Дерево отрезков с изменением на отрезке

ПОДГОТОВИЛА
ДРОЗДОВА ЮЛИЯ
2 КУРС, 14 ГРУППА

Задача 0.6. Дерево отрезков

Имеется последовательность s0, …, sn − 1, состоящая из нулей. На этой последовательности могут выполняться запросы следующих типов:

1. Установить значение s [i]равным v.

2. Прибавить к каждому элементу с индексом из отрезка [a, b] число v.

3. Найти сумму элементов с индексами из отрезка [a, b].

4. Найти минимум среди элементов с индексами из отрезка [a, b].

5. Найти максимум среди элементов с индексами из отрезка [a, b].

Какие проблемы?

Второй запрос мы пока что умеем делать только за O(N log N). Очевидно, что такой вариант не подойдет.

И что теперь делать?

Заметим несколько моментов.

Пусть есть вершина v, которая отвечает за какой-то отрезок [L, R].

Поступает запрос на прибавление числа x на отрезке [L, R]. Как изменятся t[v].Min, t[v].Max, t[v].Sum?

Очевидно, что минимум и максимум на этом отрезке изменятся на x.

Что будет с суммой?

Вершина v отвечает за отрезок [L, R], т.е за $L - R + 1$ элементов. Каждый из этих элементов увеличился на x, а значит сумма увеличилась $(L - R + 1)x$.

Немного размышлений

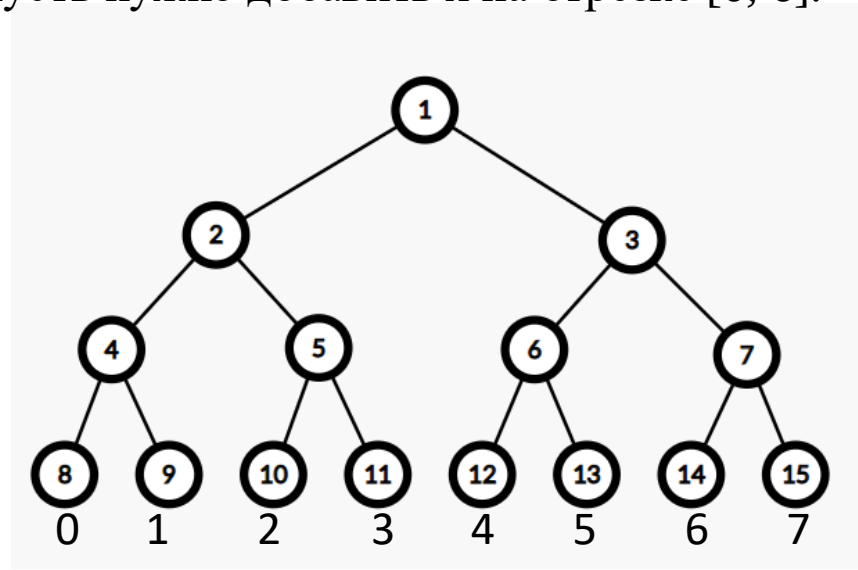
А нужно ли нам явно добавлять к каждому элементу x? Ведь сам элемент нам может понадобится только в 1ом запросе или при поиске максимума/минимума/суммы при условии, что L=R? Но в этом случае мы спустимся по дереву до нужного элемента при выполнении этих запросов. Тогда можно выполнить все добавления только в том случае, когда они явно нам понадобятся.

А как это делать?

Будем использовать технику ленивого добавления. То есть, будем обновлять наши значения только по мере необходимости.

Дальше рассмотрим на примере.

Пусть нужно добавить x на отрезке [0, 6].



Проще всего провести аналогию с поиском суммы. Если бы мы искали сумму на этом отрезке, то

$$sum = t[2] + t[6] + t[14]$$

Аналогично в обновлении на начальном этапе будут изменены только 2, 6 и 14 вершины дерева.

Все еще ничего непонятно☹

Чтобы как-то помечать вершины, в которых что-то поменялось заведем массив *lazy*.

$lazy[v]$ – значение, которое нужно прибавить ко всем числам на отрезке, за который отвечает вершина v.

```
void Add(int v, int cl, int cr, int l, int r, int x) {
    if (l > cr || r < cl){
        return;
    }
    if (cl != cr){
        Push(v, cl, cr);
    }
    if (l <= cl && cr <= r) {
        lazy[v] += x;
        t[v].Max += x;
        t[v].Min += x;
        t[v].Sum += x * (cr - cl + 1);
        return;
    }
    int mid = (cl + cr) / 2;
    Add(v * 2, cl, mid, l, r, x);
    Add(v * 2 + 1, mid + 1, cr, l, r, x);
    t[v].Sum = t[v * 2].Sum + t[v * 2 + 1].Sum;
    t[v].Min = min(t[v * 2].Min, t[v * 2 + 1].Min);
    t[v].Max = max(t[v * 2].Max, t[v * 2 + 1].Max);
}
```

Кто такой Push? И как искать ответ на запросы?

В поисках ответа на запрос мы оказались в вершине v, которая отвечает за отрезок [cl, cr].

На этом отрезке были произведены какие-то изменения, т.е. *lazy[v]* не равен 0. Хорошо, если отрезок, за который отвечает вершина v, полностью входит в отрезок, для которого мы ищем ответ. Тогда мы просто вернем t[v].Sum.

Но что делать, если это не так? Нужно спускаться ниже, а значит нужно сказать детям нашей вершины v, что их тоже меняли раньше. Это и делает функция Push().

Немного кода

```
void Push(int v, int cl, int cr) {
    if (lazy[v]) {
        int sz = (cr - cl + 1) / 2;
        lazy[v * 2] += lazy[v];
        lazy[v * 2 + 1] += lazy[v];
        t[v * 2].Sum += lazy[v] * sz;
        t[v * 2].Max += lazy[v];
        t[v * 2].Min += lazy[v];
        t[v * 2 + 1].Sum += lazy[v] * sz;
        t[v * 2 + 1].Max += lazy[v];
        t[v * 2 + 1].Min += lazy[v];
        lazy[v] = 0;
    }
}
```

Замечание: в функциях Update() и Push() мы прибавляем новые изменения, а не приравниваем, т.к. у нас могут быть добавления из предыдущих запросов.

А что изменится в функциях поиска суммы/максимума/минимума?

В общем-то код останется тот же. Единственное различие – это добавление вызова функции Push(), если спускаемся в детей.

```
long long GetSum(int v, int cl, int cr, int l, int r) {
    if (l > cr || r < cl)
        return 0;
    if (cl != cr) {
        Push(v, cl, cr);
    }
    if (l <= cl && cr <= r) {
        return t[v].Sum;
    }
    int mid = (cl + cr) / 2;
    return GetSum(v * 2, cl, mid, l, r) +
           GetSum(v * 2 + 1, mid + 1, cr, l, r);
}
```

Немного замечаний по поводу кода

В моей реализации дерева отрезков каждая вершина отвечает за отрезок [L, R]. Также длина начального массива должна быть равна степени двойки. Если не равна, то просто добавляем 0 в конец массива, пока не получим степень 2.

Но в любом случае идея алгоритма не меняется, если использовать дерево отрезков, которое рассказывалось на лекциях.

Выводы

Теперь обновление на отрезке работает за $O(\log N)$, чего мы и добивались. Все тоже самое можно реализовать и для дерева отрезков, в котором вершина отвечает за полуинтервал [L, R), и которое не будет полным, но это оставим на самостоятельное рассмотрение.

Аналогично будет решаться задача, где вместо прибавления на отрезке, будет присвоение.

Похожим образом реализуется умножение на отрезке, но для него нужен дополнительный массив, чтобы помечать вершины, которые изменились, т.к. при умножение на 0 поменяет ответ, а в нашей реализации это не учтено.

Всё! Спасибо за внимание!