Дерево Фенвика

Содержание

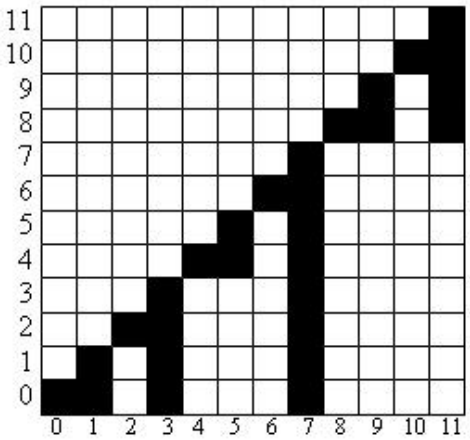
Описание структуры

Дерево Фе́нвика (англ. *Binary indexed tree*) — структура данных, требующая $O(n)$ памяти и позволяющая эффективно (за $O(\log n)$) выполнять следующие операции:

- изменять значение любого элемента в массиве,
- выполнять некоторую ассоциативную, коммутативную, обратимую операцию $\circ$ на отрезке $[i, j]$.

Впервые описано Питером Фенвиком в 1994 году.

Пусть дан массив $A = [a_0, a_1, \ldots, a_{n-1}]$. Деревом Фенвика будем называть массив T из n элементов: $T_i = \sum_{k=F(i)}^{i} a_k$, где $i = 0..n-1$ и $F(i)$ — некоторая функция, от выбора которой зависит время работы операций над деревом. Рассмотрим функцию, позволяющую делать операции вставки и изменения элемента за время $O(\log n)$. Она задается простой формулой: $F(i) = i\&(i+1)$, где $\&$ — это операция побитового логического AND. При AND числа и его значения, увеличенного на единицу, мы получаем это число без последних подряд идущих единиц.



По горизонтали — индексы массива T (T_i является суммой элементов массива A, индексы которых заштрихованы), по вертикали — индексы массива A

Эту функцию можно вычислять по другой формуле: $F(i) = i - 2^{h(i)} + 1$, где $h(i)$ — количество подряд идущих единиц в конце бинарной записи числа i. Оба варианта равносильны, так как функция, заданная какой-либо из этих формул, заменяет все подряд идущие единицы в конце числа на нули.

Запрос изменения элемента

Нам надо научиться быстро изменять частичные суммы в зависимости от того, как изменяются элементы. Рассмотрим как изменяется массив T при изменении элемента a_k.

Лемма:
Для пересчёта дерева Фенвика при изменении величины a_k необходимо изменить элементы дерева T_i, для индексов i которых верно неравенство $F(i) \leqslant k \leqslant i$.
Доказательство:

▷

$T_i = \sum_{k=F(i)}^{i} a_k, i = 0..n-1 \Rightarrow$ необходимо менять те T_i, для которых a_k попадает в $T_i \Rightarrow$ необходимые i удовлетворяют условию $F(i) \leqslant k \leqslant i$.

◁

Лемма:

Все такие i, для которых меняется T_i при изменении a_k, можно найти по формуле $i_{next} = i_{prev} \mid (i_{prev} + 1)$, где $\mid$ — это операция побитового логического OR.

Доказательство:

▷

Из доказанной выше леммы следует, что первый элемент последовательности само k. Для него выполняется равенство, так как $F(i) \leqslant i$. По формуле $i_{next} = i_{prev} \mid (i_{prev} + 1)$ мы заменим первый ноль на единицу. Неравенство при этом сохранится, так как $F(i)$ осталось прежним или уменьшилось, а i увеличилось. $F(i)$ не может увеличиться, так как функция F заменяет последние подряд идущие единицы числа i на нули, а по формуле $i_{next} = i_{prev} \mid (i_{prev} + 1)$ у нового значения i увеличивается количество единиц в конце, что не может привести к увеличению $F(i)$. Докажем от противного, что нельзя рассматривать значения i, отличные от тех, которые мы получили по формуле. Рассмотрим две различные последовательности индексов. Первая последовательность получена по формуле, вторая — некоторая последовательность чисел превосходящих k. Возьмём число j из второй последовательности, которого нет в первой последовательности. Пусть $F(j) \leqslant k$. Уберём у j все подряд идущие единицы в конце двоичной записи, столько же цифр уберём в конце числа k. Обозначим их как j_0 и k_0. Чтобы выполнялось условие $F(j) \leqslant k$, должно выполняться неравенство $j_0 \leqslant k_0$. Но если $j_0 < k_0$, то и $j \leqslant k$, что противоречит условию $j > k$. Значит, $j_0 = k_0$. Но тогда j возможно получить по формуле $i_{next} = i_{prev} \mid (i_{prev} + 1)$, следовательно, $F(j) > k$. Получили противоречие: j можно вычислить по формуле, а это значит, что оно содержится в первой последовательности. Таким образом, нужные элементы можно искать по формуле $i_{next} = i_{prev} \mid (i_{prev} + 1)$.

◁

Заметим, что $F(i)$ возрастает немонотонно. Поэтому нельзя просто перебирать значения от k, пока не нарушается условие. Например, пусть $k = 3$. При данной стратегии на следующем шаге ($i = 4$) нарушится условие и мы прекратим пересчитывать T_i. Но тогда мы упускаем остальные значения i, например 7.

i, десятичная запись	0	1	2	3	4	5	6	7	8	9	10
i, двоичная запись	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010
$F(i)$, двоичная запись	0000	0000	0010	0000	0100	0100	0110	0000	1000	1000	1010
$F(i)$, десятичная запись	0	0	2	0	4	4	6	0	8	8	10

Все i мы можем получить следующим образом: $i_{next} = i_{prev} \mid (i_{prev} + 1)$. Следующим элементом в последовательности будет элемент, у которого первый с конца ноль превратится в единицу. Можно заметить, что если к исходному элементу прибавить единицу, то необходимый ноль обратится в единицу, но при этом все

следующие единицы обнулятся. Чтобы обратно их превратить в единицы, применим операцию OR. Таким образом все нули в конце превратятся в единицы и мы получим нужный элемент. Для того, чтобы понять, что эта последовательность верна, достаточно посмотреть на таблицу.

i_{prev}	$\dots 011 \dots 1$
$i_{prev} + 1$	$\dots 100 \dots 0$
i_{next}	$\dots 111 \dots 1$

Несложно заметить, что данная последовательность строго возрастает и в худшем случае будет применена логарифм раз, так как добавляет каждый раз по одной единице в двоичном разложении числа i.

Напишем функцию, которая будет прибавлять к элементу a_i число d, и при этом меняет соответствующие частичные суммы. Так как наш массив содержит N элементов, то мы будем искать i_{next} до тех пор, пока оно не превышает значение N.

```
function modify(i, d):
   while i < N
       t[i] += d
       i = i | (i + 1)
```

Часто можно встретить задачу, где требуется заменить значение элемента a_i на x. Заметим, что если вычислить разность x и a_i, то можно свести эту задачу к операции прибавления d к a_i.

```
function set(i, x):
   d = x - a[i]
   a[i] = x
   modify(i, d)
```

Построение дерева можно осуществить, исходя из его описания. Но можно быстрее, если использовать функцию modify для каждого элемента массива A. Тогда мы получим время работы $O(n \log n)$.

```
function build():
   for i = 0 to N - 1
       modify(i, a[i])
```

Запрос получения значения функции на префиксе

Пусть существует некоторая бинарная операция $\circ$. Чтобы получить значение на отрезке $[i, j]$, нужно провести операцию, обратную к $\circ$, над значениями на отрезках $[0, j]$ и $[0, i-1]$.

В качестве бинарной операции $\circ$ рассмотрим операцию сложения.

Обозначим $G_i = \mathrm{sum(i)} = \sum_{k=0}^{i} a_k$. Тогда $\mathrm{sum(i, j)} = \sum_{k=i}^{j} a_k = G_j - G_{i-1}$.

Для нахождения $\mathrm{sum(i)}$ будем действовать следующим образом. Берём T_i, которое является суммой элементов с индексами от $F(i)$ до i. Теперь к этому значению нужно прибавить $\mathrm{sum(F(i) - 1)}$. Аналогично продолжаем складывать, пока не $F(i)$ не станет равным 0.

Покажем, что запрос суммы работает за $O(\log n)$. Рассмотрим двоичную запись числа i. Функция $F(i)$ заменила его последние единицы на нули (заметим, что количество нулей в конце станет больше, чем количество единиц в конце до этого). Теперь вычтем единицу из $F(i)$ (переход к следующему столбику). Количество единиц в конце

увеличилось, по сравнению с i, так как мы заменили все нули в конце на единицы. Проводя эти действия дальше, мы придём к тому, что получили 0. В худшем случае мы должны были повторять эти операции l раз, где l — количество цифр в двоичной записи числа i, что не превосходит $\log_2 i + 1$. Значит, запрос суммы выполняется за $O(\log n)$.

Реализация

Приведем код функции sum(i):

```
int sum(i):
   result = 0
   while i >= 0
       result += t[i]
       i = f(i) - 1
```

Сравнение дерева Фенвика и дерева отрезков

- Дерево Фенвика занимает в константное значение раз меньше памяти, чем дерево отрезков. Это следует из того, что дерево Фенвика хранит только значение операции для каких-то элементов, а дерево отрезков хранит сами элементы и частичные результаты операции на подотрезках, поэтому оно занимает как минимум в два раза больше памяти.
- Дерево Фенвика проще в реализации.
- Операция на отрезке, для которой строится дерево Фенвика, должна быть обратимой, а это значит, что минимум (как и максимум) на отрезке это дерево считать не может, в отличие от дерева отрезков. Но если нам требуется найти минимум на префиксе, то дерево Фенвика справится с этой задачей. Такое дерево Фенвика поддерживает операцию уменьшения элементов массива. Пересчёт минимума в дереве происходит быстрее, чем обновление массива минимумов на префиксе.

См. также

- Дерево отрезков

Источники информации

- Peter M. Fenwick: A new data structure for cumulative frequency (http://citeseer.ist.psu.edu/viewdoc/download;jsessionid=F180153B9C0CD797594314B736E2CCC5?doi=10.1.1.14.8917&rep=rep1&type=pdf)
- Wikipedia — Fenwick tree (http://en.wikipedia.org/wiki/Fenwick_tree)
- Maximal:: algo:: Дерево Фенвика (http://e-maxx.ru/algo/fenwick_tree)
- Хабрахабр — Дерево Фенвика (http://habrahabr.ru/post/112828)
