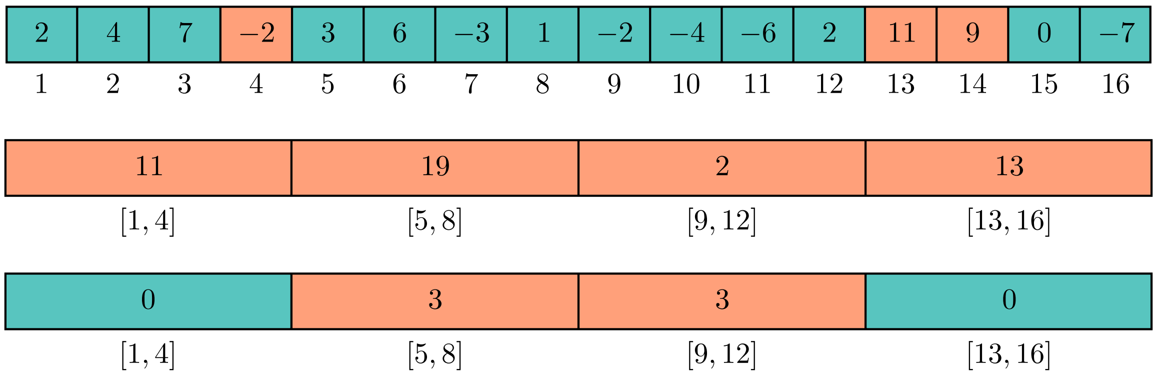
3.1 Some Motivation: $\sqrt{n}$ Blocking

Let's go back to our $\sqrt{n}$ blocking solution and see what changes we can make, and hopefully we can extend this idea back to our segment tree. If we're looking for an $O(\sqrt{n})$ implementation for *update*, we clearly can't perform point updates for all values in the range. The way we sped up *query* was by keeping track of an extra set of data, the sum of all the elements in a bucket, which we used when *the entire bucket was in the query range.*



What can we do with this idea for *update*? Let's see what we can do if an entire bucket were included in the update range. Again, we don't want to touch the original array a at all since that makes the operation linear. Let's try storing some information separately. This other information we're storing is the key idea behind lazy propagation.

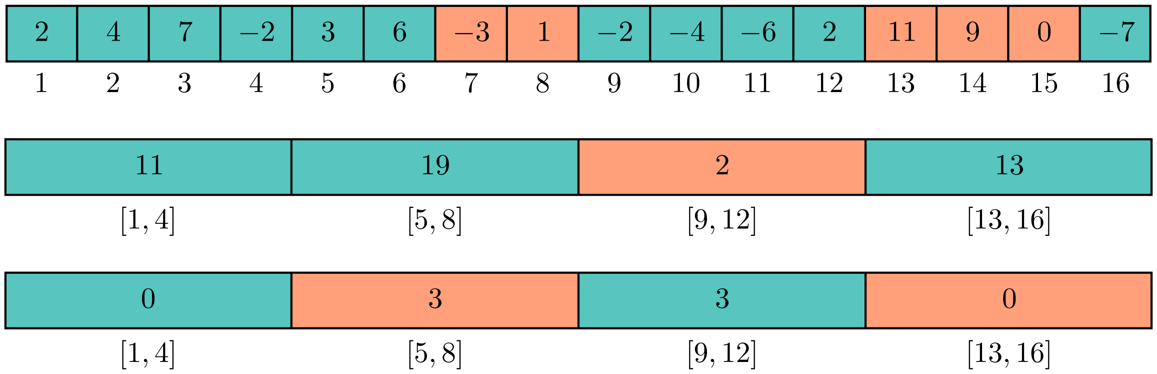
With this in mind, highlighted are the elements we'll need for $update(4, 14, 3)$.



Note that the sums associated with each bucket must adjust appropriately. In the example, there are four elements per bucket, so when an entire bucket needs to be incremented by 3, a single bucket can increment its sum by $4 \cdot 3 = 12$ easily.

To reiterate, we not storing the actual values of the elements where they were stored in solving the original formulation of the problem. Despite this fact, we are able to calculate what any single value is supposed to be. a_i is simply equal to the $\left\lceil \frac{i}{\sqrt{n}} \right\rceil$th value stored in the newest third array added to the ith value stored in the first array. Because of this, querying a range works in almost exactly the same way as it did in the original formulation.

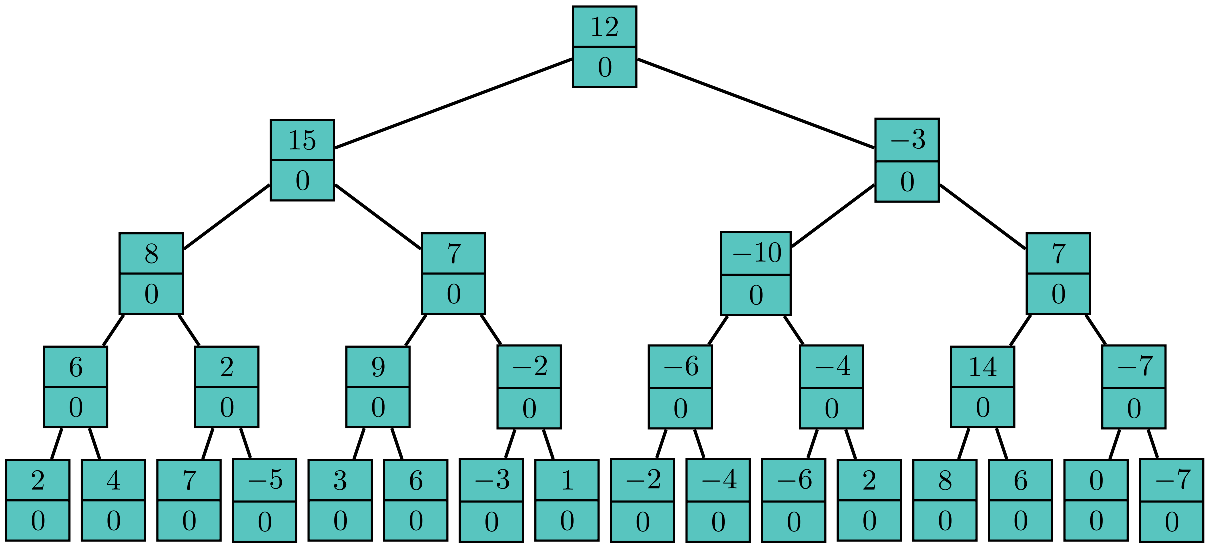
Highlighted are the values necessary for $query(7, 15)$.



Thus we have achieved $O(\sqrt{n})$ for both range updates and and range queries.

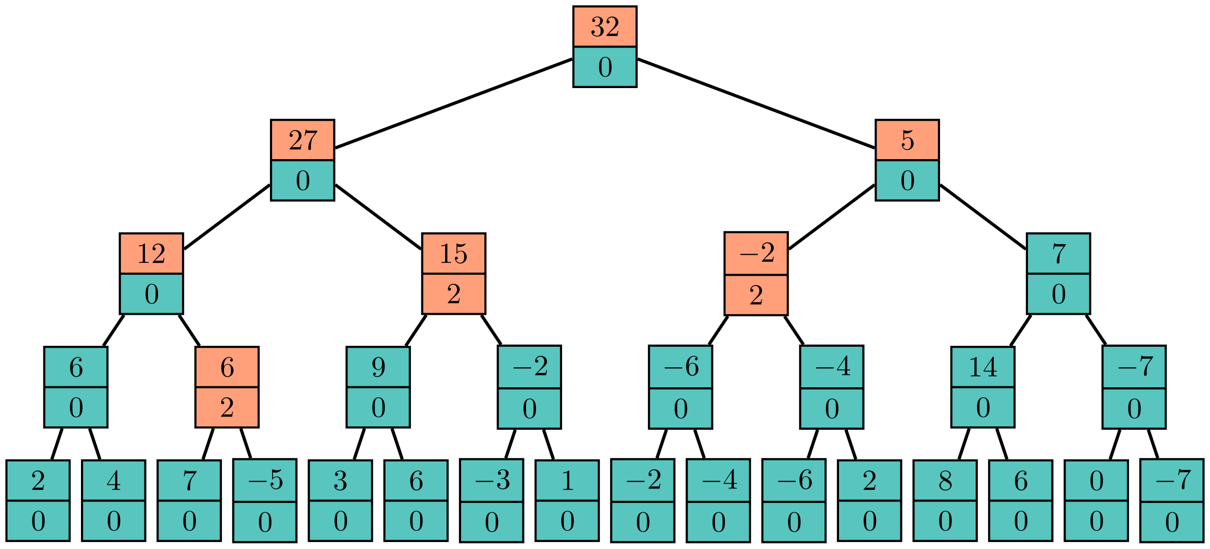
3.2 Lazy Propagation on a Segment Tree

Motivated by how we fixed our $O(\sqrt{n})$ solution, let's try adding a similar extra piece of information to our segment tree to try to get an $O(\log n)$ solution. Let's call this extra number the "lazy" number.



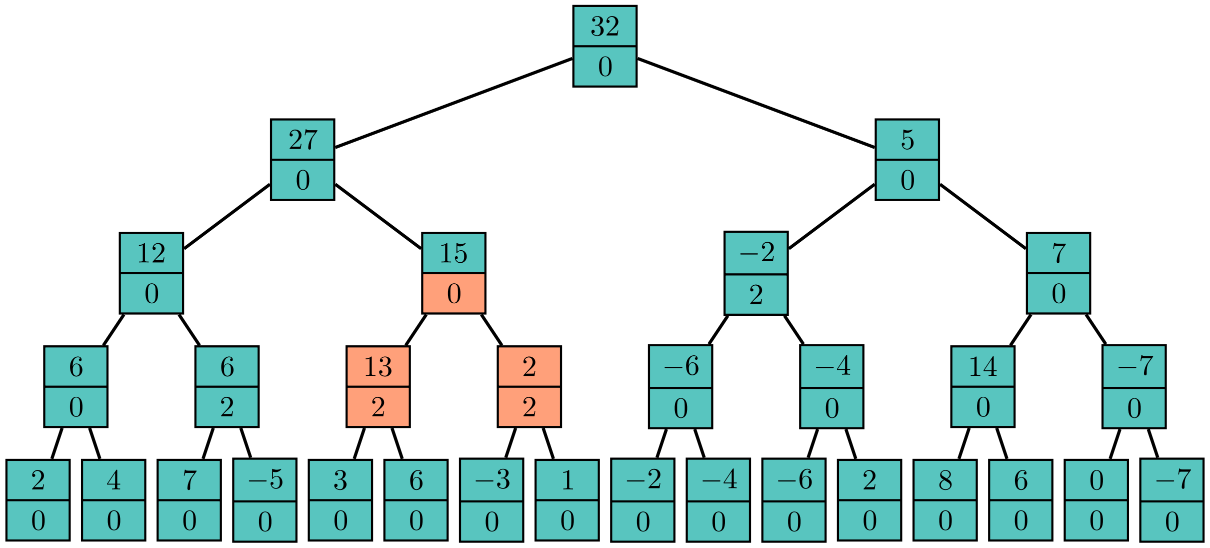
Once again, if the entire range associated with a node is contained within the update interval, we'll just make a note of it on that particular node and not update any of its children. We'll call such a node "lazy."

Here's the status of the tree after *update*$(3, 12, 2)$.



When a node is lazy, it indicates that the sum numbers of every other node in its subtree is no longer accurate. In particular, if a node is lazy, the sum number it keeps track of is not equal to the sum of the sum numbers of its children. This means whenever we need to access any node in that subtree, we'll need to update them then. Numbers in the tree therefore can change even after a query. Let's perform a query to illustrate that point.

query$(7, 13)$ requires access to the nodes for the ranges $[7, 8]$, $[9, 12]$, and $[13, 13]$. The nodes for $[9, 12]$ and $[13, 13]$ are up to date and store the correct sum, but the node for $[7, 8]$ does not, as it is a descendent of the node for $[5, 8]$, which has a nonzero lazy number. Highlighted are the nodes we'll need to update as we perform the query. Notice how $[5, 8]$ simply passes its lazy number onto its children, and they update themselves as necessary.



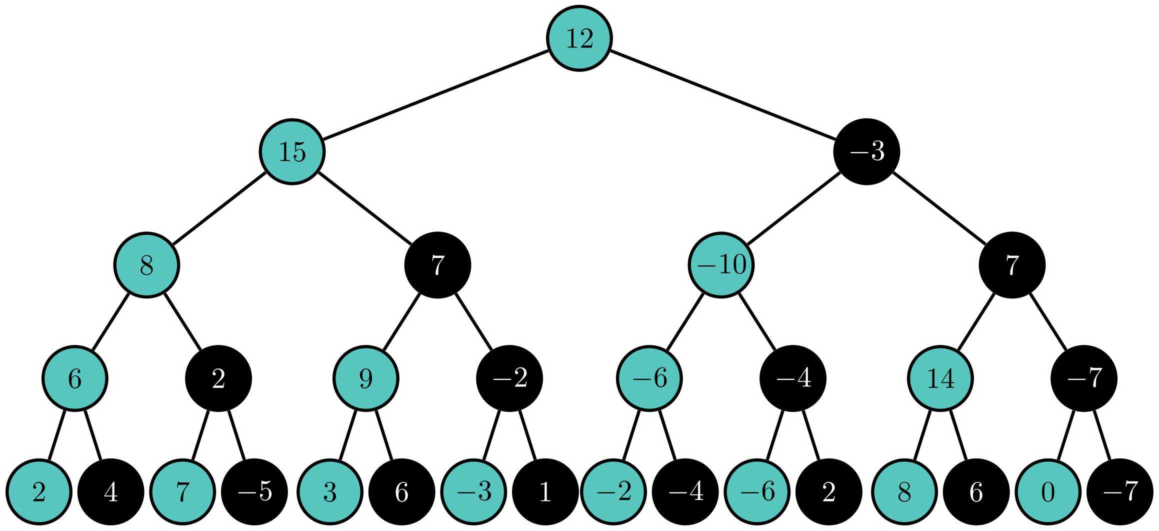
Now, the nodes that we need are all up to date, and we can perform our query. This process is necessary whenever we are performing an operation on an interval that intersects but does not contain the range associated with a lazy node. Since this can only happen for two such nodes (one at either end of the interval in question), both updating and querying change values of at most four nodes per level, so both operations are $O(\log n)$.

4 Fenwick Tree

A *Fenwick tree*, or *binary indexed tree (BIT)*, is simply a faster and easier to code segment tree when the operator in question has an inverse. Unfortunately, it's not at all intuitive, so bear with me at first and let the magic of the Fenwick tree reveal itself later. In fact, it is so magical that Richard Peng hates it. This is a fair sentiment, because it is rather gimmicky. The key idea is to compress the data stored within a segment tree in a crazy way that ends up having a really slick implementation using some bit operation tricks.

As discussed earlier, the $+$ operator has an inverse, $-$. Therefore, there is an inherent redundancy, for example, in keeping track of the sum of the first $\frac{n}{2}$ elements, the sum of all n elements, and the sum of the last $\frac{n}{2}$ elements, as we do in the segment tree. If we are given only $\sum_{k=1}^{n/2} a_k$ and $\sum_{k=1}^{n} a_k$, we can find $\sum_{k=n/2+1}^{n} a_k$ easily using subtraction.

With this in mind, let's ignore every right child in the tree. We'll mark them as black in the diagram. After that, we'll write out the tree nodes in postfix traversal order, without writing anything whenever we encounter a black node.



2	4	7	−5	3	6	−3	1	−2	−4	−6	2	8	6	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

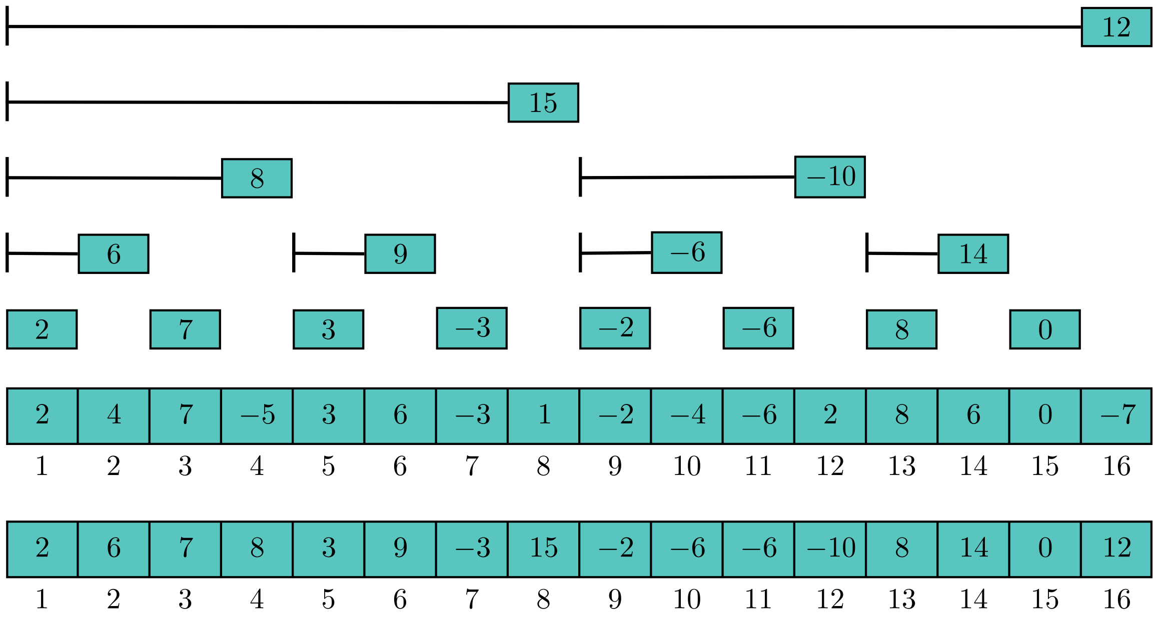
2	6	7	8	3	9	−3	15	−2	−6	−6	−10	8	14	0	12
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Our Fenwick tree is simply this last array. This should be quite confusing – it is not at all clear why this array resembles a tree, and the numbers in the array make no sense whatsoever right now.

Notice that the final position of every unblackened node is just the rightmost black child in its subtree. This leads to the fact that the ith element in the Fenwick tree array is the sum

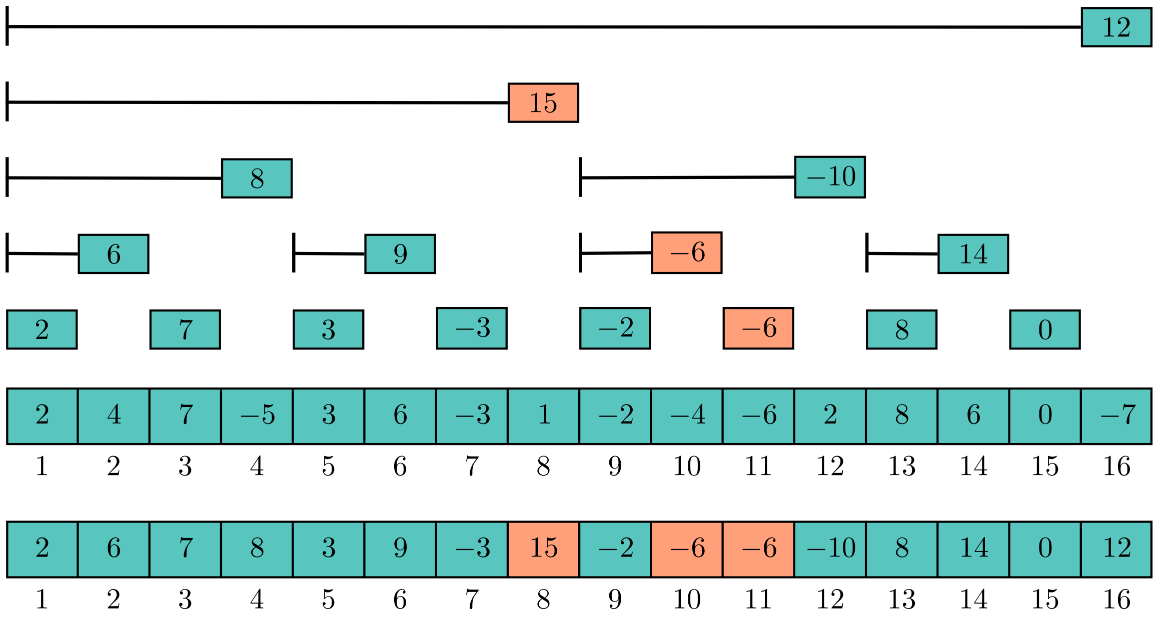
$$b_k = \sum_{k=i-2^{v_2(i)}+1}^{i} a_k,$$

where $2^{v_2(i)}$ is simply the greatest power of 2 that divides i. Let's look at a new diagram that hopefully will better illustrate this key property of the random array we just came up with.



All the framework is now in place. Now we need to find out how to query and update the Fenwick tree.

Suppose we wanted to find the sum $\sum_{k=1}^{11} a_k$. Let's take a look at the diagram to see which elements we need.

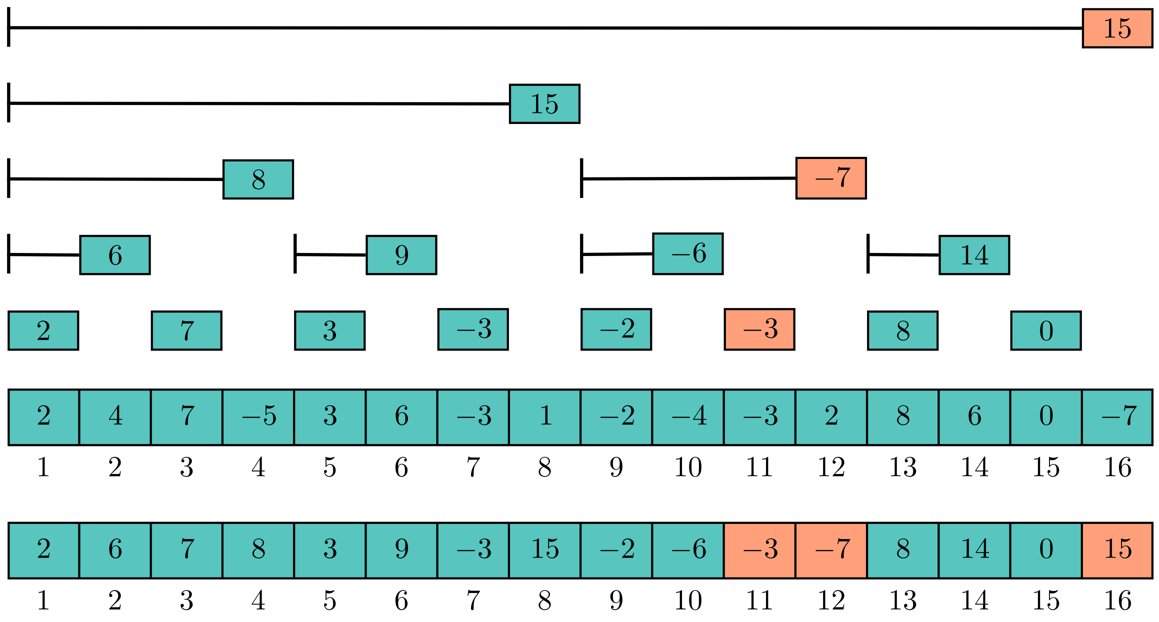


We see that the sum $\sum_{k=1}^{11} a_k = b_8 + b_{10} + b_{11}$. If we look at 11 in binary, we have $11 = 01011_2$. Let's see if we can find a pattern in these numbers in binary:

$$\begin{aligned} 11 &= 01011_2, \\ 10 = 11 - 2^{v_2(11)} &= 01010_2, \\ 8 = 10 - 2^{v_2(10)} &= 01000_2, \\ 0 = 8 - 2^{v_2(8)} &= 00000_2. \end{aligned}$$

So, we can simply subtract $11 - 2^{v_2(11)} = 10 = 01010_2$, find the sum of the first 10 elements, and add b_{11} to that sum to get the sum of the first 11 elements. We see that repeating this process takes off the last 1 in the binary representation of the number i, and since there are at most $\log n + 1$ 1s in the binary representation $\forall i \in [1, n]$, the query operation is $O(\log n)$.

And now for the update operation. Suppose we want to change the value of a_{11} from -6 to -3. Which numbers will we have to change?



We needed to increment the highlighted values, b_{11}, b_{12}, and b_{16}, by 3. Once again we'll look at 11, 12, and 16 in base 2.

$$\begin{aligned} 11 &= 01011_2, \\ 12 &= 01100_2 = 11 + 2^{v_2(11)}, \\ 16 &= 10000_2 = 12 + 2^{v_2(12)}. \end{aligned}$$

It appears that instead of subtracting the largest dividing power of 2, we are adding. Once again this is an $O(\log n)$ operation.

The real magic in the Fenwick tree is how quickly it can be coded. The only tricky part is finding exactly what $2^{v_2(i)}$ is. But it turns out, by the way bits are arranged in negative numbers, this is just `i & -i`. With this in mind, here's all the code that's necessary to code a Fenwick tree.

```
int[] b = new int[MAXN]; // Fenwick tree stored as array
void update(int i, int x) {
    for( ; i < MAXN; i += i & -i)
        b[i] += x;
}
int prefixSum(int i) {
    int sum = 0;
    for( ; i > 0; i -= i & -i)
        sum += b[i];
    return sum;
}
int query(int i, int j) {
    return prefixSum(j) - prefixSum(i - 1);
}
```