$\sqrt{n}$ Bucketing and Segment Tree

Samuel Hsiang

December 11, 2015

1 $\sqrt{n}$ Bucketing

$\sqrt{n}$ bucketing is a relatively straightforward idea – given n elements $\{x_i\}_{i=1}^n$ in a sequence, we group them into $\sqrt{n}$ equal-sized buckets. The motivation for arranging elements like this is to support an operation called a *range query*.

Let's take a concrete example. Suppose we want to support two operations:

- $update(p, k)$ – increment the value of x_p by k
- $query(a, b)$ – return $\sum_{i=a}^{b} x_i$.

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

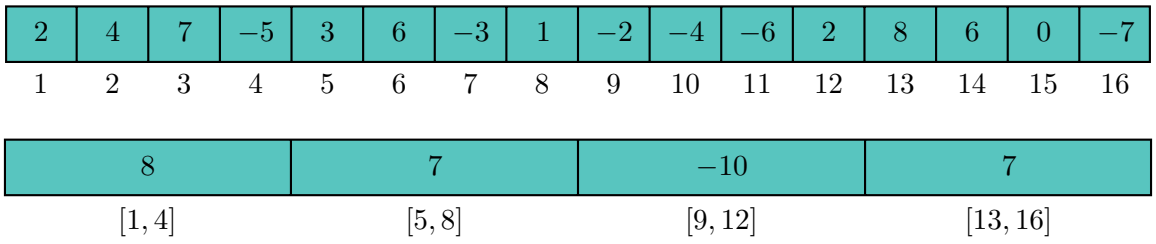
Another natural approach would be to store in a separate array the sum of the first i terms in the sequence for every index i, or store the *prefix sums*.

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

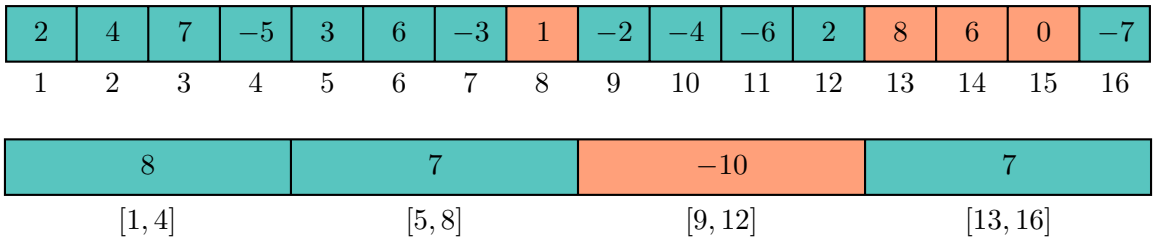
Now *query* becomes an $O(1)$ operation, as we can simply subtract two elements in the array to answer a query. Unfortunately, *update* becomes $O(n)$, as changing the value of an element in the beginning of the sequence forces us to change almost all the values in the prefix sum array.

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.

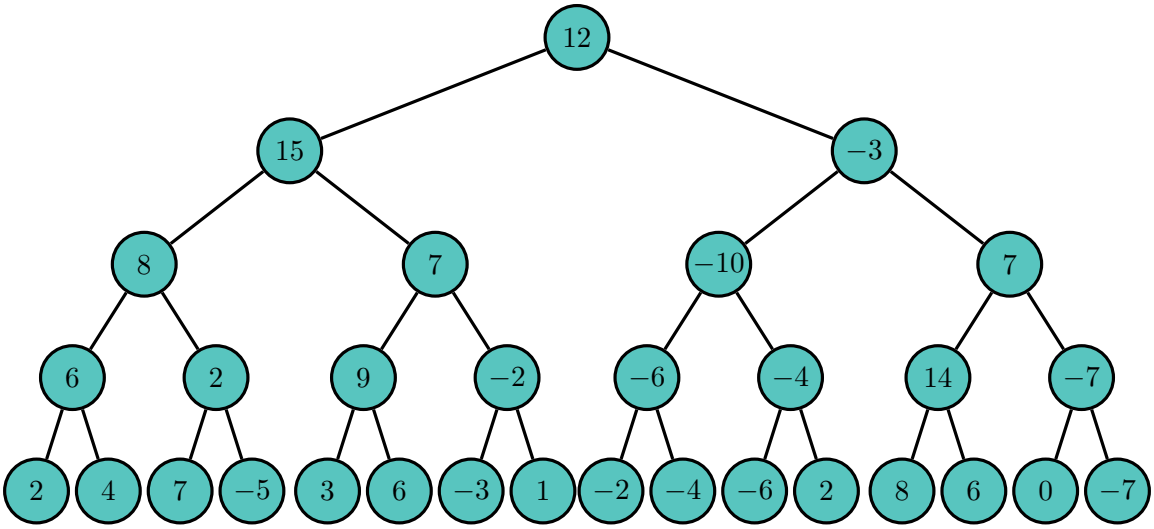
It is often the case that $O(\sqrt{n})$ bounds can be improved to $O(\log n)$ using more complex data structures like segment trees and more complex ideas like 2^n jump pointers, both of which are covered in this chapter. These are, however, more complicated to implement and as such are often comparable 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

For our range sum query problem, it turns out that we can do as well as $O(\log n)$ with a *segment tree*, also known as a *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.

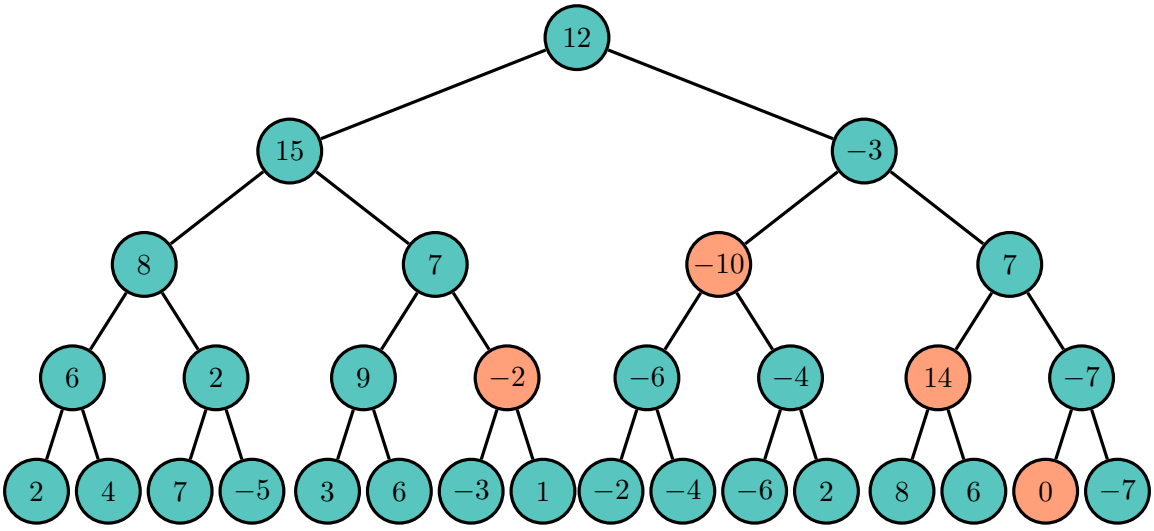
As the name "tree" suggests, we draw inspiration from a binary structure. Let's build a tree on top of the array, where each node keeps track of the sum of the numbers associated with its children. Every node keeps track of the *range* it is associated with and the *value* of the sum of all elements on that range. For example, the root of the tree is responsible for the sum of all elements in the range $[1, n]$, its left child is responsible for the range $\left[1, \left\lfloor \frac{1+n}{2} \right\rfloor\right]$ and its right child is responsible for $\left[\left\lfloor \frac{1+n}{2} \right\rfloor + 1, n\right]$.

In general, for the vertex responsible for the range $[l, r]$, its left child holds the sum for $\left[l, \left\lfloor \frac{l+r}{2} \right\rfloor\right]$ and its right child $\left[\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right]$. As we go down to the tree, eventually we'll have nodes with ranges $[l, l]$ that represent a single element in the original list. These, of course, will not have any 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 $[7, 15]$.

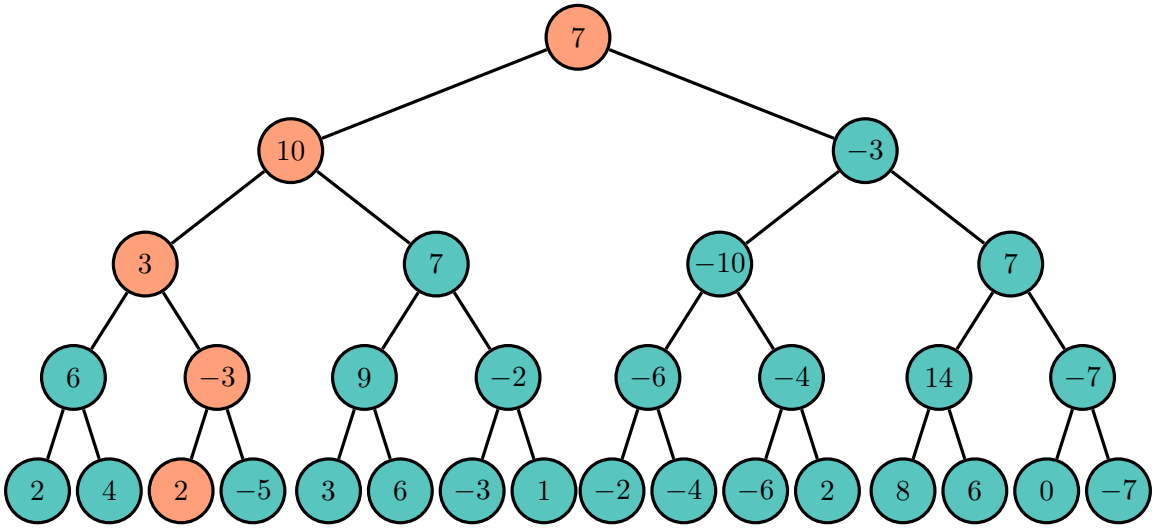


-2 represents the sum $x_7 + x_8$, -10 the sum $x_9 + x_{10} + x_{11} + x_{12}$, 14 the sum $x_{13} + x_{14}$, and 0 represents the single element x_{15}. It seems we always want to take the *largest* segments that stay within the range $[7, 15]$. But how do we know exactly which segments these are?

We handle queries using a recursive call, starting at the root of the tree. We then proceed as follows: If the current node's interval is completely disjoint from the queried interval, we return 0. If the current node's interval is completely contained within the queried interval, we return the sum associated with that node. Otherwise, we pass the query on to the node's two children. Note that this process is $O(\log n)$ because each level in the tree can have at most two highlighted nodes.

function QUERY(range $[l, r]$, range $[a, b]$) ▷ at node $[l, r]$, want $\sum_{i=a}^{b} x_i$
  if $r < a$ or $b < l$ **then** ▷ $[l, r] \cap [a, b] = \emptyset$
    return 0
  if $a \le l$ and $r \le b$ **then** ▷ $[l, r] \subseteq [a, b]$
    return $sum(l, r)$
  return QUERY$(\left[l, \left\lfloor \frac{l+r}{2} \right\rfloor\right], [a, b])$ + QUERY$(\left[\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right], [a, b])$

Segment trees also handle modifications well. If we want to change the third element to 2, then we have to update the highlighted nodes in the following diagram. We can implement this the same way we implement queries. Starting from the root, we update each modified node's children before recomputing the value stored at that node. The complexity is $O(\log n)$; we change the value of one node in each level of the tree.



function UPDATE(range $[l, r]$, p, k) ▷ $x_p \leftarrow x_p + k$
  if $r < p$ or $p < l$ **then** ▷ $p \notin [l, r]$
    return
  if $l = r$ **then** ▷ leaf node
    $sum(l, r) \leftarrow k$
    return
  UPDATE$(\left[l, \left\lfloor \frac{l+r}{2} \right\rfloor\right], p, k)$
  UPDATE$(\left[\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right], p, k)$
  $sum(l, r) \leftarrow sum(l, \left\lfloor \frac{l+r}{2} \right\rfloor) + sum(\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r)$

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. However, if we want a tree of size n, we can always round up to the nearest power of 2, which is at most $2n$. A perfectly balanced segment

tree of size n equaling a power of 2 requires $2n-1$ nodes representing our segments, so $4n$ bounds the memory we'll need.

Now all we need is a way for us to easily store the sum values associated with each node. We can clearly use a structure with two child pointers, but we can also exploit the fact that the segment tree structure is a balanced binary search tree. We can store the tree like we store a heap. The root is labeled 1, and for each node with label i, the left child is labeled $2i$ and the right child $2i+1$. Here's some sample code.

```
final int SZ = 1 << 17; // some sufficiently large power of 2
int[] sum = new int[2 * SZ]; // sum[1] contains sum for [1, SZ], and so on
int query(int i, int l, int r, int a, int b) {
    // i.e. query(1, 1, SZ, 7, 15)
    if(r < a || b < l) return 0;
    if(a <= l && r <= b) return tree[i];
    int m = (l + r) / 2;
    int ql = query(2 * i, l, m, a, b);
    int qr = query(2 * i + 1, m + 1, r, a, b);
    return ql + qr;
}
void update(int i, int l, int r, int p, int k) {
    // i.e. update(1, 1, SZ, 3, 2)
    if(r < p || p < l) return;
    if(l == r){
        sum[i] = k;
        return;
    }
    int m = (l + r) / 2;
    update(2 * i, l, m, p, k);
    update(2 * i + 1, m + 1, r, p, k);
    sum[i] = sum[2 * i] + sum[2 * i + 1];
}
```

Mathematically, what allows the segment tree to work on the addition operation lies in the fact that addition is an associative operation. This means that we can support a wide variety of other kinds of range queries, so long as the operation is associative. For example, we can also support range minimum queries and *gcd* and *lcm* queries. We can even combine different types of information stored at a node. One situation that requires this is maintaining the maximum prefix sum.

For our simple range sum query problem, we don't need the nice, completely balanced structure present when the number of elements is a nice power of two. However, it is necessary if we want to force the array *sum*[] to have the same nice properties as an actual heap so we can perform nice iterative operations on our tree, as previously, all tree operations were recursive. It is also necessary if we need the numbers representing the indices in our tree to have special properties, as in the Fenwick tree.

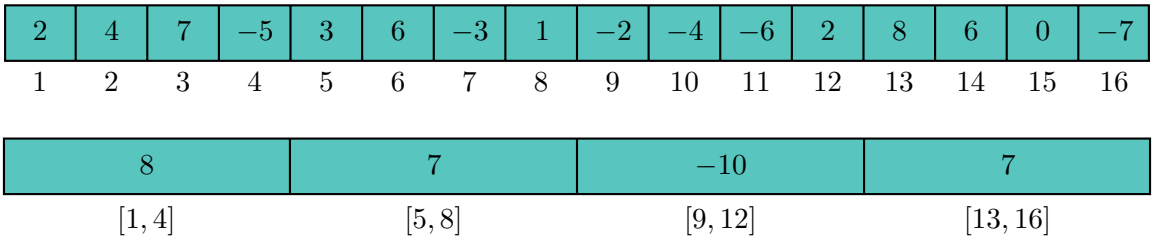
2.1 Lazy Propagation

It is often the case that in addition to performing range queries, we need 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*(a, b, k) – increment the value of x_i by k for all $i \in [a, b]$
- *query*(a, b) – return $\sum_{i=a}^{b} x_i$.

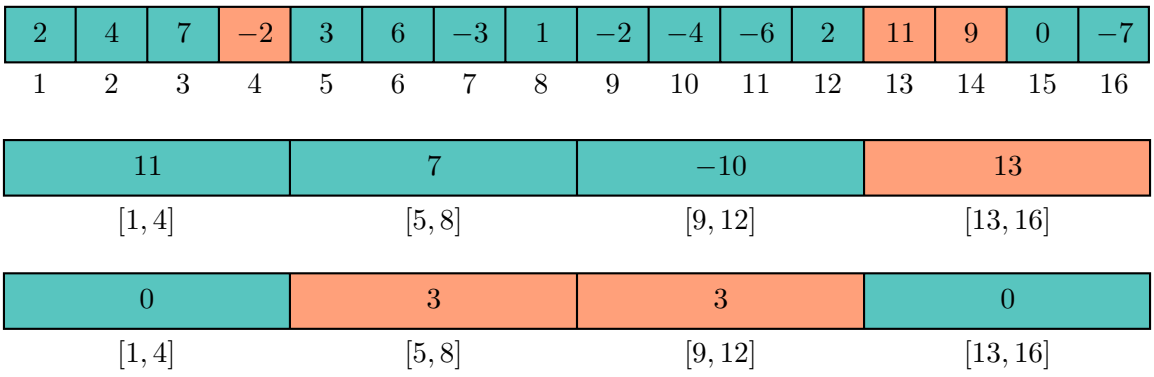
2.1.1 Some Motivation: $\sqrt{n}$ Blocking

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.*



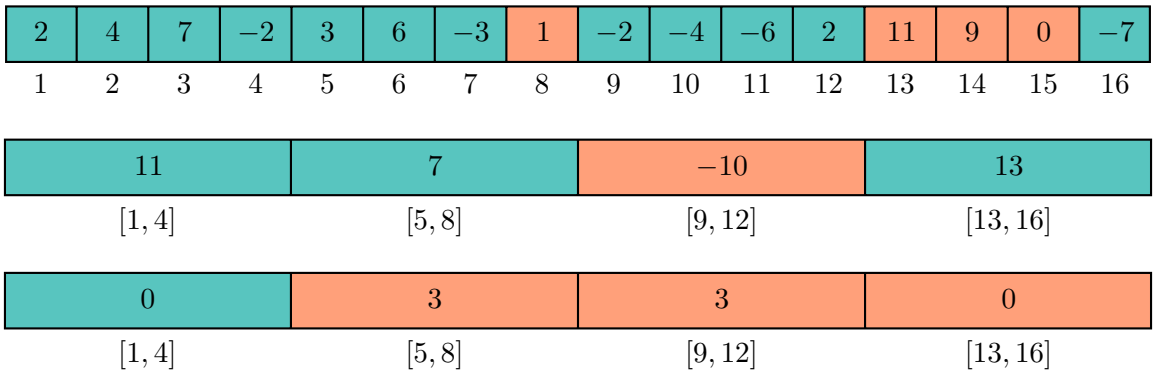
Can we do something similar for *update*? The case we need to worry about is when an entire bucket is included in the update range. Again, we don't want to touch the original array a at all, since that makes the operation linear. Instead, whenever we update an entire bucket, we track the information about the update separately. Thus we store a value for each bucket indicating the amount by which we've incremented that entire bucket.

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



To reiterate, we are not storing the actual values of the elements or buckets in the arrays, where they were stored when we solved the original formulation of the problem. Despite this fact, we can still calculate the value of any element or bucket. a_i is equal to the sum of the $\left\lceil \frac{i}{\sqrt{n}} \right\rceil$th value stored in the third array and the ith value stored in the first array. The sum of any given bucket can be calculated similarly. However, we must remember to adjust for bucket size. In the example, there are four elements per bucket, so we have to add $4 \cdot 3 = 12$ to get the correct sum for an updated bucket. Because of all this, we can query a range exactly like we did without range updates.

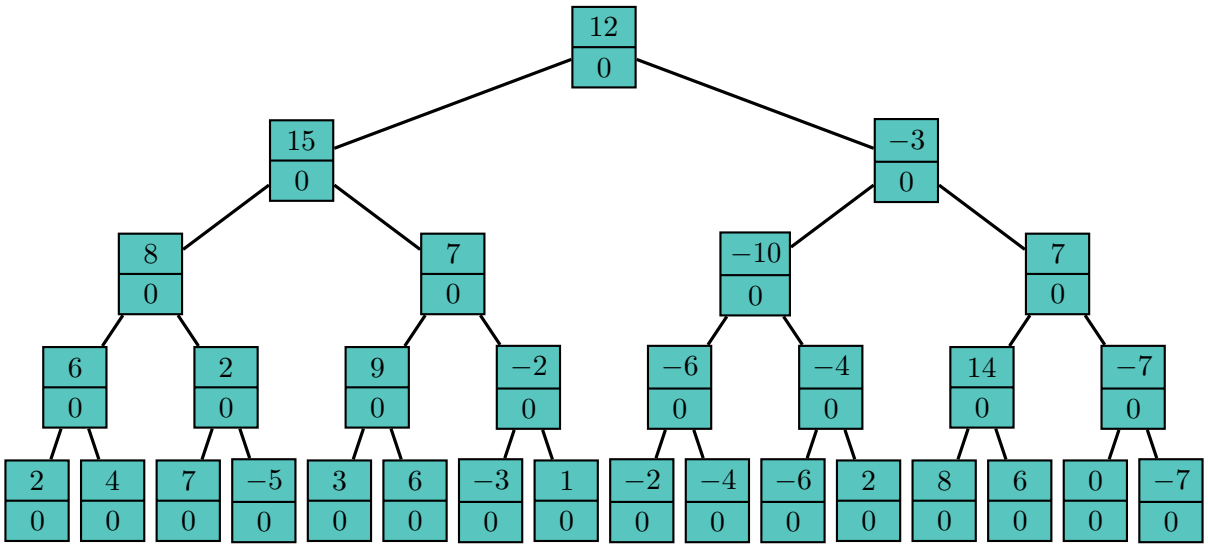
Highlighted are the values necessary for *query*$(7, 15)$.



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

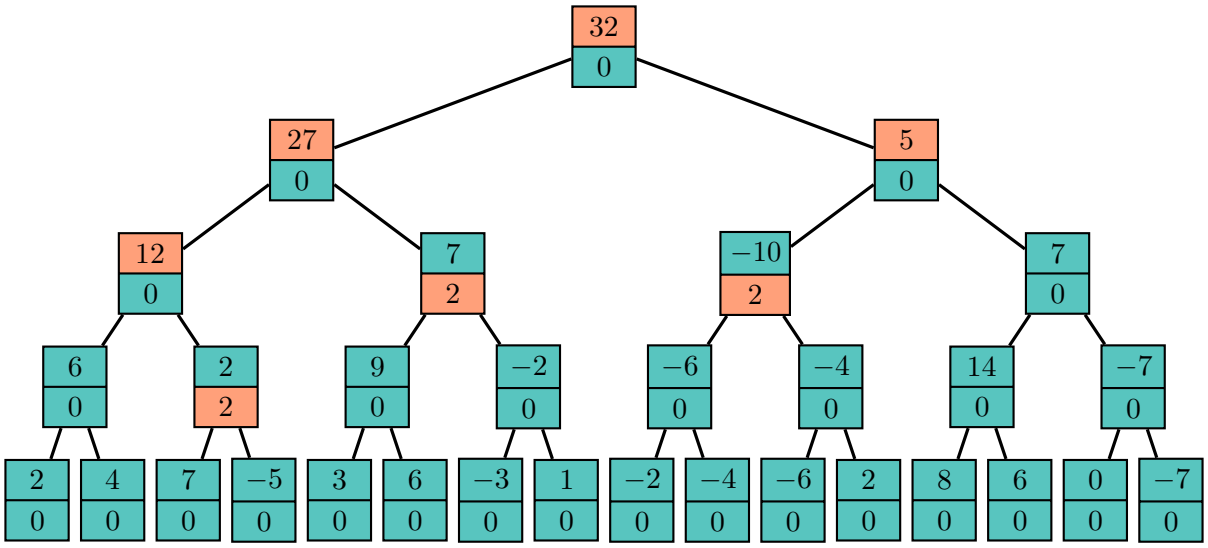
2.1.2 Lazy Propagation on a Segment Tree

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



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 that node or any of its children. We'll call such a node "lazy."

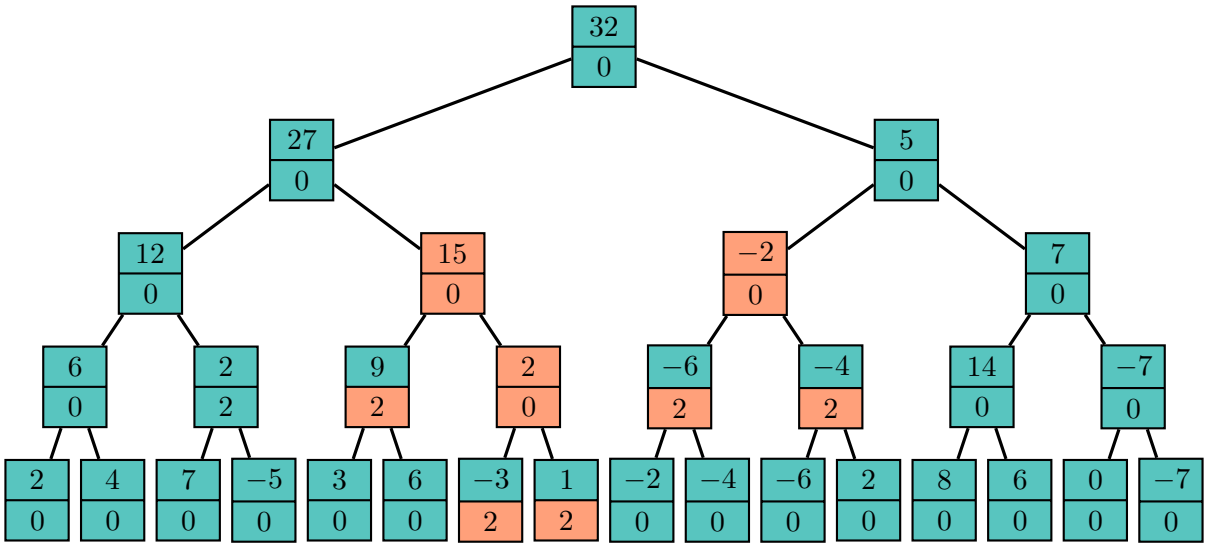
Here's the status of the tree after $update(3, 12, 2)$.



When a node is lazy, it indicates that the sum numbers of every node in its subtree are no longer accurate. In particular, if a node is lazy, the sum number it holds is not equal to the sum of the values in its leaves. This means that whenever we need to access any node in the subtree of that node, we'll need to update some values.

Suppose we encounter a lazy node while traversing down the tree. In order to get accurate sum values in that node's subtree, we need to apply the changes indicated by its lazy value. Thus we update the node's sum value, incrementing by the lazy value once for each leaf in the node's subtree. In addition, we have to propagate the lazy value to the node's children. We do this by incrementing each child's lazy value by our current node's lazy value. And finally, we need to set the lazy value to 0, to indicate that there's nothing left to update. When we implement a segment tree, all of this is usually encapsulated within a "push" function.

Let's illustrate querying with an example: $query(7, 13)$. To answer this, we need to access the nodes for the ranges $[7, 8]$, $[9, 12]$, and $[13, 13]$. The node for $[13, 13]$ is up-to-date and stores the correct sum. However, the other two nodes do not. (The node for $[7, 8]$ is in the subtree of the node for $[5, 8]$, which is lazy.) Thus as we recurse, we push our lazy values whenever we encounter them. Highlighted are the nodes we'll need to update for the query. Notice how $[5, 8]$ and $[9, 12]$ simply pass their lazy numbers to their children, where they'll update themselves when necessary.



The complexity of querying remains the same, even when propagating lazy values. We have to push at most once for each node we encounter, so our runtime is multiplied only by a constant factor. Thus, like normal segment tree queries, lazy segment tree queries also run in $O(\log n)$.

While we're thinking about complexity, we can also ask ourselves why it *doesn't* take $O(n)$ time to update $O(n)$ nodes. With lazy propagation, we take advantage of two things. The first is that on each query, we access very few nodes, so as long as the nodes we access are up-to-date, we're all set. The second is that we can combine updates while they're still being propagated, that the update operation is associative. This allows us to update only when it's absolutely necessary—the rest of the time, we can be lazy. (The next time someone tells you you're being lazy, you can say it's a good thing.)

Like normal segment trees, lazily propagating segment trees can handle a diverse set of range updates and range queries. We can support an update, where instead of *incrementing* each element on a range *by* a certain value, we *set* each element *to* that value. There is no difference between the two as point updates, but they are very different operations when applied as range updates. Sometimes, we can even have more than one range update on a single tree. When implementing this, however, it is important to be careful when pushing lazy values—composing different operations can become quite complicated.

Below is my implementation of a segment tree supporting range sum and range increment. Note that it includes one detail that we left out in our development of lazy propagation above: how we update a node whose range partially intersects the updated range. One way to do this is to calculate the length of the intersection and update directly. However, this does not work well for queries that are not as nice as incrementing, such as setting an entire range to a value. Instead, we first update the children of this node. Then we push the lazy values off the children so their sum values are accurate. This allows us to recalculate the sum value of the parent like we do for a non-lazy segtree. I have this as my `pull` function below.

```
final int SZ = 1 << 17;
int[] sum = new int[2 * SZ];
int[] lazy = new int[2 * SZ];

void push(int i, int l, int r){
  if(lazy[i] != 0){
    sum[i] += (r - l + 1) * lazy[i];
    if(l < r){
      lazy[2 * i] += lazy[i];
      lazy[2 * i + 1] += lazy[i];
    }
    lazy[i] = 0;
  }
}

void pull(int i, int l, int r){
  int m = (l + r) / 2;
  push(2 * i, l, m);
  push(2 * i + 1, m + 1, r);
  sum[i] = sum[2 * i] + sum[2 * i + 1];
}

int query(int i, int l, int r, int a, int b) {
  push(i, l, r);
  if(r < a || b < l) return 0;
  if(a <= l && r <= b){
    return sum[i];
  }
  int m = (l + r) / 2;
  int ql = query(2 * i, l, m, a, b);
  int qr = query(2 * i + 1, m + 1, r, a, b);
  return ql + qr;
}

void update(int i, int l, int r, int a, int b, int k) {
  // push(i, l, r); // Necessary for non-commutative range updates.
  if(r < a || b < l) return;
  if(a <= l && r <= b){
    lazy[i] += k;
    return;
  }
  int m = (l + r) / 2;
  update(2 * i, l, m, a, b, k);
  update(2 * i + 1, m + 1, r, a, b, k);
  pull(i, l, r);
}
```

2.2 Fenwick Tree

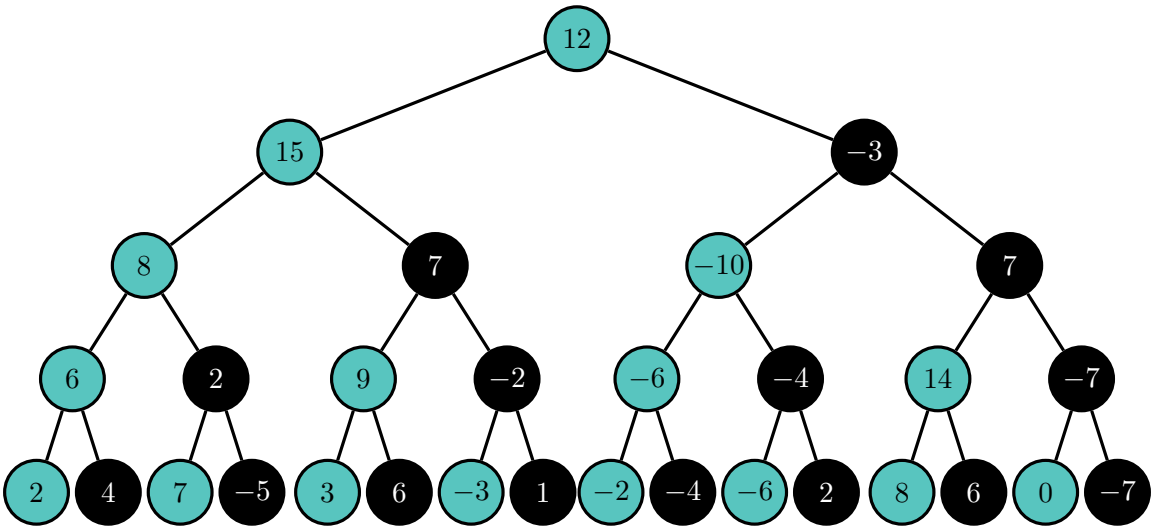
A *Fenwick tree*, or *binary indexed tree (BIT)*, is simply a faster and easier to code segment tree when the operator in question, in addition to being associative, has an inverse. Unfortunately, it's not at all intuitive, so bear with me at first and let the magic of the Fenwick tree reveal itself later.[1]

[1] In fact, it is so magical that Richard Peng hates it because it is too 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

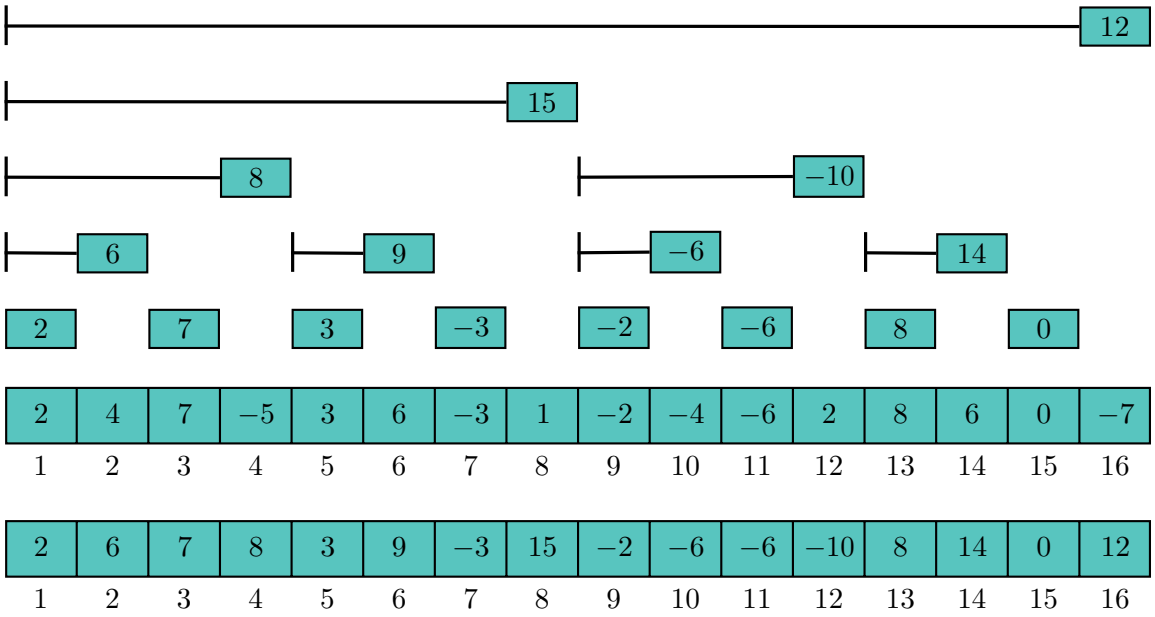
2	6	7	8	3	9	−3	15	−2	−6	−6	−10	8	14	0	12
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

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

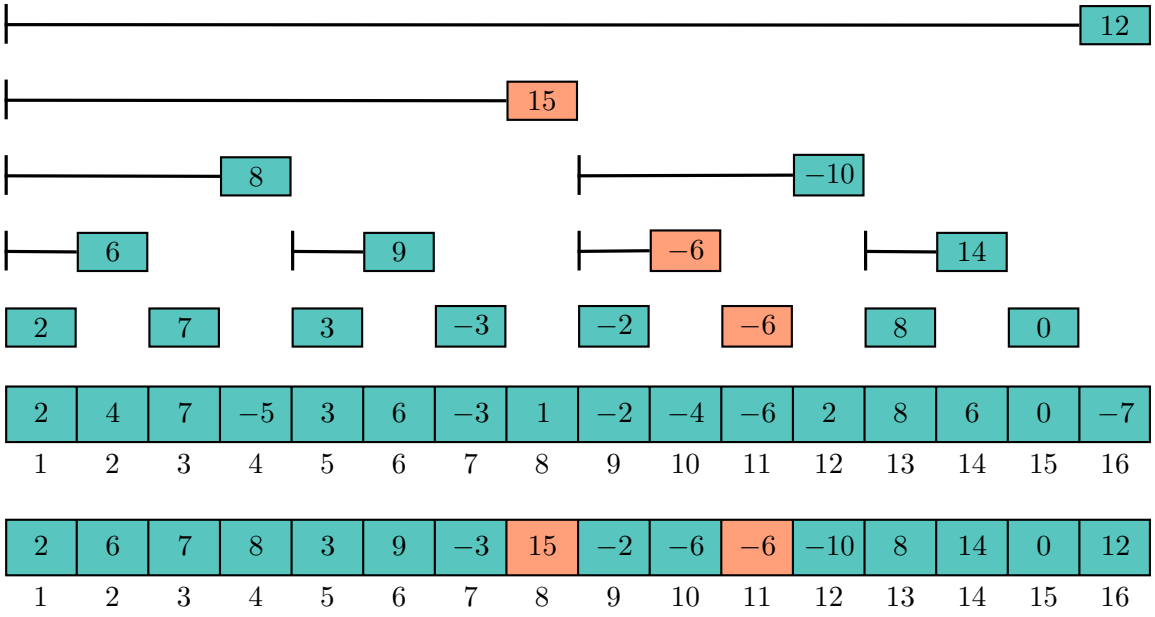
$$y_i = \sum_{k=i-2^{v_2(i)}+1}^{i} x_k,$$

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

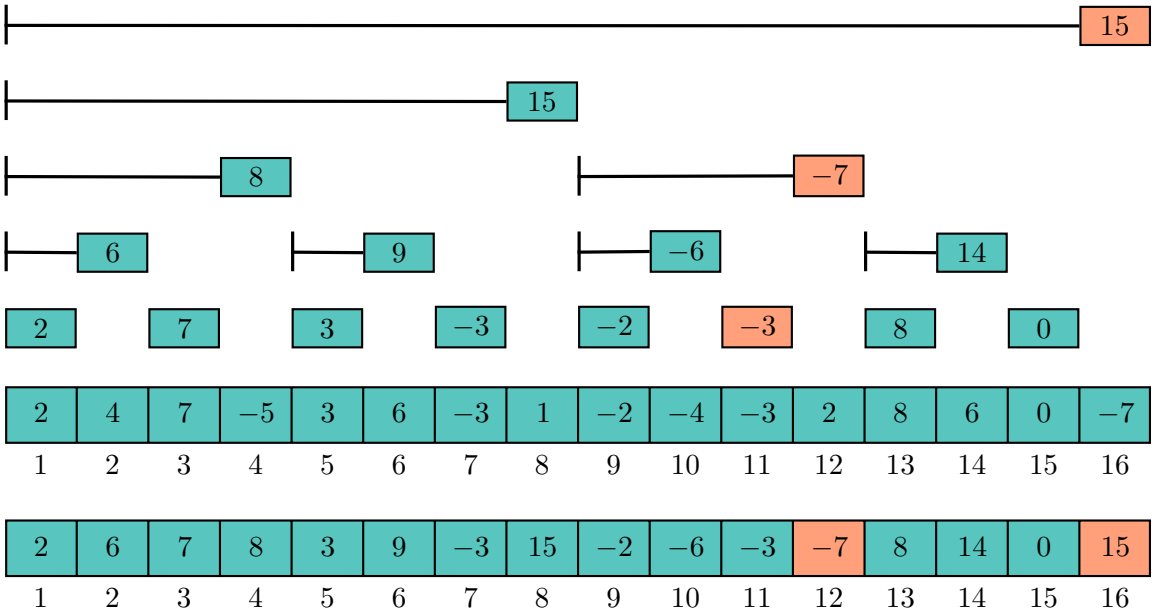


We see that the sum $\sum_{k=1}^{11} x_k = y_8 + y_{10} + y_{11}$. If we look at 11 in binary, we have $11 = 01011_2$. Let's see if we can find a pattern in these numbers in binary:

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

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 x_{11} from -6 to -3. Which numbers will we have to change?



We needed to increment the highlighted values, y_{11}, y_{12}, and y_{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. 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. Note that here, values in the array remain 1-indexed, which is different from how we code segment trees.

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