Многомерные структуры и персистентность

В прошлых главах мы разобрались с самыми простыми версиями структур: одномерными. Сегодня настало время поговорить о более сложных модификациях изученных структур. Первая из таких модификаций довольно естественная: раньше наши структуры делали что-то на одномерном массиве, а теперь нам хочется обработать двумерный массив и делать какие-то запросы на прямоугольниках. Вторая же концепция, именуемая *«персистентностью»* менее естественна: давайте при всех изменениях структуры так её частично копировать, чтобы иметь доступ сразу к нескольким версиям – это, например, позволит проще и эффективнее строить двумерные структуры.

Многомерное дерево Фенвика

Как раньше и говорилось читателю, в многомерном дереве Фенвика достаточно добавить во все места по циклу. В реальных задачах размерности выше второй не встречаются, поэтому все дальнейшие структуры мы будем рассматривать в двумерном случае, но при желании они обобщаются и на большие размерности. Вот так например будет выглядеть двумерное дерево Фенвика:

```
struct FenwickTree2D {
    vector<vector<int>> t;
    int n, m;

    FenwickTree2D(int _n, int _m) : n(_n + 1), m(_m + 1) {
        t.assign(n, vector<int>(m, 0));
    }

    void add(int x, int y, int d) {
        for (int i = x; i < n; i += i & -i) {
            for (int j = y; j < m; j += j & -j) {
                t[i][j] += d;
            }
        }
    }

    int sum(int x, int y) {
        int result = 0;
        for (int i = x; i > 0; i -= i & -i) {
            for (int j = y; j > 0; j -= j & -j) {
                result += t[i][j];
            }
        }
        return result;
    }

    int sum(int x1, int y1, int x2, int y2) {
        return sum(x2, y2) + sum(x1 - 1, y1 - 1) - sum(x1 - 1, y2) - sum(x2, y1 - 1);
    }
};
```

В этой реализации нужно пользоваться индексами $[1; n]$ по первой координате и $[1; m]$ по второй координате. Асимптотика будет $O(\log^2 n)$ на запрос и структура потребует $O(n^2)$ памяти.

Многомерная разреженная таблица

С разреженной таблицей в целом всё тоже не очень сложно, правда теперь придётся хранить не двумерный массив, а четырёхмерные, но мы справимся:

```cpp
#define FOR(i, n) for (int i = 0; i < n; ++i)
struct SparseTable2D {
    vector<vector<vector<vector<int>>>> v;          // размер n × m × log_n × log_m
    int n, m, log_n, log_m;

    SparseTable2D(int _n, int _m) : n(_n), m(_m) {
        log_n = __lg(n) + 1;
        log_m = __lg(m) + 1;
        v.assign(n, vector<vector<vector<int>>>(m,
                        vector<vector<int>>(log_n, vector<int>(log_m))));
    }

    void build(vector<vector<int>> a){
        FOR(i, n) FOR(j, m) {
            v[i][j][0][0] = a[i][j];
        }
        FOR(i, n) FOR(lnj, log_m - 1) FOR(j, m - (1 << lnj)) {
            v[i][j][0][lnj + 1] = min(v[i][j][0][lnj], v[i][j + (1 << lnj)][0][lnj]);
        }
        FOR(lni, log_n - 1) FOR(i, n - (1 << lni)) FOR(lnj, log_m) FOR(j, m) {
            v[i][j][lni + 1][lnj] = min(v[i][j][lni][lnj], v[i + (1 << lni)][j][lni][lnj]);
        }
    }

    int get_min(int x1, int y1, int x2, int y2) {
        int lgx = __lg(x2 - x1 + 1);
        int lgy = __lg(y2 - y1 + 1);
        int ans1 = v[x1][y1][lgx][lgy];
        int ans2 = v[x2 - (1 << lgx) + 1][y1][lgx][lgy];
        int ans3 = v[x1][y2 - (1 << lgy) + 1][lgx][lgy];
        int ans4 = v[x2 - (1 << lgx) + 1][y2 - (1 << lgy) + 1][lgx][lgy];
        return min(min(ans1, ans2), min(ans3, ans4));
    }
};
```

Прошу меня простить за использования макроса `FOR`, но с ним код получается значительно короче. Использовать же индексы нужно из диапазонов $[0, n)$ по первой координате и $[0, m)$ по второй. Время на запрос составляет $O(1)$, правда она займёт $O(n^2 \log^2 n)$ памяти и столько же времени на изначальное построение.

Многомерное дерево отрезков

С одной стороны концептуально многомерное дерево отрезков делается даже проще чем предыдущие структуры: в вершинах внешнего ДО храним внутренние ДО. Правда могут возникнуть некоторые сложности в написании, поэтому всё равно немного обсудим двумерное ДО.

Пусть нам нужно считать суммы на прямоугольниках и есть запросы изменения элемента. Тогда сделаем ДО по координате x, а в каждой его вершине будем хранить ДО для заданного x, то есть это ДО по координате y. Тогда если хотим посчитать сумму на прямоугольнике с углами (x_1, y_1) и (x_2, y_2) то запускаемся для ДО по координате x на отрезке $[x_1, x_2]$ – оно найдёт какие-то $O(\log n)$ вершин, которые хочет добавить к ответу (итого прямоугольник запроса разбился на вертикальные полоски). А в каждой из этих вершин запускаемся на отрезке $[y_1, y_2]$ для соответствующего ДО по координате y – так оно найдёт какие-то $O(\log n)$ вершин сумму в которых и есть сумма для соответствующего отрезка. Итого потребуется $O(\log^2 n)$ операций на запрос.

Для обновления элемента (x, y) достаточно так же спуститься по x ДО и начти все обновляемые y ДО, обновить значение в них как в обычных ДО – на это уйдёт $O(\log^2 n)$. А потом для x ДО сделать все $O(\log n)$ обновлений. Итого на вторую часть потратится столько же, сколько на первую и суммарно уйдёт $O(\log^2 n)$ времени.

Также при желании можно реализовать и запросы изменения на прямоугольниках, это делается за такую же асимптотику.

Общая информация о персистентности

Как уже было сказано выше, персистентность – это про сохранение доступа ко всем версиям при изменении структуры. Для общего развития скажем, что различают несколько видов персистентности:

- *«Частичная»* – можно изменять последнюю версию и просматривать все.
- *«Полная»* – можно изменять любую версию и просматривать любую версию.
- *«Конфлюэнтная»* – полная персистентность, в которой можно объединять несколько версий.

Самое понятное применение у полной персистентности, но иногда бывает нужна и конфлюэнтная.

В каких-то частных случаях персистентности можно добиваться простым сохранением всех запросов к определенному элементу. Но мы же рассмотрим общий случай, который применим для любых древовидных структур данных (куча, дерево отрезков и декартово дерево, если говорить про описанные в книге). На структуру накладывается два требования: во-первых в вершинах не могут храниться ссылки на детей, а во-вторых каждый запрос должен выглядеть как спуск к одной вершине дерева с приемлемой для копирования глубиной. Тогда общий алгоритм, как сделать любую такую структуру персистентнтной (как изменить структуру, чтобы иметь доступ как к новой версии, так и к старой):

- Дополнительно нужно будет хранить массив, в i-ой ячейки которого будем хранить корень i-ой версии структуры.
- Пока спускаемся по дереву – сохраняем все проходимые вершины.
- Если во время спуска мы делаем проталкивания не в вершины пути, то вершины, куда информация протолкнулась нужно скопировать.
- Когда мы окончили спуск, то делаем необходимые изменения в вершины.
- Теперь поднимаемся от вершины наверх и пересчитываем данные в родительских вершинах, новые данные сохраняем в копию вершин.

Если изложить это более кратко, то основной смысл в том, чтобы при каждом изменении структуры создавать копии всех изменяемых вершин. Тогда если прошло q запросов к структуре на n элементах со средней глубиной вершин $O(\log n)$, то структура займёт $O(n + q \log n)$ памяти и ответ на каждый запрос будет делаться за $O(\log n)$.

По описанному выше алгоритму можно сделать персистентными все описанные в книге древовидные структуры: кучу, дерево отрезков и декартово дерево.

Применение персистентности

В своё время я понял сам алгоритм персистентности, но вот когда она может быть применима – не понял.

Первый вариант, когда бывает нужна персистентность – это учебные задачи, в которых явно сказано сделать что-то персистентным и иметь возможность обращаться к разным версиям. Это понятный тип задач, на нём мы останавливаться не будем.

Второй тип задач – в которых есть в каком-то виде дерево и на нём нужно посчитать какую-то большую динамику, причём иногда приходят запросы что-то в дереве поменять. Тогда на можно данные такой динамики хранить в персистентной структуре и в каждой вершине объединять данные из детей. В итоге получится, что в каждой вершине посчитана какая-то динами и к ней можно производить обращения. Примером такой задачи может быть: нужно проверять, является ли отрезок массива ПСП – тогда заводим ДО под элементы массива и в каждой вершине ДО заводим персистентное ДД, хранящее определённые скобки этого отрезка (более подробного разбора не будет чтобы читателю было самому интересно решить эту задачу), а зная какие-то последовательности скобок в детях можно как-то из них получить последовательность скобок для родителя.

И наконец третий тип задач – в них нужно посчитать что-то на одномерном массиве, но вы придумали воспринимать задачу как двумерную. Примером такой задачи является вычисление k-ой порядковой статистики на отрезке массива неотрицательных чисел. Тогда будем воспринимать задачу как двумерную: на плоскости элементу a_i на плоскости отметим точку (i, a_i) – теперь от нас требуется на заданном отрезке $[i, j]$ найти такой y, что в прямоугольнике с углами $(i, 0), (i, y), (j, y), (j, 0)$ содержится всего k точек. Но согласно идеи с префиксными суммами, нам будет достаточно знать количество точек в прямоугольнике $(0, 0), (0, y), (i, y), (i, 0)$ и аналогичном для j. А тогда давайте сделаем массив из n деревьев отрезков, и в i-ом дереве будем хранить информацию про числа $a_0, a_1, \ldots, a_i$. Тогда пересчёт $i + 1$-го дерева это модификация i-го, что делается с помощью персистентности, а для ответа на запрос на отрезке $[i, j]$ нужно будет делать одновременный спуск по двум ДО. Опять же, если читателя заинтересовала задача, то он может самостоятельно разобраться во всех тонкостях её написания.