Binary Indexed Trees (BIT)

Salvador Roura

Consider a vector of integer numbers V with a fixed size M. For commodity through all these notes, we will use the positions $1..M$ of V. (Depending on the programming language, we just disregard $V[0]$.) Suppose that we need two operations:

- Given a position i and a quantity x, add x to $V[i]$.
- Given two positions l and r, compute the sum of all numbers in $V[l..r]$.

This is the obvious C++ code for these two operations (assume that V is global):

```
void add(int i, int x) {
  V[i] += x;
}

int sum(int l, int r) {
  int res = 0;
  for (int i = l; i ≤ r; ++i) res += V[i];
  return res;
}
```

Although these codes are trivial, $sum(l, r)$ has cost $\Theta(r - l)$, which is $\Theta(M)$ in the worst case.

We can avoid this linear cost if we use the prefix sum P of V instead of V itself. Remember that $P[i]$ stores $\sum_{j=1}^{i} V[j]$. For instance, below we have a vector with $M = 14$ and its prefix sum.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
V	-	10	15	-2	19	-9	0	17	-7	-5	-3	2	-3	4	12
P	0	10	25	23	42	33	33	50	43	38	35	37	34	38	50

Note that $P[i] = P[i-1] + V[i]$. By using P, $sum(l, r)$ has constant cost:

```
int sum(int l, int r) {
  return P[r] − P[l−1];
}
```

Unfortunately, the worst-case cost of $add(i\ ,\ x)$ now becomes linear:

```
void add(int i , int x) {
  for (int j = i ; j ≤ M; ++j) P[j] += x;
}
```

So, can we implement both operations efficiently? Say, with logarithmic cost? There are at least two ways of doing so:

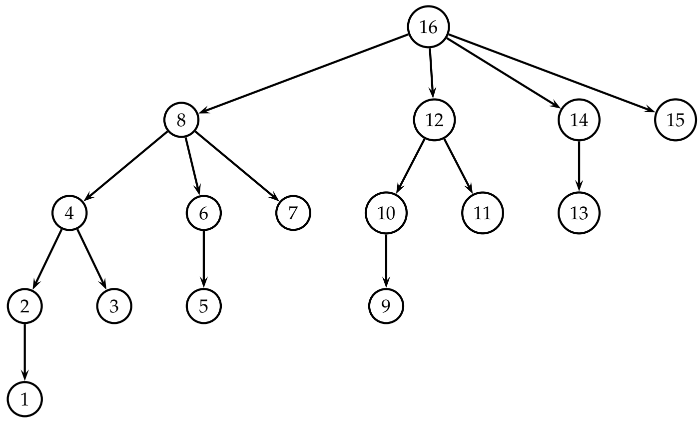
- Segment tree
- Binary Indexed Tree (BIT), also known as Fenwick Tree

Segment trees are covered elsewhere. It is a powerful data structure that allows us to solve this problem, and many more. A BIT is a more limited data structure. But, on the other hand, it is very easy to program, and with a small constant factor.

Let $b(i)$ be the (0-based) position of the last 1-bit of i. That is, $b(i)$ is the distance from the rightmost 1-bit of i to the rightmost bit of i. For instance, we have $b(13) = b(1101_2) = 0$, $b(10) = b(1010_2) = 1$, $b(12) = b(1100_2) = 2$, and $b(8) = b(1000_2) = 3$.

Let $\pi(i) = 2^{b(i)}$. Conceptually, a BIT is a tree (in fact, a forest) with its nodes labelled from 1 to M, where each node i stores the sum of $\pi(i)$ elements of V, namely $V[i - \pi(i) + 1] + \cdots + V[i]$. For instance, 13 stores $V[13]$, 10 stores $V[9] + V[10]$, 12 stores $V[9] + \cdots + V[12]$, and 8 stores $V[1] + \cdots + V[8]$.

For instance, this a BIT for $M = 14$. Note that the nodes 15 and 16 are missing, but we keep them in the picture for the sake of clarity. Every node i stores the sum of all the elements of V at the positions in the subtree rooted at i. For instance, underneath 12 we indeed have 9, 10, 11 and 12.



Two important properties of this forest are:

(a) Every element of V is stored in $O(\log M)$ nodes.

(b) For every i, $P[i]$ can be computed by adding the content of $O(\log M)$ nodes.

One advantage of a BIT is that it can be stored in a plain vector B. Following with the example with $M = 14$, we have

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
V	-	10	15	-2	19	-9	0	17	-7	-5	-3	2	-3	4	12
P	0	10	25	23	42	33	33	50	43	38	35	37	34	38	50
B	-	10	25	-2	42	-9	-9	17	43	-5	-8	2	-9	4	16

For instance, we can check that $B[12]$ stores $V[9] + V[10] + V[11] + V[12] =$ (-5) + (-3) + 2 + (-3) = -9. Observe how, because of the definition, B and P have the same values at every power of 2. Also, take into account that we only store the vector B. The vectors V and P are implicit.

Let & be as usual the C++ bitwise **and**. There is a programming trick that greatly simplifies implementing a BIT: For every $i > 0$, $(-i)\&i$ gives us $\pi(i)$. For instance (assuming 8-bit integers),

$$\begin{aligned}
-13\&13 &= 11110011_2\&00001101_2 = 00000001_2 = 1,\\
-10\&10 &= 11110010_2\&00001010_2 = 00000010_2 = 2,\\
-12\&12 &= 11110100_2\&00001100_2 = 00000100_2 = 4,\\
-8\&8 &= 11111000_2\&00001000_2 = 00001000_2 = 8.
\end{aligned}$$

To implement *add*$(i,\ x)$ we make use of property (a). To add x to $V[i]$ we only need to update $O(\log M)$ positions of B. Consider for instance $i = 5$. $V[5]$ is stored in $B[5]$, $B[6]$ and $B[8]$. To find those positions, we start at i, and we keep adding $\pi(i)$ to i while i is not larger than M: $5 + \pi(5) = 5 + 1 = 6$, $6 + \pi(6) = 6 + 2 = 8$, $8 + \pi(8) = 8 + 8 = 16$. This is the code:

```
void add(int i, int x) {
  while (i ≤ M) {
    B[i] += x;
    i += −i&i;
  }
}
```

The easiest way of implementing *sum*$(l,\ r)$ is through an auxiliary function *prefix* (i) that computes $P[i]$:

```
int sum(int l, int r) {
  return prefix(r) − prefix(l − 1);
}
```

To implement *prefix* (i) we make use of property (b): to compute $P[i]$ we only need to add $O(\log M)$ positions of B. Consider for example $i = 13$. We need to pick $B[13]$ ($B[14]$ for instance would include $V[14]$). Afterwards, we can choose $B[12]$, which stores $V[9] + \cdots + V[12]$. Finally, it is enough to choose $B[8]$.

In general, we start at the given i, and we keep subtracting $\pi(i)$ from i while i is larger than 0. With the example for $i = 13$, $13 - \pi(13) = 13 - 1 = 12$, $12 - \pi(12) = 12 - 4 = 8$, $8 - \pi(8) = 8 - 8 = 0$. This is the code:

```
int prefix (int i) {
  int res = 0;
  while (i > 0) {
    res += B[i];
    i −= −i&i;
  }
  return res;
}
```

A final comment: If you search for online material, you will see that some depicted BITs are different than the one given in these pages. Some ways are better for understanding *add*(i, x), while others are better for understanding *prefix* (i).