

You Could Have Invented Fenwick Trees!

Brent Yorgey

ICFP, 13 October 2025

Motivation

Sequence operations

a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8

↓ update

a_1 a_2 a_3 v a_5 a_6 a_7 a_8

↓ range query

$$a_2 + a_3 + v + a_5$$

Solutions

approach update range query

Solutions

approach	update	range query
just store sequence	$O(1)$	$O(n)$

Solutions

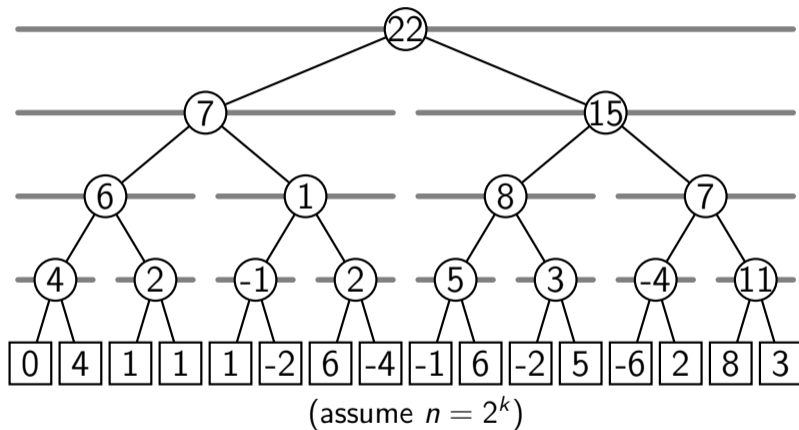
approach	update	range query
just store sequence	$O(1)$	$O(n)$
segment tree	$O(\lg n)$	$O(\lg n)$

$$P[i] = a_1 + \dots + a_i$$
$$RQ(i, j) = P[j] - P[i - 1]$$

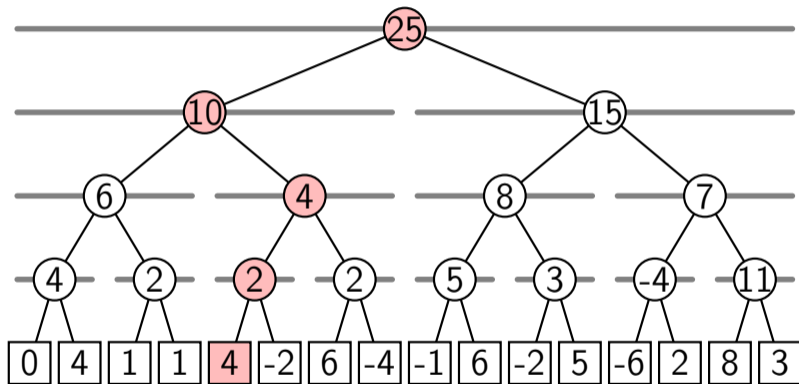
Solutions

approach	update	range query
just store sequence	$O(1)$	$O(n)$
segment tree	$O(\lg n)$	$O(\lg n)$
Fenwick tree	$O(\lg n)$	$O(\lg n)$

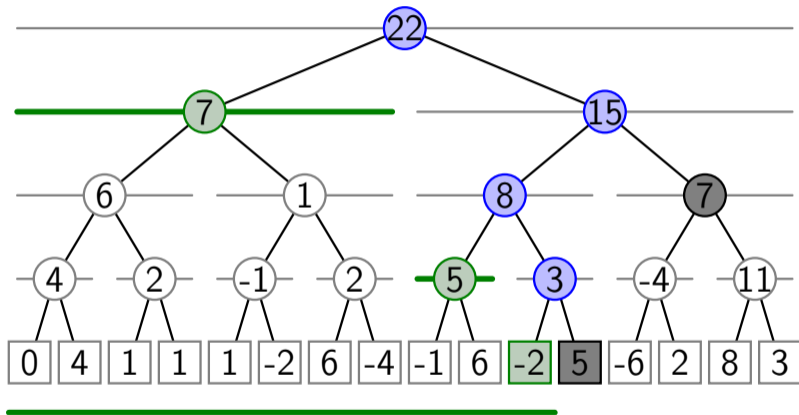
Segment trees



Updating a segment tree



Prefix query on a segment tree



A New Data Structure for Cumulative Frequency Tables

peter m. fenwick

Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand (email: p.fenwick@cs.auckland.ac.nz)

SUMMARY

A new method (the 'binary indexed tree') is presented for maintaining the cumulative frequencies which are needed to support dynamic arithmetic data compression. It is based on a decomposition of the cumulative frequencies into portions which parallel the binary representation of the index of the table element (or symbol). The operations to traverse the data structure are based on the binary coding of the index. In comparison with previous methods, the binary indexed tree is faster, using more compact data and simpler code. The access time for all operations is either constant or proportional to the logarithm of the table size. In conjunction with the compact data structure, this makes the new method particularly suitable for large symbol alphabets.

key words: Binary indexed tree Arithmetic coding Cumulative frequencies

INTRODUCTION

A major cost in adaptive arithmetic data compression is the maintenance of the table of cumulative frequencies which is needed in reducing the range for successive symbols. Witten, Neal and Cleary¹ ease the problem by providing a move-to-front mapping of the symbols which ensures that the most frequent symbols are kept near the front of the search space. It works well for highly skewed alphabets (which may be expected to compress well) but is much less efficient for more uniform distributions of symbol frequency. Moffat² describes a tree structure (actually a heap) which provides a linear-time access to all symbols. Jones³ uses splay trees to provide an optimized data structure for handling the frequency tables. The three techniques will be referred to in this paper as MTF, HEAP and SPLAY, respectively. In all cases they attempt to keep frequently used symbols in quickly-referenced positions within the data structure, but at the cost of sometimes extensive data reorganization.

This current paper describes a new method which uses only a single array to store the frequencies, but stores them in a carefully chosen pattern to suit a novel search technique whose cost is proportional to the number of 1 bits in the element index. This cost applies to both updating and interrogating the table. In comparison with the other methods it is simple, compact and fast and involves no reorganization or movement of the data.

Peter Fenwick, 1994

A New Data Structure for Cumulative Frequency Tables

Fenwick trees

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(3), 327–336 (MARCH 1994)

A New Data Structure for Cumulative Frequency Tables

peter m. fenwick

Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand (email: p.fenwick@cs.auckland.ac.nz)

SUMMARY

A new method (the 'binary indexed tree') is presented for maintaining the cumulative frequencies which are needed to support dynamic arithmetic data compression. It is based on a decomposition of the cumulative frequencies into portions which parallel the binary representation of the index of the table element (or symbol). The operations to traverse the data structure are based on the binary coding of the index. In comparison with previous methods, the binary indexed tree is faster, using more compact data and simpler code. The access time for all operations is either constant or proportional to the logarithm of the table size. In conjunction with the compact data structure, this makes the new method particularly suitable for large symbol alphabets.

key words: Binary indexed tree Arithmetic coding Cumulative frequencies

INTRODUCTION

A major cost in adaptive arithmetic data compression is the maintenance of the table of cumulative frequencies which is needed in reducing the range for successive symbols. Witten, Neal and Cleary¹ ease the problem by providing a move-to-front mapping of the symbols which ensures that the most frequent symbols are kept near the front of the search space. It works well for highly skewed alphabets (which may be expected to compress well) but is much less efficient for more uniform distributions of symbol frequency. Moffat² describes a tree structure (actually a heap) which provides a linear-time access to all symbols. Jones³ uses splay trees to provide an optimized data structure for handling the frequency tables. The three techniques will be referred to in this paper as MTF, HEAP and SPLAY, respectively. In all cases they attempt to keep frequently used symbols in quickly-referenced positions within the data structure, but at the cost of sometimes extensive data reorganization.

This current paper describes a new method which uses only a single array to store the frequencies, but stores them in a carefully chosen pattern to suit a novel search technique whose cost is proportional to the number of 1 bits in the element index. This cost applies to both updating and interrogating the table. In comparison with the other methods it is simple, compact and fast and involves no reorganization or movement of the data.

CCC 0038-0644/94/030327-10
© 1994 by John Wiley & Sons, Ltd.

Received 7 June 1993
Revised 6 October 1993

Peter Fenwick, 1994

A New Data Structure for Cumulative Frequency Tables

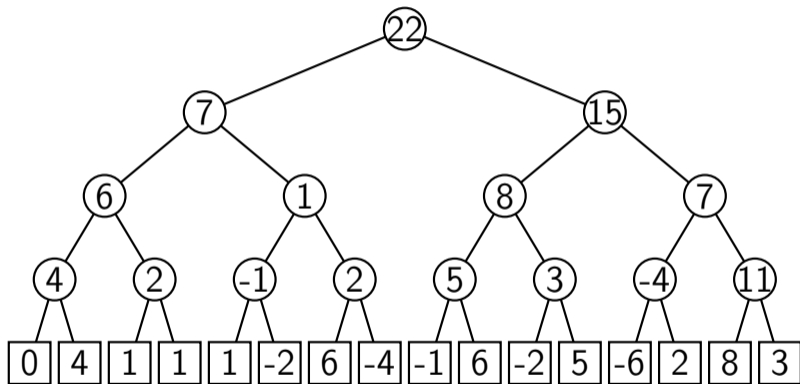
& Boris Ryabko, 1989: A fast on-line code

Implementing Fenwick trees

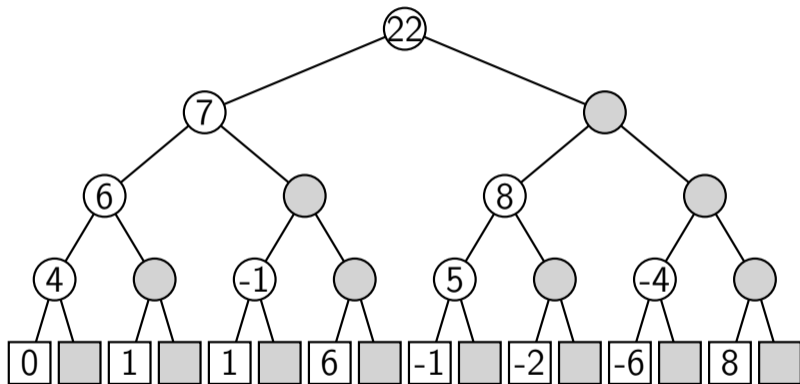
```
class FenwickTree {
    private long[] a;
    public FenwickTree(int n) { a = new long[n+1]; }
    public long prefix(int i) {
        long s = 0;
        for (; i > 0; i -= LSB(i)) s += a[i]; return s;
    }
    public void update(int i, long delta) {
        for (; i < a.length; i += LSB(i)) a[i]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

Deriving Fenwick Trees

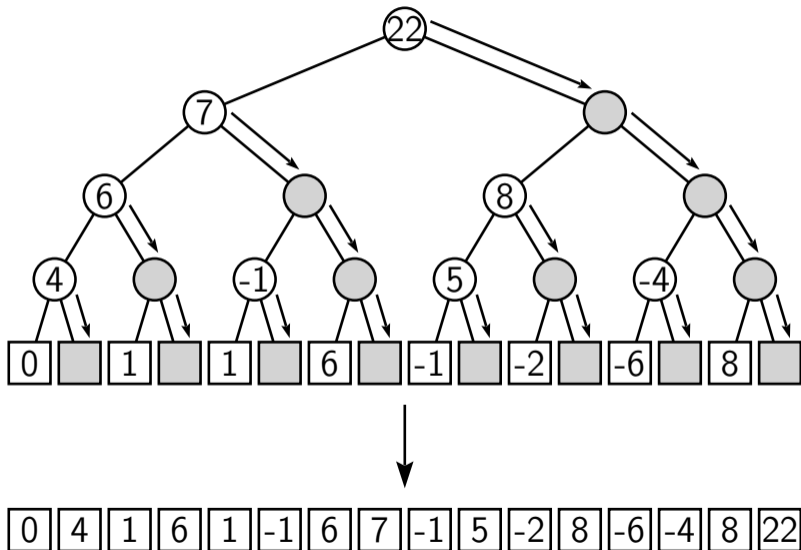
Thinning segment trees



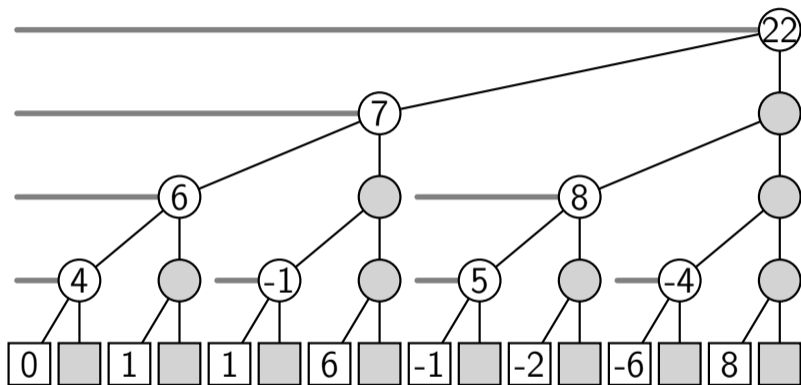
Thinning segment trees



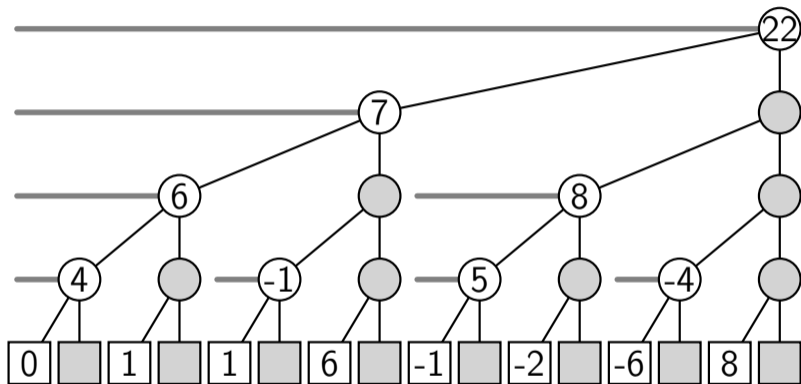
Storing a thinned segment tree



Fenwick tree = right-leaning, thinned segment tree

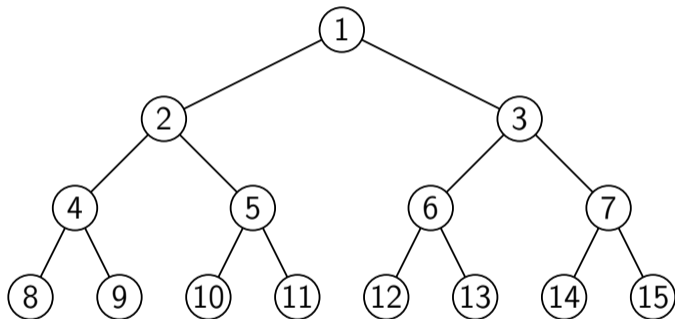


Fenwick tree = right-leaning, thinned segment tree

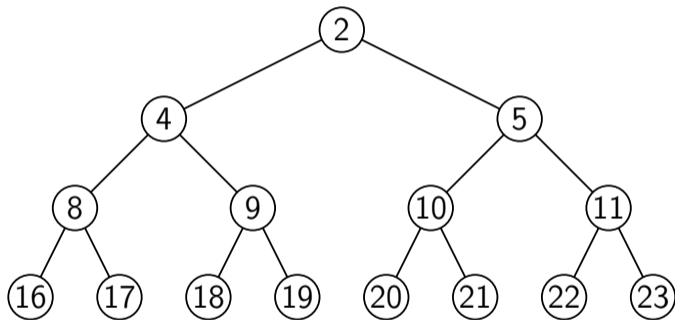


... but how to move around?

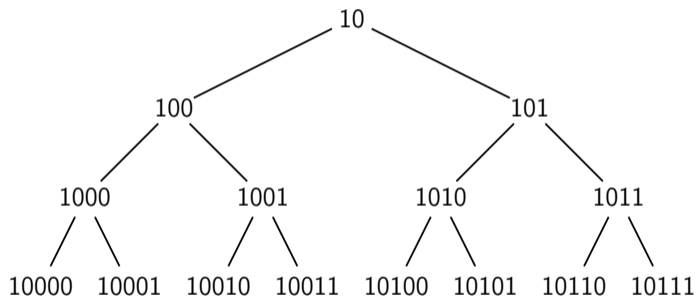
Indexing full binary trees



Indexing full binary trees



Indexing full binary trees, in binary



The Plan

- ▶ Derive Fenwick tree \leftrightarrow binary tree index conversions $f2b$, $b2f$
- ▶ Compute motion through a Fenwick tree (array) as

$$b2f \circ binaryTreeMotion \circ f2b$$

- ▶ Fuse
- ▶ ...
- ▶ Profit!

The Plan

- ▶ Derive Fenwick tree \leftrightarrow binary tree index conversions $f2b$, $b2f$
- ▶ Compute motion through a Fenwick tree (array) as

$$b2f \circ binaryTreeMotion \circ f2b$$

- ▶ Fuse
- ▶ ...
- ▶ Profit!

We're going to need some kind of DSL for manipulating numbers in binary...

Binary EDSL

Bits

$\text{data } \textit{Bit} = \textit{O} \mid \textit{I}$

$\neg :: \textit{Bit} \rightarrow \textit{Bit}$

$\neg \textit{O} = \textit{I}$

$\neg \textit{I} = \textit{O}$

$(\wedge), (\vee) :: \textit{Bit} \rightarrow \textit{Bit} \rightarrow \textit{Bit}$

$\textit{O} \wedge _ = \textit{O}$

$\textit{I} \wedge b = b$

$\textit{I} \vee _ = \textit{I}$

$\textit{O} \vee b = b$

2's complement

...000101	5
...000100	4
...000011	3
...000010	2
...000001	1
...000000	0

2's complement

...000101	5
...000100	4
...000011	3
...000010	2
...000001	1
...000000	0
...111111	-1

2's complement

...000101	5
...000100	4
...000011	3
...000010	2
...000001	1
...000000	0
...111111	-1
...111110	-2

2's complement

...000101	5
...000100	4
...000011	3
...000010	2
...000001	1
...000000	0
...111111	-1
...111110	-2
...111101	-3

Encoding infinite 2's complement bit strings?

type *Bits* = [*Bit*] ?

Encoding infinite 2's complement bit strings?

type *Bits* = [*Bit*] ?

- ▶ No decidable equality, can't convert *Bits* \rightarrow *Int*
- ▶ "Junk" values like *cycle* [*O, I*] = [*O, I, O, I, O, I, ...*]

Encoding infinite 2's complement bit strings

Valid bit strings must have some finite part followed by an infinite tail of all 0's or all 1's.

data *Bits* where

$Rep :: Bit \rightarrow Bits$

$(:.) :: Bits \rightarrow Bit \rightarrow Bits$ -- see paper for real details

Examples:

▶ $2 = \dots 000010 = Rep\ 0 :. 1 :. 0$

▶ $-5 = \dots 1111011 = Rep\ 1 :. 0 :. 1 :. 1$

Operations on infinite bit strings

$(\oplus) :: \text{Bits} \rightarrow \text{Bits} \rightarrow \text{Bits}$

$\text{Rep } x \oplus \text{Rep } y = \text{Rep } (x \wedge y)$

$(xs :: x) \oplus (ys :: y) = (xs \oplus ys) :: (x \wedge y)$

Operations on infinite bit strings

$inc :: Bits \rightarrow Bits$

$inc (Rep\ I) = Rep\ O$

$inc (bs :: O) = bs :: I$

$inc (bs :: I) = inc\ bs :: O$

$inv :: Bits \rightarrow Bits$

$inv (Rep\ b) = Rep\ (\neg b)$

$inv (bs :: b) = inv\ bs :: \neg b$

$neg :: Bits \rightarrow Bits$

$neg = inc \circ inv$

LSB

$lsb :: Bits \rightarrow Bits$

$lsb (- :: 1) = Rep\ 0 :: 1$

$lsb (bs :: 0) = lsb\ bs :: 0$

$lsb (Rep\ 0) = Rep\ 0$

LSB

$lsb :: Bits \rightarrow Bits$

$lsb \text{ } (_ :: 1) = Rep \text{ } 0 :: 1$

$lsb \text{ } (bs :: 0) = lsb \text{ } bs :: 0$

$lsb \text{ } (Rep \text{ } 0) = Rep \text{ } 0$

```
private int LSB(int i) { return i & (-i); }
```

LSB

$lsb :: Bits \rightarrow Bits$

$lsb (_ :: 1) = Rep\ 0 :: 1$

$lsb (bs :: 0) = lsb\ bs :: 0$

$lsb (Rep\ 0) = Rep\ 0$

```
private int LSB(int i) { return i & (-i); }
```

$$lsb\ x = x \oplus neg\ x$$

Proof: induction on x .

Other operations on *Bits*

set, clear

test, even, odd

shl, shr

while :: $(a \rightarrow \text{Bool}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a$

while $p f x$

| $p x$ = *while* $p f (f x)$

| *otherwise* = x

Other operations on *Bits*

set, clear

test, even, odd

shl, shr

while :: $(a \rightarrow \text{Bool}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a$

while $p f x$

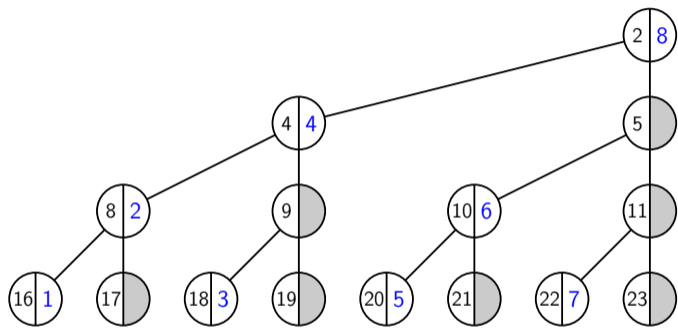
| $p x$ = *while* $p f (f x)$

| *otherwise* = x

& various rewriting lemmas, e.g. $\text{inc} \circ \text{while odd shr} = \text{while even shr} \circ \text{inc}$

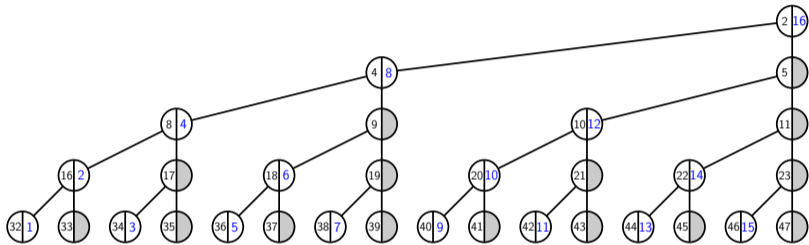
Fenwick/binary conversion

f2b



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

f2b



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

f2b

$(\Upsilon) :: [a] \rightarrow [a] \rightarrow [a]$

$[] \Upsilon - = []$

$(x : xs) \Upsilon ys = x : (ys \Upsilon xs)$

$b :: Int \rightarrow [Int]$

$b\ 0 = [2]$

$b\ n = \text{map } (2 \cdot) [2^n \dots 2^n + 2^{n-1} - 1] \Upsilon b\ (n - 1)$

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

f_{2b}

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

f2b

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

f2b\ n = dec \circ while\ even\ shr \circ set\ (n + 1)

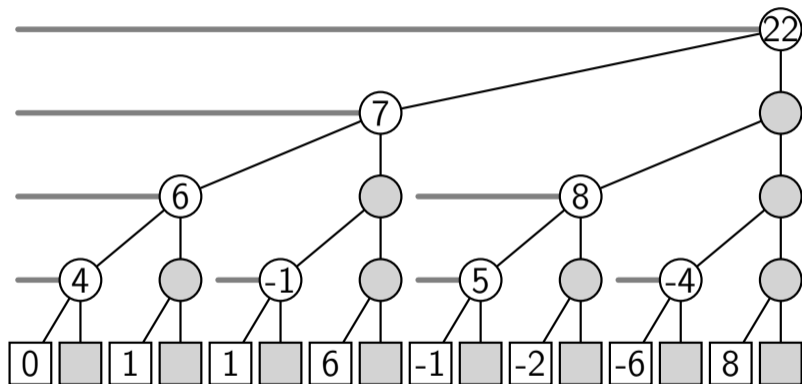
f2b

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

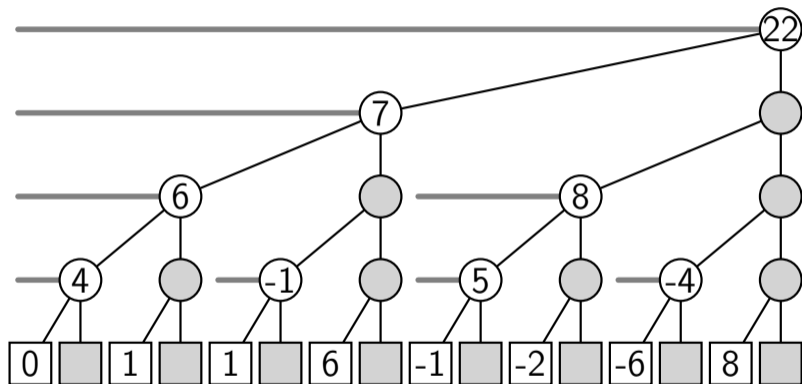
f2b\ n = dec \circ while\ even\ shr \circ set\ (n + 1)

b2f\ n = clear\ (n + 1) \circ while\ (not \circ test\ (n + 1))\ shl \circ inc

update via *activeParent*



update via *activeParent*



$activeParent = b2f \circ while\ odd\ shr \circ shr \circ f2b$

Calculating *activeParent*

$$\begin{aligned} \text{activeParent} &= b2f \circ \text{while odd shr} \circ \text{shr} \circ f2b \\ &= \{ \text{inline + rewrite...} \} \\ &\text{clear}(n+1) \circ \text{while}(\text{not} \circ \text{test}(n+1)) \text{shl} \circ \text{inc} \circ \text{while even shr} \circ \text{set}(n+1) \end{aligned}$$

0 0 1 1 0 1 0 0

↓ set sentinel bit

1 0 1 1 0 1 0 0

↓ shift right

0 0 1 0 1 1 0 1

↓ increment

0 0 1 0 1 1 1 0

↓ shift left

1 0 1 1 1 0 0 0

↓ unset sentinel bit

0 0 1 1 1 0 0 0

update

$$activeParent = \dots = \lambda x \rightarrow x + lsb\ x$$

update

$$activeParent = \dots = \lambda x \rightarrow x + lsb\ x$$

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


Thanks!

data *Bits* where

Rep :: *Bit* → *Bits*

Snoc :: !*Bits* → *Bit* → *Bits*

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 (:.) #-}