Journal of Functional Programming, pages, 2024.
doi:10.1017/xxxxx

FUNCTIONAL PEARL

You Could Have Invented Fenwick Trees

BRENT A. YORGEY

Hendrix College
1600 Washington Ave, Conway, AR 72032, USA
e-mail: yorgey@hendrix.edu

Abstract

Fenwick trees, also known as *binary indexed trees*, are a clever solution to the problem of maintaining a sequence of values while allowing both updates and range queries in sublinear time. Their implementation is concise and efficient—but also somewhat baffling, consisting largely of nonobvious bitwise operations on indices. We begin with *segment trees*, a much more straightforward, easy-to-verify, purely functional solution to the problem, and use equational reasoning to explain the implementation of Fenwick trees as an optimized variant, making use of a Haskell EDSL for operations on infinite two's complement binary numbers.

1 Introduction

Suppose we have a sequence of n integers $a_1, a_2, \ldots, a_n$, and want to be able to perform arbitrary interleavings of the following two operations, illustrated in Figure 1:

- *Update* the value at any given index[1] i by adding some value v.
- Find the sum of all values in any given range $[i, j]$, that is, $a_i + a_{i+1} + \cdots + a_j$. We call this operation a *range query*.

Note that update is phrased in terms of *adding* some value v to the existing value; we can also *set* a given index to a new value v by adding $v - u$, where u is the old value.

If we simply store the integers in a mutable array, then we can update in constant time, but range queries require time linear in the size of the range, since we must iterate through the entire range $[i, j]$ to add up the values.

In order to improve the running time of range queries, we could try to cache (at least some of) the range sums. However, this must be done with care, since the cached sums must be kept up to date when updating the value at an index. For example, a straightforward approach would be to use an array P where P_i stores the prefix sum $a_1 + \cdots + a_i$; P can be precomputed in linear time via a scan. Now range queries are fast: we can obtain $a_i + \cdots + a_j$ in constant time by computing $P_j - P_{i-1}$ (for convenience we set $P_0 = 0$ so

[1] Note that we use 1-based indexing here and throughout the paper, that is, the first item in the sequence has index 1. The reasons for this choice will become clear later.

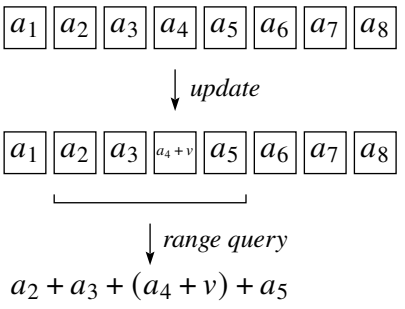


Fig. 1. Update and range query operations

this works even when $i = 1$). Unfortunately, it is update that now takes linear time, since changing a_i requires updating P_j for every $j \geqslant i$.

Is it possible to design a data structure that allows *both* operations to run in sublinear time? (You may wish to pause and think about it before reading the next paragraph!) This is not just academic: the problem was originally considered in the context of *arithmetic coding* (Rissanen and Langdon, 1979; Bird and Gibbons, 2002), a family of techniques for turning messages into sequences of bits for storage or transmission. In order to minimize the bits required, one generally wants to assign shorter bit sequences to more frequent characters, and vice versa; this leads to the need to maintain a dynamic table of character frequencies. We *update* the table every time a new character is processed, and *query* the table for cumulative frequencies in order to subdivide a unit interval into consecutive segments proportional to the frequency of each character (Fenwick, 1994; Ryabko, 1989).

So, can we get both operations to run in sublinear time? The answer, of course, is yes. One simple technique is to divide the sequence into $\sqrt{n}$ buckets, each of size $\sqrt{n}$, and create an additional array of size $\sqrt{n}$ to cache the sum of each bucket. Updates still run in $O(1)$, since we simply have to update the value at the given index and the corresponding bucket sum. Range queries now run in $O(\sqrt{n})$ time: to find the sum $a_i + \cdots + a_j$, we manually add the values from a_i to the end of its bucket, and from a_j to the beginning of its bucket; for all the buckets in between we can just look up their sum.

We can make range queries even faster, at the cost of making updates slightly slower, by introducing additional levels of caching. For example, we can divide the sequence into $\sqrt[3]{n}$ "big buckets", and then further subdivide each big bucket into $\sqrt[3]{n}$ "small buckets", with each small bucket holding $\sqrt[3]{n}$ values. The sum of each bucket is cached; now each update requires modifying three values, and range queries run in $O(\sqrt[3]{n})$ time.

In the limit, we end up with a binary divide-and-conquer approach to caching range sums, with both update and range query taking $O(\lg n)$ time. In particular, we can make a balanced binary tree where the leaves store the sequence itself, and every internal node stores the sum of its children. (This will be a familiar idea to many functional programmers; for example, finger trees (Hinze and Paterson, 2006; Apfelmus, 2009) use a similar sort of caching scheme.) The resulting data structure is popularly known as a *segment tree*[2],

[2] There is some confusion of terminology here. As of this writing, the Wikipedia article on *segment trees* (Wikipedia contributors, 2024) is about an interval data structure used in computational geometry. However, most of the Google search results for "segment tree" are from the world of competitive programming, where

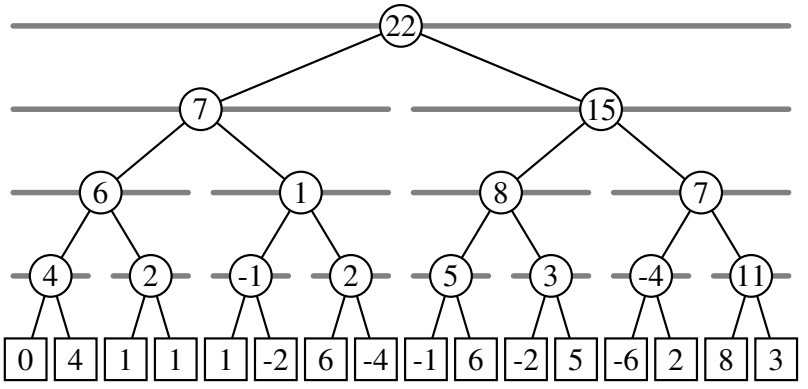


Fig. 2. A segment tree

presumably because each internal node ultimately caches the sum of a (contiguous) *segment* of the underlying sequence. Figure 2 shows a segment tree built on a sample array of length $n = 16$ (for simplicity, we will assume that n is a power of two, although it is easy to generalize to situations where it is not). Each leaf of the tree corresponds to an array entry; each internal node is drawn with a grey bar showing the segment of the underlying array of which it is the sum.

Let's see how we can use a segment tree to implement the two required operations so that they run in logarithmic time.

- To update the value at index i, we also need to update any cached range sums which include it. These are exactly the nodes along the path from the leaf at index i to the root of the tree; there are $O(\lg n)$ such nodes. Figure 3 illustrates this update process for the example segment tree from Figure 2; updating the entry at index 5 requires modifying only the shaded nodes along the path from the root to the updated entry.
- To perform a range query, we descend through the tree while keeping track of the range covered by the current node.
  - If the range of the current node is wholly contained within the query range, return the value of the current node.
  - If the range of the current node is disjoint from the query range, return 0.
  - Otherwise, recursively query both children and return the sum of the results.

  Figure 4 illustrates the process of computing the sum of the range $[4 \ldots 11]$. Blue nodes are the ones we recurse through; green nodes are those whose range is wholly contained in the query range, and are returned without recursing further; grey nodes are disjoint from the query range and return zero. The final result in this example is the sum of values at the green nodes, $1 + 1 + 5 + -2 = 5$ (it is easily verified that this is in fact the sum of values in the range $[4 \ldots 11]$).

  On this small example tree, it may seem that we visit a significant fraction of the total nodes, but in general, we visit no more than about $4 \lg n$. Figure 5 makes this more

it refers to the data structure considered in this paper (see, for example, Halim et al. (2020, §2.8) or Ivanov (2011)). The two data structures are largely unrelated.

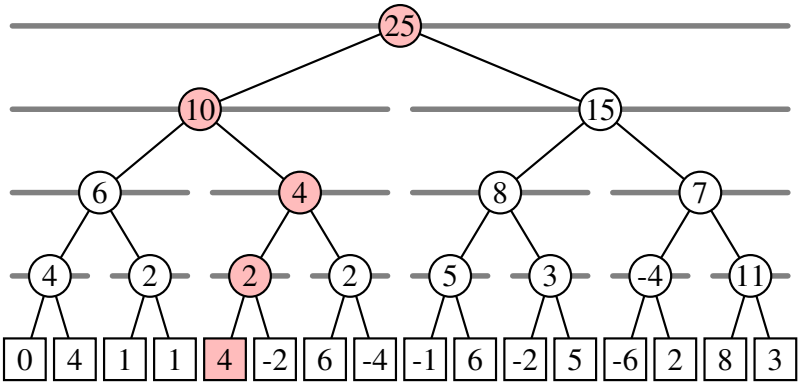


Fig. 3. Updating a segment tree

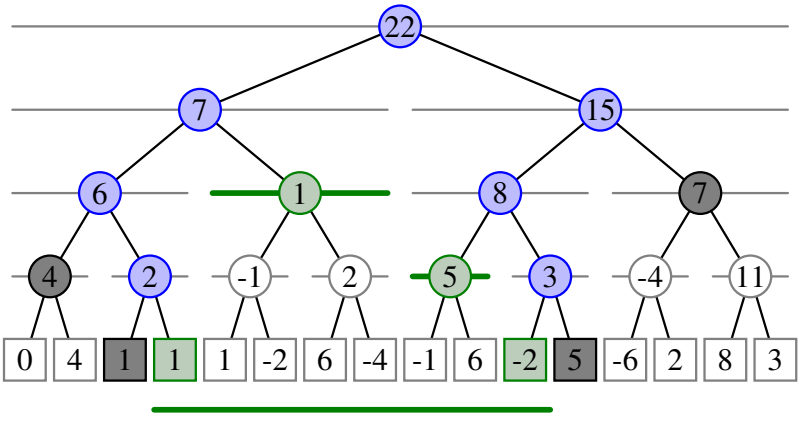


Fig. 4. Performing a range query on a segment tree

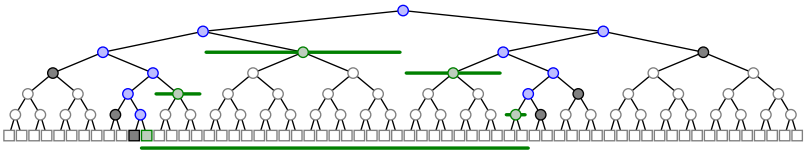


Fig. 5. Performing a range query on a larger segment tree

clear. Only one blue node in the entire tree can have two blue children, and hence each level of the tree can contain at most two blue nodes and two non-blue nodes. We essentially perform two binary searches, one to find each endpoint of the query range.

Segment trees are a very nice solution to the problem: as we will see in Section 2, they fit well in a functional language; they also lend themselves to powerful generalizations

```
class FenwickTree {
    private long[] a;
    public FenwickTree(int n) { a = new long[n+1]; }
    public long prefix(int i) {
        long s = 0;
        for (; i > 0; i -= LSB(i)) s += a[i]; return s;
    }
    public void update(int i, long delta) {
        for (; i < a.length; i += LSB(i)) a[i] += delta;
    }
    public long range(int i, int j) {
        return prefix(j) - prefix(i-1);
    }
    public long get(int i) { return range(i,i); }
    public void set(int i, long v) { update(i, v - get(i)); }
    private int LSB(int i) { return i & (-i); }
}
```

Fig. 6. Implementing Fenwick trees with bit tricks

such as lazily propagated range updates and persistent update history via shared immutable structure (Ivanov, 2011).

Fenwick trees, or *binary indexed trees* (Fenwick, 1994; Ivanov, 2011), are an alternative solution to the problem. What they lack in generality, they make up for with an extremely small memory footprint—they require literally nothing more than an array storing the values in the tree—and a blazing fast implementation. In other words, they are perfect for applications such as low-level coding/decoding routines where we don't need any of the advanced features that segment trees offer, and want to squeeze out every last bit of performance.

Figure 6 shows a typical implementation of Fenwick trees in Java. As you can see, the implementation is incredibly concise, and consists mostly of some small loops doing just a few arithmetic and bit operations per iteration. It is not at all clear what this code is doing, or how it works! Upon closer inspection, the `range`, `get`, and `set` functions are straightforward, but the other functions are a puzzle. We can see that both the `prefix` and `update` functions call another function `LSB`, which for some reason performs a bitwise logical AND of an integer and its negation. In fact, `LSB(x)` computes the *least significant bit* of x, that is, it returns the smallest 2^k such that the kth bit of x is a one. However, it is not obvious how the implementation of `LSB` works, nor how and why least significant bits are being used to compute updates and prefix sums.

Our goal is *not* to write elegant functional code for this—already solved!—problem. Rather, our goal will be to use a functional domain-specific language for bit strings, along with equational reasoning, to *derive* and *explain* this baffling imperative code from first principles—a demonstration of the power of functional thinking and equational reasoning to understand code written even in other, non-functional languages. After developing more intuition for segment trees (Section 2), we will see how Fenwick trees can be viewed as a variant on segment trees (Section 3). We will then take a detour into two's complement binary encoding, develop a suitable DSL for bit manipuations, and explain the implementation of the `LSB` function (Section 4). Armed with the DSL, we will then derive functions for converting back and forth between Fenwick trees and standard binary trees (Section 5). Finally, we will be able to derive functions for moving within a Fenwick tree by converting

```
type Index = Int
data Range = Index :—: Index    -- (a :—: b) represents the closed interval [a, b]
  deriving (Eq, Show)
(⊆) :: Range → Range → Bool
(lo1 :—: hi1) ⊆ (lo2 :—: hi2) = lo2 ⩽ lo1 ∧ hi1 ⩽ hi2
(∈) :: Index → Range → Bool
k ∈ i = (k :—: k) ⊆ i
disjoint :: Range → Range → Bool
disjoint (lo1 :—: hi1) (lo2 :—: hi2) = hi1 < lo2 ∨ hi2 < lo1
```

Fig. 7. Range utilities

to binary tree indices, doing the obvious operations to effect the desired motion within the binary tree, and then converting back. Fusing away the conversions via equational reasoning will finally reveal the hidden LSB function, as expected (Section 6).

This paper was produced from a literate Haskell document; the source is available from GitHub, at https://github.com/byorgey/fenwick/blob/master/Fenwick.lhs.

2 Segment Trees

Figure 8 exhibits a simple implementation of a segment tree in Haskell, using some utilities for working with index ranges shown in Figure 7. We store a segment tree as a recursive algebraic data type, and implement *update* and *rq* using code that directly corresponds to the recursive descriptions given in the previous section; *get* and *set* can then also be implemented in terms of them. It is not hard to generalize this code to work for segment trees storing values from either an arbitrary commutative monoid if we don't need the *set* operation—or from an arbitrary Abelian group (*i.e.* commutative monoid with inverses) if we do need *set*—but we keep things simple since the generalization doesn't add anything to our story.

Although this implementation is simple and relatively straightforward to understand, compared to simply storing the sequence of values in an array, it incurs a good deal of overhead. We can be more clever in our use of space by storing all the nodes of a segment tree in an array, using the standard left-to-right breadth-first indexing scheme illustrated in Figure 9 (for example, this scheme, or something like it, is commonly used to implement binary heaps). The root has label 1; every time we descend one level we append an extra bit: 0 when we descend to the left child and 1 when we descend to the right. Thus, the index of each node expressed in binary records the sequence of left-right choices along the path to that node from the root. Going from a node to its children is as simple as doing a left bit-shift and optionally adding 1; going from a node to its parent is a right bit-shift. This defines a bijection from the positive natural numbers to the nodes of an infinite binary tree. If we label the segment tree array with $s_1 \ldots s_{2n-1}$, then s_1 stores the sum of all the a_i, s_2

```
data SegTree where
  Empty  :: SegTree
  Branch :: Integer → Range → SegTree → SegTree → SegTree

update :: Index → Integer → SegTree → SegTree
update _ _ Empty = Empty
update i v b@(Branch a rng l r)
  | i ∈ rng      = Branch (a + v) rng (update i v l) (update i v r)
  | otherwise = b

rq :: Range → SegTree → Integer
rq _ Empty = 0
rq q (Branch a rng l r)
  | disjoint rng q = 0
  | rng ⊆ q        = a
  | otherwise      = rq q l + rq q r

get :: Index → SegTree → Integer
get i = rq (i :—: i)

set :: Index → Integer → SegTree → SegTree
set i v t = update i (v − get i t) t
```

Fig. 8. Simple segment tree implementation in Haskell

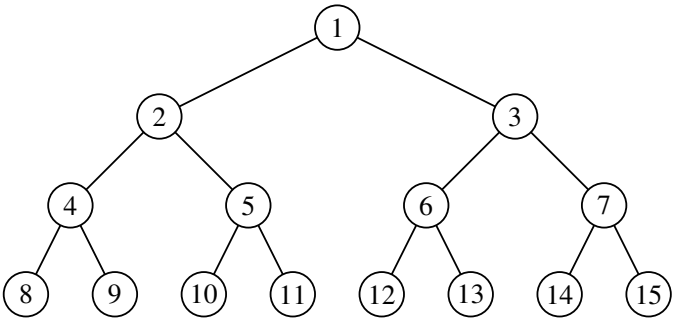


Fig. 9. Indexing a binary tree

stores the sum of the first half of the a_i, s_3 stores the sum of the second half, and so on. $a_1 \ldots a_n$ themselves are stored as $s_n \ldots s_{2n-1}$.

The important point is that since descending recursively through the tree corresponds to simple operations on indices, all the algorithms we have discussed can be straightforwardly transformed into code that works with a (mutable) array: for example, instead of storing a reference to the current subtree, we store an integer index; every time we want to descend to the left or right we simply double the current index or double and add one; and so on. Working with tree nodes stored in an array presents an additional opportunity: rather than being forced to start at the root and recurse downwards, we can start at a particular index of interest and move *up* the tree instead.

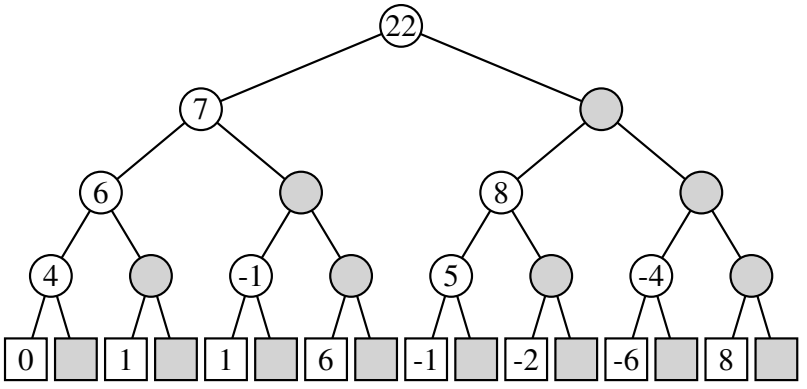


Fig. 10. Inactivating all right children in a segment tree

So how do we get from segment trees to Fenwick trees? We start with an innocuous-seeming observation: *not all the values stored in a segment tree are necessary*. Of course, all the non-leaf nodes are "unnecessary" in the sense that they represent cached range sums which could easily be recomputed from the original sequence. That's the whole point: caching these "redundant" sums trades off space for time, allowing us to perform arbitrary updates and range queries quickly, at the cost of doubling the required storage space.

But that's not what I mean! In fact, there is a different set of values we can forget about, but in such a way that we still retain the logarithmic running time for updates and range queries. Which values, you ask? Simple: just forget the data stored in *every node which is a right child*. Figure 10 shows the same example tree we have been using, but with the data deleted from every right child. Note that "every right child" includes both leaves and internal nodes: we forget the data associated to *every* node which is the right child of its parent. We will refer to the nodes with discarded data as *inactive* and the remaining nodes (that is, left children and the root) as *active*. We also say that a tree with all its right children inactivated in this way has been *thinned*.

Updating a thinned segment tree is easy: just update the same nodes as before, ignoring any updates to inactive nodes. But how do we answer range queries? It's not too hard to see that there is enough information remaining to reconstruct the information that was discarded (you might like to try convincing yourself of this: can you deduce what values must go in the greyed-out nodes in Figure 10, without peeking at any previous figures?). However, in and of itself, this observation does not give us a nice algorithm for computing range sums.

It turns out the key is to think about *prefix sums*. As we saw in the introduction and the implementation of `range` in Figure 6, if we can compute the prefix sum $P_k = a_1 + \cdots + a_k$ for any k, then we can compute the range sum $a_i + \cdots + a_j$ as $P_j - P_{i-1}$.

Theorem 1. *Given a thinned segment tree, the sum of* any prefix *of the original array (and hence also any range sum) can be computed, in logarithmic time, using only the values of active nodes.*

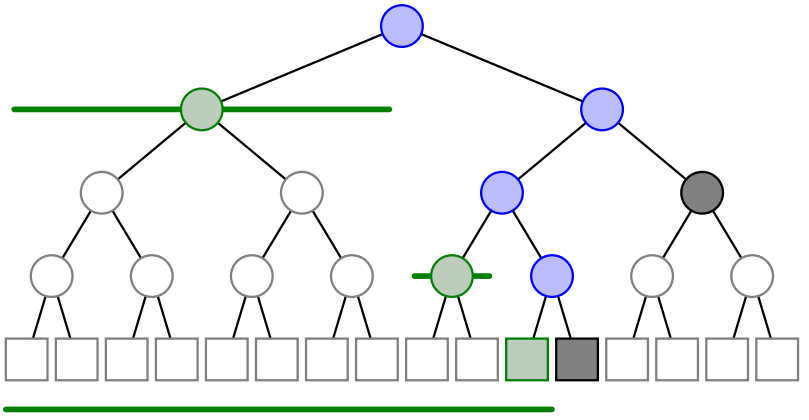


Fig. 11. Performing a prefix query on a segment tree

Proof Surprisingly, in the special case of prefix queries, the original range query algorithm described in Section 1 and implemented in Figure 8 works unchanged! That is to say, the base case in which the range of the current node is wholly contained within the query range—and we thus return the value of the current node—will only ever happen at active nodes.

First, the root itself is active, and hence querying the full range will work. Next, consider the case where we are at a node and recurse on both children. The left child is always active, so we only need to consider the case where we recurse to the right. It is impossible that the range of the right child will be wholly contained in the query range: since the query range is always a prefix of the form $[1, j]$, if the right child's range is wholly contained in $[1, j]$ then the left child's range must be as well—which means that the parent node's range (which is the union of its children's ranges) would also be wholly contained in the query range. But in that case we would simply return the parent's value without recursing into the right child. Thus, when we do recurse into a right child, we might end up returning 0, or we might recurse further into both grandchildren, but in any case we will never try to look at the value of the right child itself. ■

Figure 11 illustrates performing a prefix query on a segment tree. Notice that visited right children are only ever blue or grey; the only green nodes are left children.

3 Fenwick trees

How should we actually store a thinned segment tree in memory? If we stare at Figure 10 again, one strategy suggests itself: simply take every active node and "slide" it down and to the right until it lands in an empty slot in the underlying array, as illustrated in Figure 12. This sets up a one-to-one correspondence between active nodes and indices in the range $1 \ldots n$. Another way to understand this indexing scheme is to use a postorder traversal of the tree, skipping over inactive nodes and giving consecutive indices to active nodes encountered during the traversal. We can also visualize the result by drawing the tree in a

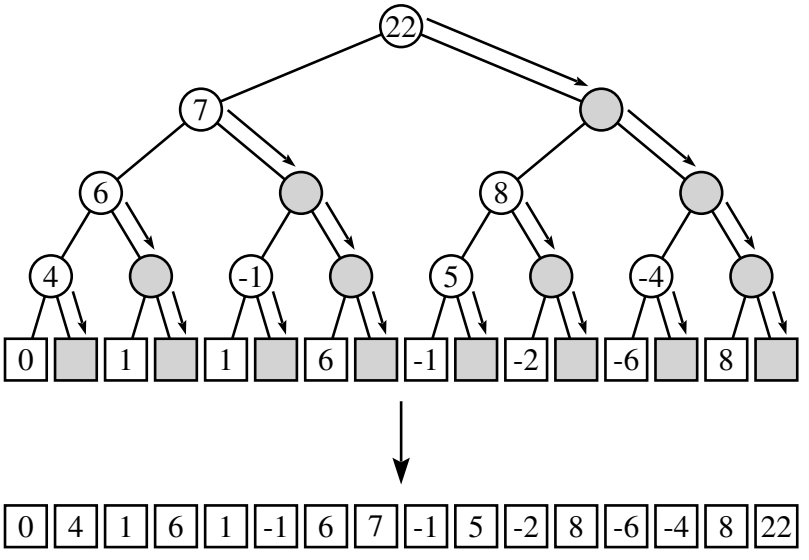


Fig. 12. Sliding active values down a thinned segment tree

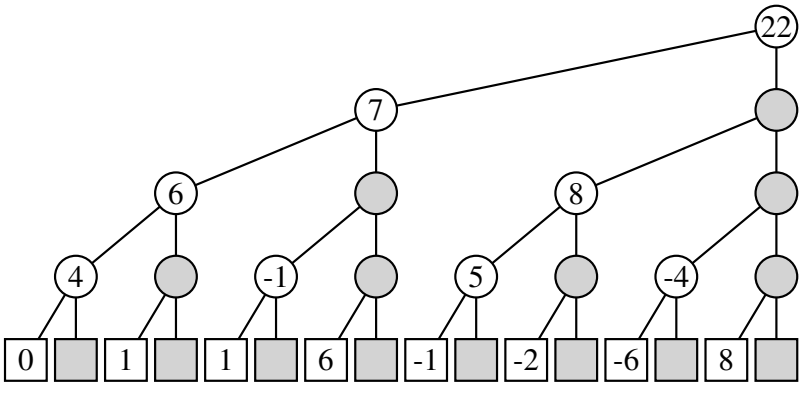


Fig. 13. Right-leaning drawing of a thinned segment tree, vertically aligning nodes with their storage location

"right-leaning" style (Figure 13), vertically aligning each active node with the array slot where it is stored.

This method of storing the active nodes from a thinned segment tree in an array is precisely a *Fenwick tree*. I will also sometimes refer to it as a *Fenwick array*, when I want to particularly emphasize the underlying array data structure. Although it is certainly a clever use of space, the big question is how to implement the update and range query operations. Our implementations of these operations for segment trees worked by recursively descending through the tree, either directly if the tree is stored as a recursive data structure, or using simple operations on indices if the tree is stored in an array. However, when storing the active nodes of a thinned tree in a Fenwick array, it is not *a priori* obvious what operations on array indices will correspond to moving around the tree. In order to attack this problem,

we first take a detour through a domain-specific language for two's complement binary values.

4 Two's Complement Binary

The bit tricks usually employed to implement Fenwick trees rely on a *two's complement* representation of binary numbers, which allow positive and negative numbers to be represented in a uniform way; for example, a value consisting of all 1 bits represents −1. We therefore turn now to developing a domain-specific language, embedded in Haskell, for manipulating two's complement binary representations.

First, we define a type of bits, with functions for inversion, logical conjunction, and logical disjunction:

```
data Bit = O | I deriving (Eq, Ord, Show, Enum)

¬ :: Bit → Bit
¬ = λcase {O → I; I → O}

(∧), (∨) :: Bit → Bit → Bit
O ∧ _ = O
I ∧ b = b

I ∨ _ = I
O ∨ b = b
```

Next, we must define bit strings, *i.e.* sequences of bits. Rather than fix a specific bit width, it will be much more elegant to work with *infinite* bit strings.[3] It is tempting to use standard Haskell lists to represent potentially infinite bit strings, but this leads to a number of problems. For example, equality of infinite lists is not decidable, and there is no way in general to convert from an infinite list of bits back to an *Integer*—how would we know when to stop? In fact, these practical problems stem from a more fundamental one: infinite lists of bits are actually a bad representation for two's complement bit strings, because of "junk", that is, infinite lists of bits which do not correspond to values in our intended semantic domain. For example, *cycle* [*I*, *O*] is an infinite list which alternates between *I* and *O* forever, but it does not represent a valid two's complement encoding of an integer. Even worse are non-periodic lists, such as the one with *I* at every prime index and *O* everywhere else.

In fact, the bit strings we want are the *eventually constant* ones, that is, strings which eventually settle down to an infinite tail of all zeros (which represent nonnegative integers) or all ones (which represent negative integers). Every such string has a finite representation, so directly encoding eventually constant bit strings in Haskell not only gets rid of the junk but also leads to elegant, terminating algorithms for working with them.

```
data Bits where
  Rep  :: Bit → Bits
  Snoc :: !Bits → Bit → Bits
```

[3] Some readers may recognize infinite two's complement bit strings as 2-*adic* numbers, that is, p-adic numbers for the specific case $p = 2$, but nothing in our story depends on understanding the connection.

Rep b represents an infinite sequence of bit *b*, whereas *Snoc bs b* represents the bit string *bs* followed by a final bit *b*. We use *Snoc*, rather than *Cons*, to match the way we usually write bit strings, with the least significant bit last. Note also the use of a *strictness annotation* on the *Bits* field of *Snoc*; this is to rule out infinite lists of bits using only *Snoc*, such as $bs = Snoc\ (Snoc\ bs\ O)\ I$. In other words, the only way to make a non-bottom value of type *Bits* is to have a finite sequence of *Snoc* finally terminated by *Rep*.

Although we have eliminated junk values, one remaining problem is that there can be multiple distinct representations of the same value. For example, $Snoc\ (Rep\ O)\ O$ and $Rep\ O$ both represent the infinite bit string containing all zeros. However, we can solve this with a carefully constructed *bidirectional pattern synonym* (Pickering et al., 2016).

```
toSnoc :: Bits → Bits
toSnoc (Rep a) = Snoc (Rep a) a
toSnoc as = as

pattern (:.) :: Bits → Bit → Bits
pattern (:.) bs b ← (toSnoc → Snoc bs b)
  where
    Rep b :. b' | b ≡ b' = Rep b
    bs :. b = Snoc bs b

{-# COMPLETE (:.) #-}
```

Matching with the pattern $(bs :. b)$ uses a *view pattern* (Erwig and Jones, 2001) to potentially expand a *Rep* one step into a *Snoc*, so that we can pretend *Bits* values are always constructed with (:.). Conversely, constructing a *Bits* with (:.) will do nothing if we happen to snoc an identical bit *b* onto an existing *Rep b*. This ensures that as long as we stick to using (:.) and never directly use *Snoc*, *Bits* values will always be *normalized* so that the terminal *Rep b* is immediately followed by a different bit. Finally, we mark the pattern (:.) as `COMPLETE` on its own, since matching on (:.) is indeed sufficient to handle every possible input of type *Bits*. However, in order to obtain terminating algorithms we will often include one or more special cases for *Rep*.

Let's begin with some functions for converting *Bits* to and from *Integer*, and for displaying *Bits* (intended only for testing).

```
toBits :: Int → Bits
toBits n
  | n ≡ 0 = Rep O
  | n ≡ -1 = Rep I
  | otherwise = toBits (n `div` 2) :. toEnum (n `mod` 2)

fromBits :: Bits → Int
fromBits (Rep O) = 0
fromBits (Rep I) = -1
fromBits (bs :. b) = 2 · fromBits bs + fromEnum b

instance Show Bits where
  show = reverse ∘ go
    where
```

$$
\begin{aligned}
&\mathit{go}\ (\mathit{Rep}\ b) = \mathit{replicate}\ 3\ (\mathit{showBit}\ b)\ \texttt{++"..."}\\
&\mathit{go}\ (\mathit{bs} :.\ b) = \mathit{showBit}\ b : \mathit{go}\ \mathit{bs}\\
&\mathit{showBit} = (\texttt{"01"!!}) \circ \mathit{fromEnum}
\end{aligned}
$$

Let's try it out, using QuickCheck (Claessen and Hughes, 2000) to verify our conversion functions:

```
ghci> Rep O :. O :. I :. O :. I
...00101
ghci> Rep I :. O :. I
...11101
ghci> toBits 26
...00011010
ghci> toBits (-30)
...11100010
ghci> fromBits (toBits (-30))
-30
ghci> quickCheck $ \x -> fromBits (toBits x) == x
+++ OK, passed 100 tests.
```

We can now begin implementing some basic operations on *Bits*. First, incrementing and decrementing can be implemented recursively as follows:

$$
\begin{aligned}
&\mathit{inc} :: \mathit{Bits} \to \mathit{Bits}\\
&\mathit{inc}\ (\mathit{Rep}\ I) \quad = \mathit{Rep}\ O\\
&\mathit{inc}\ (\mathit{bs} :.\ O) \ = \mathit{bs} :.\ I\\
&\mathit{inc}\ (\mathit{bs} :.\ I) \quad = \mathit{inc}\ \mathit{bs} :.\ O
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dec} :: \mathit{Bits} \to \mathit{Bits}\\
&\mathit{dec}\ (\mathit{Rep}\ O) = \mathit{Rep}\ I\\
&\mathit{dec}\ (\mathit{bs} :.\ I) \ = \mathit{bs} :.\ O\\
&\mathit{dec}\ (\mathit{bs} :.\ O) = \mathit{dec}\ \mathit{bs} :.\ I
\end{aligned}
$$

The *least significant bit*, or LSB, of a sequence of bits can be defined as follows:

$$
\begin{aligned}
&\mathit{lsb} :: \mathit{Bits} \to \mathit{Bits}\\
&\mathit{lsb}\ (\mathit{Rep}\ O) = \mathit{Rep}\ O\\
&\mathit{lsb}\ (\mathit{bs} :.\ O) = \mathit{lsb}\ \mathit{bs} :.\ O\\
&\mathit{lsb}\ (\_ :.\ I) \quad = \mathit{Rep}\ O :.\ I
\end{aligned}
$$

Note that we add a special case for *Rep O* to ensure that *lsb* is total. Technically, *Rep O* does not have a least significant bit, so defining $\mathit{lsb}\ (\mathit{Rep}\ O) = \mathit{Rep}\ O$ seems sensible.

```
ghci> toBits 26
"...00011010"
ghci> lsb $ toBits 26
"...00010"
ghci> toBits 24
```

```
"...00011000"
ghci> lsb $ toBits 24
"...0001000"
```

Bitwise logical conjunction can be defined straightforwardly. Note that we only need two cases; if the finite parts of the inputs have different lengths, matching with (:.) will automatically expand the shorter one to match the longer one.

$$
\begin{aligned}
&(\owedge) :: \mathit{Bits} \to \mathit{Bits} \to \mathit{Bits} \\
&\mathit{Rep}\ x \owedge \mathit{Rep}\ y = \mathit{Rep}\ (x \wedge y) \\
&(xs :.\ x) \owedge (ys :.\ y) = (xs \owedge ys) :.\ (x \wedge y)
\end{aligned}
$$

Bitwise inversion is likewise straightforward.

$$
\begin{aligned}
&\mathit{inv} :: \mathit{Bits} \to \mathit{Bits} \\
&\mathit{inv}\ (\mathit{Rep}\ b) = \mathit{Rep}\ (\neg\, b) \\
&\mathit{inv}\ (bs :.\ b) = \mathit{inv}\ bs :.\ \neg\, b
\end{aligned}
$$

The above functions follow familiar patterns. We could easily generalize to eventually constant streams over an arbitrary element type, and then implement $(\owedge)$ in terms of a generic *zipWith* and *inv* in terms of *map*. However, for the present purpose we do not need the extra generality.

We implement addition with the usual carry-propagation algorithm, along with some special cases for *Rep*.

$$
\begin{aligned}
&(\oplus) :: \mathit{Bits} \to \mathit{Bits} \to \mathit{Bits} \\
&xs \quad\ \oplus \mathit{Rep}\ O \quad\ = xs \\
&\mathit{Rep}\ O \quad \oplus ys \quad\ = ys \\
&\mathit{Rep}\ I \quad \oplus \mathit{Rep}\ I \quad = \mathit{Rep}\ I :.\ O \\
&\mathit{Snoc}\ xs\ I \oplus \mathit{Snoc}\ ys\ I = \mathit{inc}\ (xs \oplus ys) :.\ O \\
&\mathit{Snoc}\ xs\ x \oplus \mathit{Snoc}\ ys\ y = (xs \oplus ys) :.\ (x \vee y)
\end{aligned}
$$

It is not too hard to convince ourselves that this definition of addition is terminating and yields correct results; but we can also be fairly confident by just trying it with QuickCheck:

```
ghci> quickCheck $ \x y -> fromBits (toBits x .+. toBits y) == x + y
+++ OK, passed 100 tests.
```

Finally, the following definition of negation is probably familiar to anyone who has studied two's complement arithmetic; I leave it as an exercise for the interested reader to prove that $x \oplus \mathit{neg}\ x \equiv \mathit{Rep}\ O$ for all $x :: \mathit{Bits}$.

$$
\begin{aligned}
&\mathit{neg} :: \mathit{Bits} \to \mathit{Bits} \\
&\mathit{neg} = \mathit{inc} \circ \mathit{inv}
\end{aligned}
$$

We now have the tools to resolve the first mystery of the Fenwick tree implementation.

Theorem 4.1. *For all $x :: \mathit{Bits}$,*

$$\mathit{lsb}\ x = x \owedge \mathit{neg}\ x.$$

Proof By induction on x.

- First, if $x = Rep\ O$, it is an easy calculation to verify that $lsb\ x = x \oslash neg\ x = Rep\ O$.
- Likewise, if $x = Rep\ I$, both $lsb\ x$ and $x \oslash neg\ x$ reduce to $Rep\ O :.\ I$.
- If $x = xs :.\ O$, then $lsb\ x = lsb\ (xs :.\ O) = lsb\ xs :.\ O$ by definition, whereas

$$
\begin{array}{ll}
 & (xs :.\ O) \oslash neg\ (xs :.\ O) \\
= & \quad \{ \quad \text{Definition of } neg \quad \} \\
 & (xs :.\ O) \oslash inc\ (inv\ (xs :.\ O)) \\
= & \quad \{ \quad \text{Definition of } inv \text{ and } \neg \quad \} \\
 & (xs :.\ O) \oslash inc\ (inv\ xs :.\ I) \\
= & \quad \{ \quad \text{Definition of } inc \quad \} \\
 & (xs :.\ O) \oslash (inc\ (inv\ xs) :.\ O) \\
= & \quad \{ \quad \text{Definition of } \oslash \text{ and } neg \quad \} \\
 & (xs \oslash neg\ xs) :.\ O \\
= & \quad \{ \quad \text{Induction hypothesis} \quad \} \\
 & lsb\ xs :.\ O
\end{array}
$$

- Next, if $x = xs :.\ I$, then $lsb\ (xs :.\ I) = Rep\ O :.\ I$ by definition, whereas

$$
\begin{array}{ll}
 & (xs :.\ I) \oslash neg\ (xs :.\ I) \\
= & \quad \{ \quad \text{Definition of } neg \quad \} \\
 & (xs :.\ I) \oslash inc\ (inv\ (xs :.\ I)) \\
= & \quad \{ \quad \text{Definition of } inv \text{ and } \neg \quad \} \\
 & (xs :.\ I) \oslash inc\ (inv\ xs :.\ O)) \\
= & \quad \{ \quad \text{Definition of } inc \quad \} \\
 & (xs :.\ I) \oslash (inv\ xs :.\ I) \\
= & \quad \{ \quad \text{Definition of } \oslash \quad \} \\
 & (xs \oslash inv\ xs) :.\ I \\
= & \quad \{ \quad \text{Bitwise AND of } xs \text{ and its inverse is } Rep\ O \quad \} \\
 & Rep\ O :.\ I
\end{array}
$$

■

For the last equality we need a lemma that $xs \oslash inv\ xs = Rep\ O$, which should be intuitively clear and can easily be proved by induction as well.

Finally, in order to express the index conversion functions we will develop in the next section, we need a few more things in our DSL. First, some functions to set and clear individual bits, and to test whether particular bits are set:

$$
\begin{array}{l}
setTo :: Bit \to Int \to Bits \to Bits \\
setTo\ b'\ 0\ (bs :.\ \_) = bs :.\ b' \\
setTo\ b'\ k\ (bs :.\ b) = setTo\ b'\ (k-1)\ bs :.\ b \\
\\
set, clear :: Int \to Bits \to Bits \\
set = setTo\ I \\
clear = setTo\ O \\
\\
test :: Int \to Bits \to Bool
\end{array}
$$

```
test 0 (bs :. b) = b ≡ I
test n (bs :. _) = test (n − 1) bs

even, odd :: Bits → Bool
odd = test 0
even = not ∘ odd
```

The only other things we will need are left and right shift, and a generic *while* combinator that iterates a given function, returning the first iterate for which a predicate is false.

```
shr :: Bits → Bits
shr (bs :. _) = bs

shl :: Bits → Bits
shl = (:. O)

while :: (a → Bool) → (a → a) → a → a
while p f x
  | p x          = while p f (f x)
  | otherwise = x
```

5 Index Conversion

Before deriving our index conversion functions we must deal with one slightly awkward fact. In a traditional binary tree indexing scheme, as shown in Figure 9, the root has index 1, every left child is twice its parent, and every right child is one more than twice its parent. Recall that in a thinned segment tree, the root node and every left child are active, with all right children being inactive. This makes the root an awkward special case—all active nodes have an even index, *except* the root, which has index 1. This makes it more difficult to check whether we are at an active node—it is not enough to simply look at the least significant bit.

One easy way to fix this is simply to give the root index 2, and then proceed to label the rest of the nodes using the same scheme—every left child is twice its parent, and every right child is one more than twice its parent. This results in the indexing shown in Figure 14, as if we had just taken the left subtree of the tree rooted at 1, and ignored the right subtree. Of course, this means about half the possible indices are omitted—but that's not a problem, since we will only use these indices as an intermediate step which will eventually get fused away.

Figure 15 shows a binary tree where nodes have been numbered in two different ways: the left side of each node shows the node's binary tree index (with the root having index 2). The right side of each node shows its index in the Fenwick array, if it has one (inactive nodes simply have their right half greyed out). The table underneath shows the mapping from Fenwick array indices (top row) to binary tree indices (bottom row). As a larger example, Figure 16 shows the same thing on a binary tree one level deeper.

Our goal is to come up with a way to calculate the binary index for a given Fenwick index or vice versa. Staring at the table in Figure 16, a few patterns stand out. Of course, all the numbers in the bottom row are even, which is precisely because the binary tree is

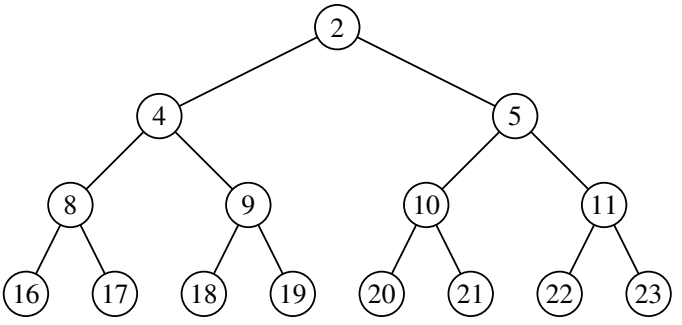
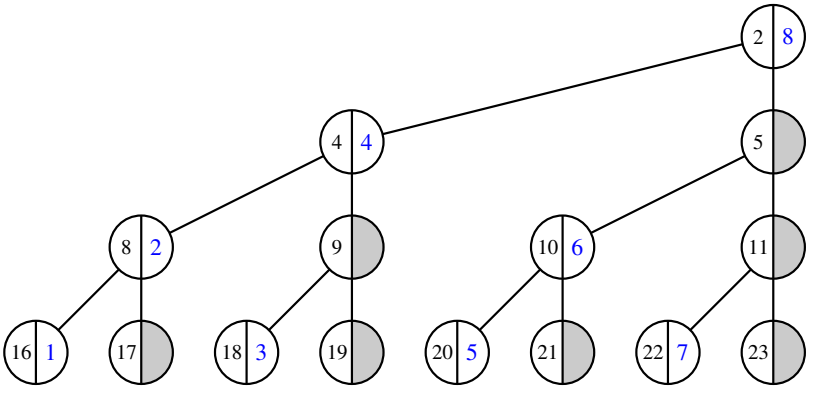
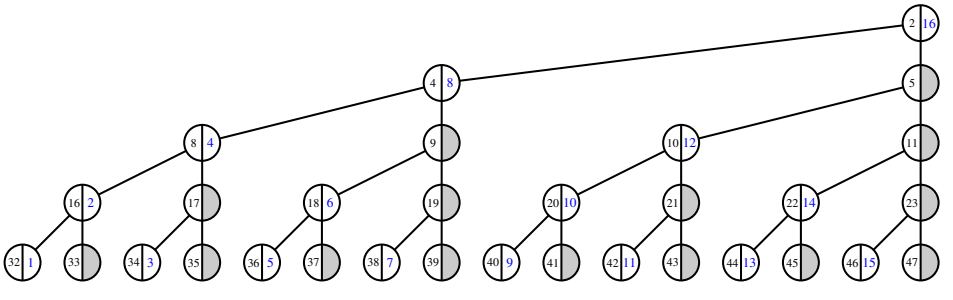


Fig. 14. Indexing a binary tree with 2 at the root



1	2	3	4	5	6	7	8
16	8	18	4	20	10	22	2

Fig. 15. Binary tree labelled with both binary and Fenwick indexing



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
32	16	34	8	36	18	38	4	40	20	42	10	44	22	46	2

Fig. 16. Binary tree labelled with both binary and Fenwick indexing

$$
\begin{aligned}
&(\curlyvee) :: [a] \to [a] \to [a] \\
&[\,] \curlyvee \_ = [\,] \\
&(x : xs) \curlyvee ys = x : (ys \curlyvee xs) \\
&b :: Int \to [Int] \\
&b\ 0 = [2] \\
&b\ n = map\ (2\cdot)\ [2^n \,..\, 2^n + 2^{n-1} - 1] \curlyvee b\ (n-1)
\end{aligned}
$$

Fig. 17. Recurrence for sequence of binary tree indices in a Fenwick array

$$
\begin{aligned}
&(a : \_)\ !\ 1 = a \\
&(\_ : as)\ !\ k = as\ !\ (k-1) \\
&\\
&\text{-- If } |xs| \equiv |ys|: \\
&(xs \curlyvee ys)\ !\ (2 \cdot j) \quad\ = ys\ !\ j \\
&(xs \curlyvee ys)\ !\ (2 \cdot j - 1) = xs\ !\ j
\end{aligned}
$$

Fig. 18. Indexing and interleaving

numbered in such a way that all active nodes have an even index. Second, we can see the even numbers 32, 34 . . . 46, in order, in all the odd positions. These are exactly the leaves of the tree, and indeed, every other node in the Fenwick array will be a leaf from the original tree. Alternating with these, in the even positions, are the numbers 16 8 18 4 . . . , which correspond to all the non-leaf nodes; but these are exactly the sequence of binary indices from the bottom row of the table in Figure 15—since the internal nodes in a tree of height 4 themselves constitute a tree of height 3, with the nodes occurring in the same order.

These observations lead to the recurrence shown in Figure 17 for the sequence b_n of binary indices for the nodes stored in a Fenwick array of length 2^n: b_0 is just the singleton sequence [2], and otherwise b_n is the even numbers $2^{n+1}, 2^{n+1} + 2, \ldots, 2^{n+1} + 2^n - 2$ interleaved with b_{n-1}.

We can check that this does in fact reproduce the observed sequence for $n = 4$:

```
ghci> b 4
[32,16,34,8,36,18,38,4,40,20,42,10,44,22,46,2]
```

Let $s\,!\,k$ denote the kth item in the list s (counting from 1), as defined in Figure 18. The same figure also lists two easy lemmas about the interaction between indexing and interleaving, namely, $(xs \curlyvee ys)\,!\,(2 \cdot j) = ys\,!\,j$ and $(xs \curlyvee ys)\,!\,(2 \cdot j - 1) = xs\,!\,j$ (as long as xs and ys have equal lengths). With these in hand, we can define the Fenwick to binary index conversion function as

$$f2b\ n\ k = b\ n\ !\ k.$$

Of course, since b_n is of length 2^n, this function is only defined on the range $[1, 2^n]$.

We can now simplify the definition of *f2b* as follows. First of all, for even inputs, we have

$$
\begin{aligned}
& f2b\ n\ (2 \cdot j) \\
= & \qquad \{ \quad \text{Definition of } f2b \quad \} \\
& b\ n\ !\ (2 \cdot j) \\
= & \qquad \{ \quad \text{Definition of } b \quad \} \\
& (map\ (2\cdot)\ [2^n\ ..\ 2^n + 2^{n-1} - 1] \curlyvee b\ (n-1))\ !\ (2 \cdot j) \\
= & \qquad \{ \quad \curlyvee\text{–!} \text{ lemma} \quad \} \\
& b(n-1)!j \\
= & \qquad \{ \quad \text{Definition of } f2b \quad \} \\
& f2b\ (n-1)\ j.
\end{aligned}
$$

Whereas for odd inputs,

$$
\begin{aligned}
& f2b\ n\ (2 \cdot j - 1) \\
= & \qquad \{ \quad \text{Definition of } f2b \quad \} \\
& b\ n\ !\ (2 \cdot j - 1) \\
= & \qquad \{ \quad \text{Definition of } b \quad \} \\
& (map\ (2\cdot)\ [2^n\ ..\ 2^n + 2^{n-1} - 1] \curlyvee b\ (n-1))\ !\ (2 \cdot j - 1) \\
= & \qquad \{ \quad \curlyvee\text{–!} \text{ lemma} \quad \} \\
& map\ (2\cdot)\ [2^n\ ..\ 2^n + 2^{n-1} - 1]\ !\ j \\
= & \qquad \{ \quad \text{Definition of } map\text{, algebra} \quad \} \\
& 2 \cdot (2^n + j - 1) \\
= & \qquad \{ \quad \text{algebra} \quad \} \\
& 2^{n+1} + 2j - 2
\end{aligned}
$$

Thus we have

$$
f2b\ n\ k = \begin{cases} f2b\ (n-1)\ (k\ /\ 2) & k \text{ even} \\ 2^{n+1} + k - 1 & k \text{ odd} \end{cases}
$$

Note that when $n = 0$ we must have $k = 1$, and hence $f2b\ 0\ 1 = 2^0 + 1 - 1 = 1$, as required, so this definition is valid for all $n \geqslant 0$. Now factor k uniquely as $2^a \cdot b$ where b is odd. Then by induction we can see that

$$
f2b\ n\ (2^a \cdot b) = f2b\ (n-a)\ b = 2^{n-a+1} + b - 1.
$$

So, in other words, computing *f2b* consists of repeatedly dividing by 2 (*i.e.* right bit shifts) as long as the input is even, and then finally decrementing and adding a power of 2. However, knowing what power of 2 to add at the end depends on knowing how many times we shifted. A better way to think of it is to add 2^{n+1} at the *beginning*, and then let it be shifted along with everything else. Thus, we have the following definition of *f2b′* using our *Bits* DSL. Defining *shift n* = *while even shr* ∘ *set n* separately will make some of our proofs more compact later.

```
shift :: Int → Bits → Bits
shift n = while even shr ∘ set n
```

$f2b' :: Int \to Bits \to Bits$
$f2b'\ n = dec \circ shift\ (n+1)$

For example, we can verify that this produces identical results to $f2b\,4$ on the range $[1, 2^4]$ (for convenience, we define $(f === g)\ k = f\ k \equiv g\ k$):

```
ghci> all (f2b 4 === fromBits . f2b' 4 . toBits) [1 .. 2^4]
True
```

We now turn to deriving $b2f\ n$, which converts back from binary to Fenwick indices. $b2f\ n$ should be a left inverse to $f2b\ n$, that is, for any $k \in [1, 2^n]$ we should have $b2f\ n\ (f2b\ n\ k) \equiv k$. If k is an input to $f2b$, we have $k = 2^a \cdot b \leqslant 2^n$, and so $b - 1 < 2^{n-a}$. Hence, given the output $f2b\ n\ k = m = 2^{n-a+1} + b - 1$, the highest bit of m is 2^{n-a+1}, and the rest of the bits represent $b - 1$. So, in general, given some m which is the output of $f2b\ n$, we can write it uniquely as $m = 2^c + d$ where $d < 2^{c-1}$; then

$$b2f\ n\ (2^c + d) = 2^{n-c+1} \cdot (d+1).$$

In other words, given the input $2^c + d$, we subtract off the highest bit 2^c, increment, then left shift $n - c + 1$ times. Again, though, there is a simpler way: we can increment first (note since $d < 2^{c-1}$, incrementing cannot disturb the bit at 2^c), then left shift enough times to bring the leftmost bit into position $n + 1$, and finally remove it. That is:

$unshift :: Int \to Bits \to Bits$
$unshift\ n = clear\ n \circ while\ (not \circ test\ n)\ shl$

$b2f' :: Int \to Bits \to Bits$
$b2f'\ n = unshift\ (n+1) \circ inc$

Verifying:

```
ghci> all (fromBits . b2f' 4 . f2b' 4 . toBits === id) [1 .. 2^4]
True
```

6 Deriving Fenwick Operations

We can now finally derive the required operations on Fenwick array indices for moving through the tree, by starting with operations on a binary indexed tree and conjugating by conversion to and from Fenwick indices. First, in order to fuse away the resulting conversion, we will need a few lemmas.

Lemma 6.1 (shr-inc-dec)**.** *For all $bs :: Bits$ which are odd (that is, end with I),*

- $(shr \circ dec)\ bs = shr\ bs$
- $(shr \circ inc)\ bs = (inc \circ shr)\ bs$

Proof Both are immediate by definition. ∎

Lemma 6.2 (while-inc-dec)**.** *The following both hold for all Bits values:*

- $inc \circ while\ odd\ shr = while\ even\ shr \circ inc$
- $dec \circ while\ even\ shr = while\ odd\ shr \circ dec$

Proof Easy proof by induction on *Bits*. For example, for the *inc* case, the functions on both sides discard consecutive 1 bits and then flip the first 0 bit to a 1. ■

Finally, we will need a lemma about shifting zero bits in and out of the right side of a value.

Lemma 6.3 (shl-shr)**.** *For all* $0 < x < 2^{n+2}$,

$$(while\ (not \circ test\ (n+1))\ shl \circ while\ even\ shr)\ x = while\ (not \circ test\ (n+1))\ shl\ x.$$

Proof Intuitively, this says that if we first shift out all the zero bits and then left shift until bit $n+1$ is set, we could get the same result by forgetting about the right shifts entirely; shifting out zero bits and then shifting them back in should be the identity.

Formally, the proof is by induction on x. If $x = xs :. I$ is odd, the equality is immediate since $while\ even\ shr\ x = x$. Otherwise, if $x = xs :. O$, on the left-hand side the O is immediately discarded by *shr*, whereas on the right-hand side $xs :. O = shl\ xs$, and the extra *shl* can be absorbed into the *while* since $xs < 2^{n+1}$. What remains is simply the induction hypothesis. ■

With these lemmas under our belt, let's see how to move around a Fenwick array in order to implement *update* and *query*; we'll begin with *update*. When implementing the *update* operation, we need to start at a leaf and follow the path up to the root, updating all the active nodes along the way. In fact, for any given leaf, its closest active parent is precisely the node stored in the slot that used to correspond to that leaf (see Figure 13). So to update index i, we just need to start at index i in the Fenwick array, and then repeatedly find the closest active parent, updating as we go. Recall that the imperative code for *update* works this way, apparently finding the closest active parent at each step by adding the LSB of the current index:

```
public void update(int i, long delta) {
    for (; i < a.length; i += LSB(i)) a[i] += delta;
}
```

Let's see how to derive this behavior.

To find the closest active parent of a node under a binary indexing scheme, we first move up to the immediate parent (by dividing the index by two, *i.e.* performing a right bit shift); then continue moving up to the next immediate parent as long as the current node is a right child (*i.e.* has an odd index). This yields the definition:

$activeParentBinary :: Bits \to Bits$
$activeParentBinary = while\ odd\ shr \circ shr$

This is why we used the slightly strange indexing scheme with the root having index 2—otherwise this definition would not work for any node whose active parent is the root!

Now, to derive the corresponding operation on Fenwick indices, we conjugate by conversion to and from Fenwick indices, and compute as follows. To make the computation easier to read, the portion being rewritten is underlined at each step.

$$
\begin{aligned}
& b2f'\ n \circ activeParentBinary \circ f2b'\ n \\
= & \quad \{ \text{ expand definitions } \} \\
& unshift\ (n+1) \circ \underline{inc \circ while\ odd\ shr} \circ shr \circ dec \circ shift\ (n+1) \\
= & \quad \{ \text{ Lemma 6.2 (while-inc-dec) } \} \\
& unshift\ (n+1) \circ while\ even\ shr \circ inc \circ \underline{shr \circ dec} \circ shift\ (n+1) \\
= & \quad \{ \text{ Lemma 6.1 (shr-inc-dec); } shift\ (n+1)\ x \text{ is always odd } \} \\
& unshift\ (n+1) \circ while\ even\ shr \circ \underline{inc \circ shr} \circ shift\ (n+1) \\
= & \quad \{ \text{ Lemma 6.1 (shr-inc-dec) } \} \\
& unshift\ (n+1) \circ \underline{while\ even\ shr \circ shr} \circ inc \circ shift\ (n+1) \\
= & \quad \{ \text{ } while\ even\ shr \circ shr = while\ even\ shr \text{ on an even input } \} \\
& \underline{unshift\ (n+1)} \circ while\ even\ shr \circ inc \circ shift\ (n+1) \\
= & \quad \{ \text{ Definition of } unshift \} \\
& clear\ (n+1) \circ \underline{while\ (not \circ test\ (n+1))\ shl \circ while\ even\ shr} \circ inc \circ \underline{shift\ (n+1)} \\
= & \quad \{ \text{ Lemma 6.3 (shl-shr); definition of } shift \} \\
& clear\ (n+1) \circ while\ (not \circ test\ (n+1))\ shl \circ inc \circ while\ even\ shr \circ set\ (n+1)
\end{aligned}
$$

In the final step, since the input x satisfies $x \leqslant 2^n$, we have $inc \circ shift\ (n+1) < 2^{n+2}$, so Lemma 6.3 applies.

Reading from right to left, the pipeline we have just computed performs the following steps:

1. Set bit $n+1$
2. Shift out consecutive zeros until finding the least significant 1 bit
3. Increment
4. Shift zeros back in to bring the most significant bit back to position $n+1$, then clear it.

Intuitively, this does look a lot like adding the LSB! In general, to find the LSB, one must shift through consecutive 0 bits until finding the first 1; the question is how to keep track of how many 0 bits were shifted on the way. The *lsb* function itself keeps track via the recursion stack; after finding the first 1 bit, the recursion stack unwinds and re-snocs all the 0 bits recursed through on the way. The above pipeline represents an alternative approach: set bit $n+1$ as a "sentinel" to keep track of how much we have shifted; right shift until the first 1 is literally in the ones place, at which point we increment; and then shift all the 0 bits back in by doing left shifts until the sentinel bit gets back to the $n+1$ place. One example of this process is illustrated in Figure 19. Of course, this only works for values that are sufficiently small that the sentinel bit will not be disturbed throughout the operation.

To make this more formal, we begin by defining a helper function *atLSB*, which does an operation "at the LSB", that is, it shifts out 0 bits until finding a 1, applies the given function, then restores the 0 bits.

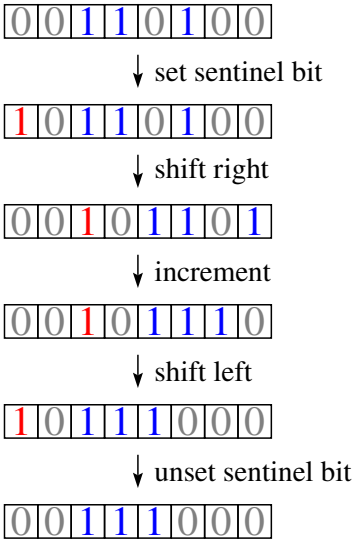


Fig. 19. Adding LSB with a sentinel bit + shifts

$$
\begin{aligned}
&\mathit{atLSB} :: (\mathit{Bits} \to \mathit{Bits}) \to \mathit{Bits} \to \mathit{Bits}\\
&\mathit{atLSB}\ \_\ (\mathit{Rep}\ O) = \mathit{Rep}\ O\\
&\mathit{atLSB}\ f\ (bs :. O) = \mathit{atLSB}\ f\ bs :. O\\
&\mathit{atLSB}\ f\ bs = f\ bs
\end{aligned}
$$

Lemma 6.4 (add-lsb)**.** *For all* $x :: \mathit{Bits}$, $x + \mathit{lsb}\ x = \mathit{atLSB}\ \mathit{inc}\ x$ *and* $x - \mathit{lsb}\ x = \mathit{atLSB}\ \mathit{dec}\ x$.

Proof Straightforward induction on x. ■

We can formally relate the "shifting with a sentinel" scheme to the use of *atLSB*, with the following (admittedly rather technical) lemma:

Lemma 6.5 (sentinel)**.** *Let* $n \geqslant 1$ *and let* $f :: \mathit{Bits} \to \mathit{Bits}$ *be a function such that*

1. $(f \circ \mathit{set}\ (n+1))\ x = (\mathit{set}\ (n+1) \circ f)\ x$ *for any* $0 < x < 2^n$, *and*
2. $f\ x < 2^{n+1}$ *for any* $0 < x < 2^n + 2^{n-1}$.

Then for all $0 < x < 2^n$,

$$(\mathit{unshift}\ (n+1) \circ f \circ \mathit{shift}\ (n+1))\ x = \mathit{atLSB}\ f\ x.$$

The proof is rather tedious and not all that illuminating, so we omit it (an extended version including a full proof may be found on the author's website, at `http://ozark.hendrix.edu/~yorgey/pub/Fenwick-ext.pdf`). However, we do note that both *inc* and *dec* fit the criteria for f: incrementing or decrementing some $0 < x < 2^n$ cannot affect the $(n+1)$st bit as long as $n \geqslant 1$, and the result of incrementing or decrementing a number less than

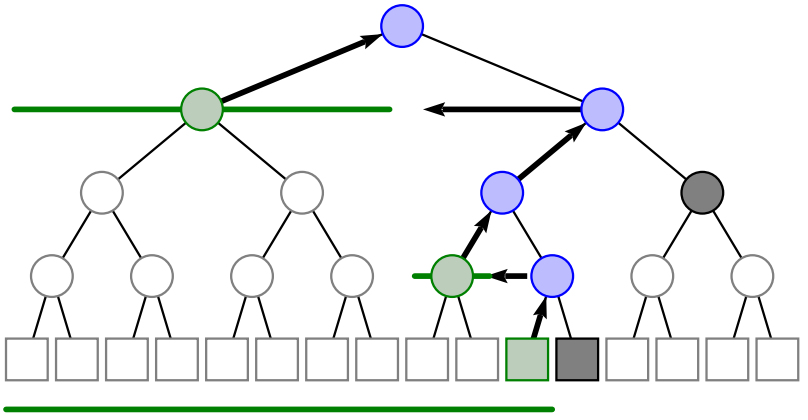


Fig. 20. Moving up a segment tree to find successive prefix segments

$2^n + 2^{n-1}$ will be a number less than 2^{n+1}. We can now put all the pieces together show that adding the LSB at each step is the correct way to implement *update*.

Theorem 6.6. *Adding the LSB is the correct way to move up a Fenwick-indexed tree to the nearest active parent, that is,*

$$activeParentFenwick = b2f'\ n \circ activeParentBinary \circ f2b'\ n = \lambda x \longrightarrow x + lsb\ x$$

everywhere on the range $[1, 2^n)$. *(We exclude* 2^n *since it corresponds to the root of the tree under a Fenwick indexing scheme.)*

Proof

$$\begin{aligned}
& b2f'\ n \circ activeParentBinary \circ f2b'\ n \\
= & \quad \{ \text{ Previous calculation } \} \\
& unshift\ (n+1) \circ inc \circ shift\ (n+1) \\
= & \quad \{ \text{ Lemma 6.5 (sentinel) } \} \\
& atLSB\ inc \\
= & \quad \{ \text{ Lemma 6.4 (add-lsb) } \} \\
& \lambda x \longrightarrow x + lsb\ x
\end{aligned}$$

■

We can carry out a similar process to derive an implementation for prefix query (which suppsedly involves *subtracting* the LSB). Again, if we want to compute the sum of $[1, j]$, we can start at index j in the Fenwick array, which stores the sum of the unique segment ending at j. If the node at index j stores the segment $[i, j]$, we next need to find the unique node storing a segment that ends at $i - 1$. We can do this repeatedly, adding up segments as we go.

Staring at Figure 20 for inspiration, we can see that what we want to do is find the *left sibling* of our *closest inactive parent*, that is, we go up until finding the first ancestor which is a right child, then go to its left sibling. Under a binary indexing scheme, this can be implemented simply as:

$$prevSegmentBinary :: Bits \to Bits$$
$$prevSegmentBinary = dec \circ while\ even\ shr$$

Theorem 6.7. *Subtracting the LSB is the correct way to move up a Fenwick-indexed tree to the to active node covering the segment previous to the current one, that is,*

$$prevSegmentFenwick = b2f'\ n \circ prevSegmentBinary \circ f2b'\ n = \lambda x \to x - lsb\ x$$

everywhere on the range $[1, 2^n)$.

Proof

$$\begin{aligned}
& b2f'\ n \circ prevSegmentBinary \circ f2b'\ n \\
= & \quad \{ \text{ expand definitions } \} \\
& unshift\ (n+1) \circ \underline{inc \circ dec} \circ while\ even\ shr \circ dec \circ shift\ (n+1) \\
= & \quad \{ \ inc \circ dec = id \ \} \\
& \underline{unshift\ (n+1)} \circ while\ even\ shr \circ dec \circ shift\ (n+1) \\
= & \quad \{ \text{ Definition of } unshift \ \} \\
& clear\ (n+1) \circ \underline{while\ (not \circ test\ (n+1))\ shl \circ while\ even\ shr} \circ dec \circ shift\ (n+1) \\
= & \quad \{ \text{ Lemma 6.3 (shl-shr) } \} \\
& \underline{clear\ (n+1) \circ while\ (not \circ test\ (n+1))\ shl} \circ dec \circ shift\ (n+1) \\
= & \quad \{ \text{ Definition of } unshift \ \} \\
& unshift\ (n+1) \circ dec \circ shift\ (n+1) \\
= & \quad \{ \text{ Lemma 6.5 (sentinel) } \} \\
& atLSB\ dec \\
= & \quad \{ \text{ Lemma 6.4 (add-lsb) } \} \\
& \lambda x \to x - lsb\ x
\end{aligned}$$

■

7 Conclusion

Historically, to my knowledge, Fenwick trees were not actually developed as an optimization of segment trees as presented here. This has merely been a fictional—but hopefully illuminating—alternate history of ideas, highlighting the power of functional thinking, domain-specific languages, and equational reasoning to explore relationships between different structures and algorithms. As future work, it would be interesting to explore some of the mentioned generalizations of segment trees, to see whether one can derive Fenwick-like structures that support additional operations.

Acknowledgements

Thanks to the anonymous JFP reviewers for their helpful feedback, which resulted in a much improved presentation. Thanks also to Penn PL Club for the opportunity to present an early version of this work.

Conflicts of Interest. None

References

Apfelmus, H. (2009) Monoids and finger trees. https://apfelmus.nfshost.com/articles/monoid-fingertree.html. [Online; accessed 10-Jun-2024].

Bird, R. & Gibbons, J. (2002) Arithmetic coding with folds and unfolds. In *International School on Advanced Functional Programming*. Springer. pp. 1–26.

Claessen, K. & Hughes, J. (2000) Quickcheck: a lightweight tool for random testing of haskell programs. Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 268–279.

Erwig, M. & Jones, S. P. (2001) Pattern guards and transformational patterns. *Electronic Notes in Theoretical Computer Science*. **41**(1), 3.

Fenwick, P. M. (1994) A new data structure for cumulative frequency tables. *Software: Practice and Experience*. **24**(3), 327–336.

Halim, S., Halim, F. & Effendy, S. (2020) *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*. Lulu Press.

Hinze, R. & Paterson, R. (2006) Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*. **16**(2), 197–217.

Ivanov, M. (2011) Fenwick tree. https://cp-algorithms.com/data_structures/fenwick.html. [Online; accessed 21-Nov-2024].

Ivanov, M. (2011) Segment tree. https://cp-algorithms.com/data_structures/segment_tree.html. [Online; accessed 03-Jun-2024].

Pickering, M., Érdi, G., Peyton Jones, S. & Eisenberg, R. A. (2016) Pattern synonyms. Proceedings of the 9th ACM SIGPLAN International Symposium on Haskell. p. 80–91.

Rissanen, J. & Langdon, G. G. (1979) Arithmetic coding. *IBM Journal of research and development*. **23**(2), 149–162.

Ryabko, B. Y. (1989) A fast on-line code. Doklady Akademii Nauk. Russian Academy of Sciences. pp. 548–552.

Wikipedia contributors. (2024) Segment tree — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Segment_tree. [Online; accessed 03-Jun-2024].