Fenwick Tree

Binary Indexing, Least Significant One-bit

beOI Training

July 10, 2016

Table of Contents

Motivating Problem

Dynamic Range Sum Query

Well, can't we simply use a Segment Tree?

Yes we can, but the Fenwick Tree datastructure has some cool advantages.

Table of Contents

How it works

Let's define $rsq(i)$ as the sum of the elements from 1 up to i

$$rsq(i) = A[1] + A[2] + \cdots + A[i]$$

We can then answer any range sum query by computing

$$rsq(a, b) = rsq(b) - rsq(a - 1)$$

We also need a point update function $update(k, v)$ which updates element k to a new value v.

Like segment trees, Fenwick trees address these functions in $\mathcal{O}(\log n)$.

Least Significant One-bit

The **least significant 1-bit** of an integer is the rightmost 1-bit of its **binary representation**.

$$(42)_{10} = (101010)_2$$
$$(1337)_{10} = (10100111001)_2$$
$$(2016)_{10} = (11111100000)_2$$

Let's define function $LSOne(i)$ as the least significant 1-bit isolated from i

$$LSOne(42) = 000010$$
$$LSOne(1337) = 00000000001$$
$$LSOne(2016) = 0000100000$$

Binary Indexing

Like in segment trees, each node is responsible for (i.e. stores the sum of) a certain range of elements.

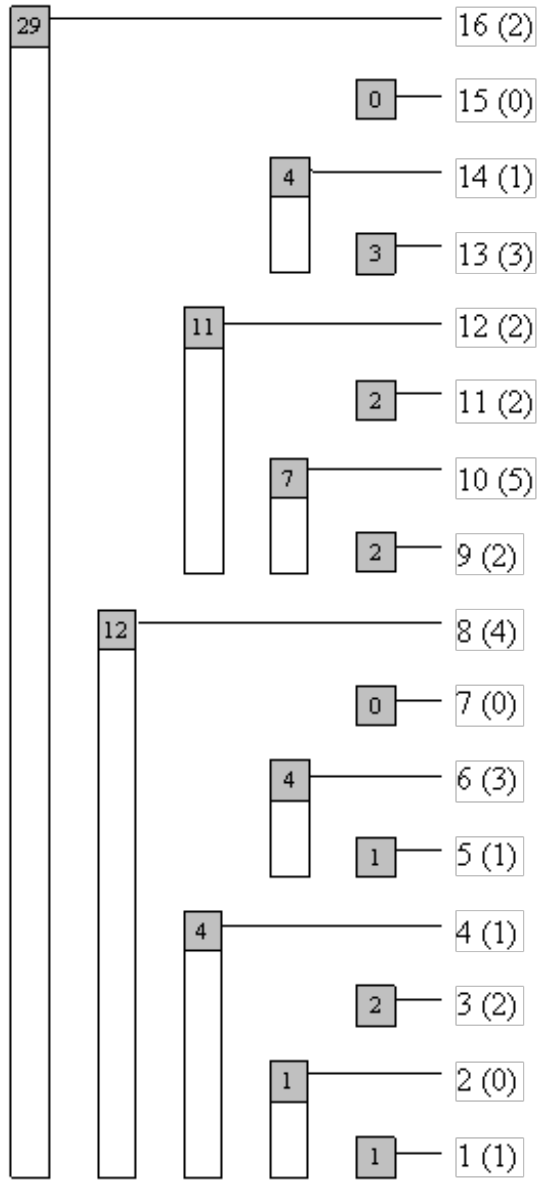
Specifically, node i is responsible for range

$$[i - LSOne(i) + 1; i]$$

This range has size $LSOne(i)$.

e.g. $LSOne(2016) = 32$, thus node 2016 will store sum of range $[1985; 2016]$ which has size 32.

Visualization



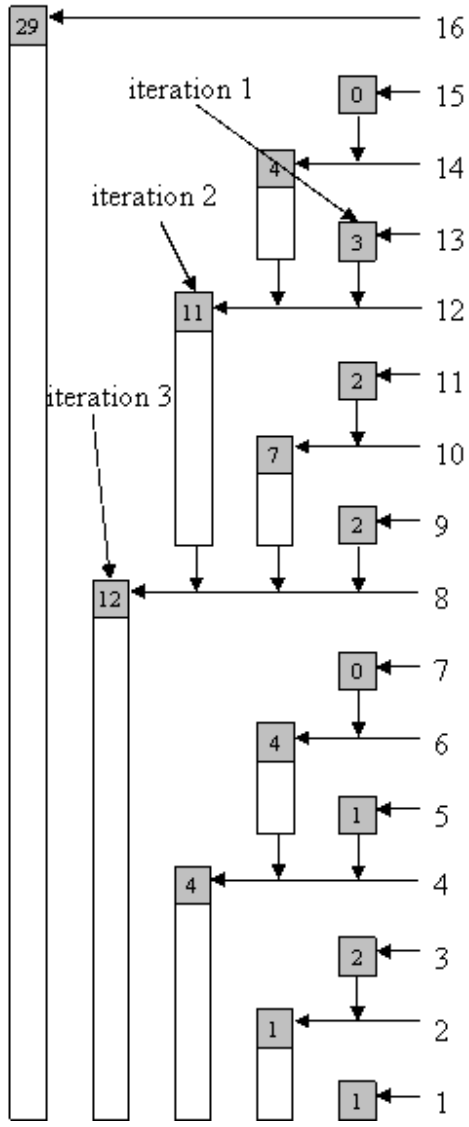
Querying

One can observe that

$$\begin{aligned} rsq(13) &= ft[1101] + ft[1100] + ft[1000] \\ &= rsq(13,13) + rsq(9,12) + rsq(1,8) \end{aligned}$$

So to obtain $rsq(i)$ we need to iteratively strip off $LSOne(i)$ and sum the values stored at the nodes we come accross.

Query Visualization



Query Complexity

The number of iterations is equal to the number of 1-bits in the starting index.

If the starting index n consists of b bits, we need at most b operations to find $rsq(n)$.

$$\mathcal{O}(b) = \mathcal{O}(\log n)$$

LSOne Implementation

There is a binary trick to compute the Least Significant One-bit really fast

$$LSOne(i) = (i \ \& \ (-i))$$

Query Implementation

```
int rsq(int i) {
    int sum = 0;
    for(; i > 0; i -= LSOne(i))
        sum += ft[i];
    return sum;
}
```

Updating

When updating index i, we need to update all nodes that are responsible for index i.

Specifically, if we update index i from v_{old} to v_{new}, we need to add $v_{\text{new}} - v_{\text{old}}$ to each node responsible for index i.

Updating

One can prove that if node k is responsible for index i, then node $k + LSOne(k)$ is the smallest node $> k$ that is also responsible for i.

$$38 = (100110)_2$$
$$38 + LSOne(38) = 40 = (101000)_2$$

Updating

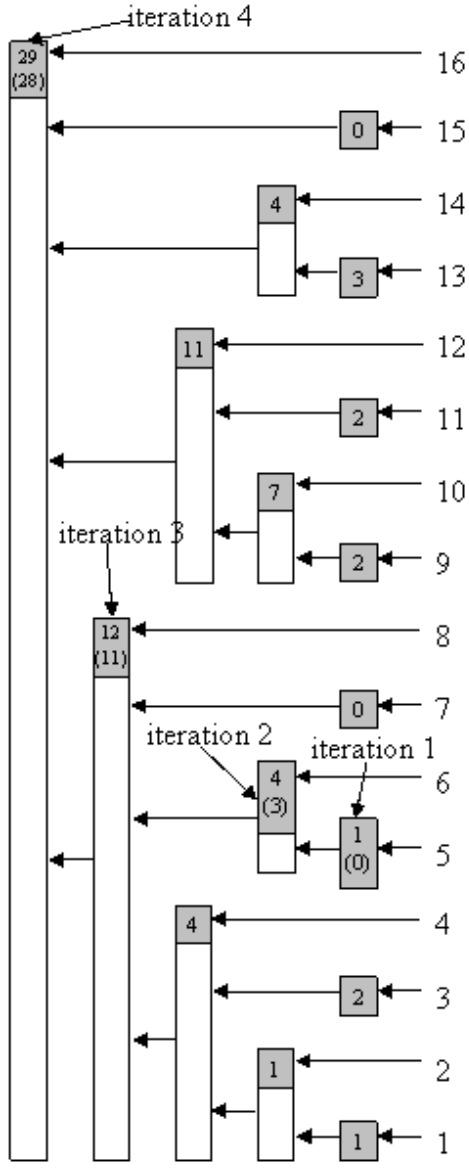
i is the smallest node responsible for index i.

Thus, to find all nodes responsible for i, we can simply start from i and iteratively add the Least Significant 1-bit.

$$\begin{aligned} \text{update } (101)_2 &= 5 \\ \text{update } (110)_2 &= 6 \\ \text{update } (1000)_2 &= 8 \\ \text{update } (10000)_2 &= 16 \\ &\vdots \end{aligned}$$

When do we stop? When we arrive at a node outside our initial array.

Update Visualization



Update Complexity

When adding $LSOne(i)$, the least significant bit is shifted at least 1 place to the left.

So, in the worst case, we need to loop through all bits until we get to an index $> n$.

$$\mathcal{O}(\log n)$$

Update Implementation

```
void adjust(int k, int v) {
    for(; k <= n; k += LSOne(k))
        ft[k] += v;
}
```

Building

To build the Fenwick Tree from the array A, there is no better way than updating each index separately.

$$\mathcal{O}(n \log n)$$

Table of Contents

Time Complexity

	Segment Tree	Fenwick Tree
Query	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Update	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Build	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$

In general, Fenwick Trees are slightly more efficient because of the fast bit trick and few memory accesses.

Memory Usage

Both use $\mathcal{O}(n)$ memory.

Code

Segment Tree: 40 lines of code

Fenwick Tree: 10 lines of code!

In contest, always use Fenwick if possible!

Applications

Beware: you cannot use Fenwick Trees for any function of your liking.

$$rsq(a, b) = rsq(b) - rsq(a - 1)$$

There is **no** such property for min/max, so Fenwick is not suitable for Dynamic RMQ.