ETT. Splay. Link Cut Tree.

Содержание.

1. Euler Tour Tree
2. Амортизация
3. Splay Tree
4. Link Cut Tree

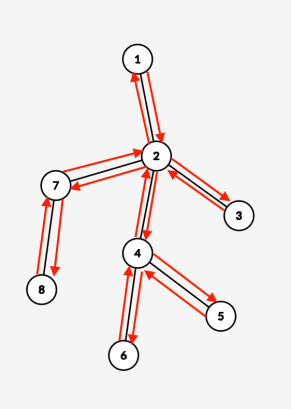
Euler Tour Tree

Мы хотим научиться решать следующую задачу:

Дан лес на n вершинах. В онлайне приходят запросы:

1. Добавить ребро (u, v), u и v находятся в разных компонентах связности
2. Удалить ребро (u, v)
3. Сказать, лежат ли u и v в одной компоненте связности

Давайте рассмотрим любой эйлеров обход дерева, выпишем рёбра в порядке перехода по ним.



Сохраним ребра в декартовом дереве. Для каждой вершины сохраним $out(v)$ — указатель на ребро, которое выходит из неё, $in(v)$ — указатель на ребро, которое входит в неё.

Как обрабатывать запросы?

- **Проверка на связность:** пусть $r(e)$ — корень ДД, в котором лежит ребро e. Проверим, что $r(in(u)) = r(in(v))$.
- **Добавление ребра:** введём функцию $make\_root(v)$, которая сделает вершину v коренем её дерева. Для этого циклически сдвинем ДД, чтобы $in(v)$ было первым ребром в массиве. Теперь выполним $make\_root(u)$, $make\_root(v)$, после чего добавим в конец ДД вершины u [ребро (u, v), ДД вершины v, ребро (v, u)].
- **Удаление ребра:** для каждого ребра сохраним обратное ему ребро $back(e)$. Сделаем ребро e первым в ДД. Найдем индекс ребра $back(e)$ в ДД (для этого нужно для каждой вершине в ДД хранить её предка). Отрежем всё левее $back(e)$, удалим рёбра e и $back(e)$. Структура перестроилась корректно.

Другие задачи

1. Дан лес на n вершинах. Нужно научиться обрабатывать запросы:
   - Добавить ребро (u, v), u и v находятся в разных компонентах связности
   - Удалить ребро (u, v)
   - Найти XOR чисел на ребрах на пути (u, v)

   Для этого найдем префиксный XOR до $in(u)$ и до $in(v)$. Тогда ответ равен XOR этих чисел.

2. Дано дерево на n вершинах, корень постоянный. Нужно научиться обрабатывать запросы:
   - Подвесить вершину u к вершине v, вершина u до этого не встречалась
   - Удалить ребро (u, v), вершина v является листом
   - Найти $lca(u, v)$

   Если мы подвесили дерево, то у каждого ребра есть направление — “вниз” или “вверх”. На ребрах “вниз” поставим число 1, а на ребрах вверх число −1. Достаточно просто вырезать отрезок от $in(u)$ до $in(v)$ и на нём найти вершину ДД с наименьшей префиксной суммой.

Амортизация

Очередь на двух стеках

Рассмотрим потенциал $\Phi = |st_{in}|$. Изначально $\Phi = 0$. При операции `push` потенциал увеличивается на 1. При операции `pop` потенциал либо не меняется, либо становится нулевым. Приведеннное время работы $\tilde{t_i} = t_i + \Delta\Phi_i$. В первом случае $\tilde{t_i} = 1 + 1 = 2$, во втором — либо 1, если st_{out} не пуст, либо $1 + |st_{in}| - |st_{in}| = 1$.

Дек на минимум

Рассмотрим потенциал $\Phi = |st_{in} - st_{out}|$. При операции добавления Φ просто увеличивается на 1, поэтому $\tilde{t_i} \in \{0, 2\}$. При операции удаления либо стек не пуст и $\tilde{t_i} \in \{0, 2\}$, либо происходит движение половины стека и Φ становится равным 0 или 1. В этом случае $\Delta\Phi = |st_{out}|$ (без ограничения общности), поэтому $\tilde{t_i} = |st_{out}| - |st_{out}| + 1 = 1$.

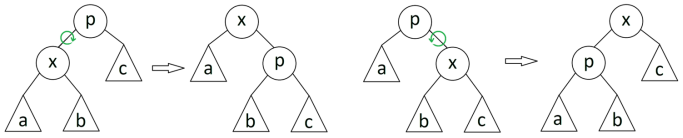
Splay Tree

Сплей дерево — двоичное дерево поиска с амортизированным временем работы.

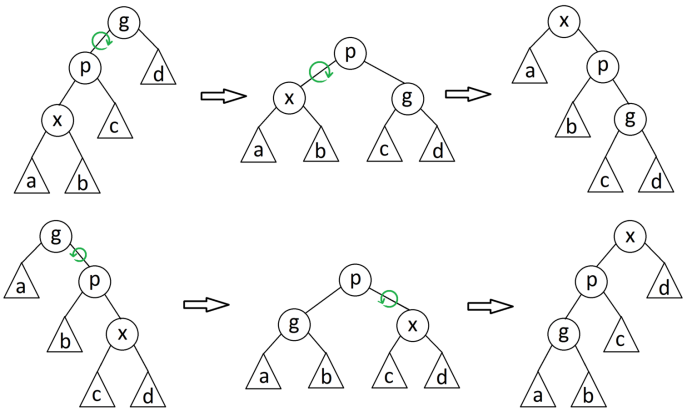
Операции со Splay Tree

Основной нашей операцией будет `splay`. Она будет брать некоторую вершину и перестраивать дерево, чтобы она стала корнем. Мы будем просто подниматься из вершины наверх, применяя одну из трёх операций (для каждой из них ещё есть симметричная):

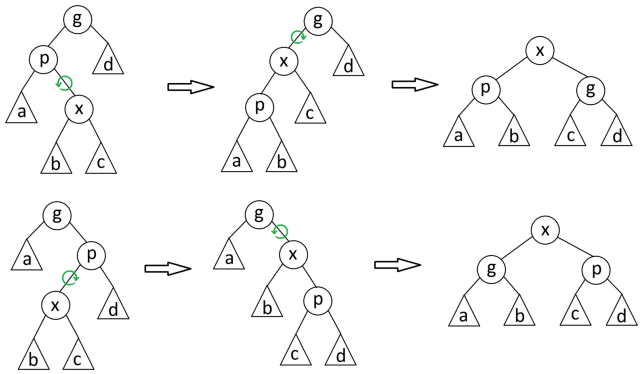
- **zig** применяется, если p — корень дерева и мы хотим поднять в корень вершину x.



- **zig-zig** применяется, если p не корень и x расположено с той же стороны от p, что и p от g.



- **zig-zag** применяется, если p и x расположены по разные стороны от своих родителей



Окей, давайте научимся делать `split` и `merge`.

merge: пусть есть два дерева t_1 и t_2, что все элементы t_1 левее всех элементов t_2. Возьмем самый левый лист t_2 и сделаем `splay` от него, после чего у корня t_2 не будет левого ребенка — туда мы и подвесим t_1.

split: пусть есть дерево t и число k. Мы хотим разбить t на два дерева t_1 и t_2, что в t_1 попадут все элементы $\le k$, а в t_2 — остальные. Спустимся в наибольший элемент $\le k$. Сделаем `splay` от него. Отрежем правое поддерево — оно и будет t_2.

Анализ асимптотики

Пусть $s(x)$ — размер поддерева вершины x, $r(x) = \log_2 C$.

Утв. Амортизированное время `splay` вершины x в дереве с корнем t не более $3r(t) - 3r(x) + 1$.

□ Мы докажем для `zig-zig`. Остальное делается аналогично. Итак, мы делаем `zig-zig` для вершин x, p, g, как на картинке. За $r'(v)$ обозначим ранг вершины v после поворота.
$\tilde{t_i} = 2 + (r'(x) + r'(p) + r'(g)) - (r(x) + r(p) + r(g))$

Поддерево x после поворота такое же, как у g до него $\Rightarrow r'(x) = r(g)$. $\tilde{t_i} = 2 + r'(p) + r'(g) - r(x) - r(p)$.
1. $r(p) \ge r(x)$
2. $r'(p) \le r'(x)$

$\tilde{t_i} \le 2 + r'(x) + r'(g) - 2r(x)$

$\frac{\lg(a) + \lg(b)}{2} \le \lg\left(\frac{a+b}{2}\right)$ и $s(x) + s'(z) \le s'(x)$, откуда $\frac{r(x) + r'(g)}{2} \le r'(x) - 1$, то есть
$\tilde{t_i} \le 2 + r'(x) + 2(r'(x) - 1) - 3r(x) = 3r'(x) - 3r(x)$ ■

Link Cut

Окей, настало время изучить еще одну структуру данных, которая позволит нам хранить набор деревьев, а также добавлять/удалять ребра. В этот раз структура будет мощнее ETT и также позволит вычислять функции на путях в нашем дереве (а на самом деле — и на поддеревьях).

Изначально мы разобьем дерево на непересекающиеся вертикальные пути произвольным образом, как в HLD будем для каждого пути хранить его предка, то есть предка самой высокой вершины пути.

expose

Основной операцией, которую мы будем использовать в нашей структуре данных (а также операцией, за счет которой у нас будет хорошая асимптотика), будет операция `expose`.

1. В Splay-дереве, соответствующему нашему пути, делаем `splay` от нашей вершины.
2. Наша вершина — корень. Смотрим на предка нашего пути. Делаем `splay` от него. Отрезаем часть пути, которая ниже этой вершины. Объединяем два Splay-дерева. Не забываем назначить кусочку пути, который мы отрезали нового предка.
3. Продолжаем так делать, пока не дойдем до корня.

make_root

Как и в случае с ETT, мы бы очень хотели научиться делать вершину корнем дерева — тогда все остальные

операции становятся тривиальными. Для этого нужно сделать `expose(v)`, а потом развернуть путь от v до корня.

link

```
link(a, b):
  make_root(v);
  b->treeParent = a;
```

cut

```
cut(a, b):
  make_root(a);
  expose(b);
  auto [l, r] = split(a, 1);
  r->treeParent = a;
```

connected?

```
connected(a, b):
  make_root(a);
  expose(b);
  splay(b);
  return left_son(b) == a;
```

min_on_path

```
min_on_path(a, b):
  make_root(a);
  expose(b);
  splay(a);
  posa = get_pos(a);
  posb = get_pos(b);
  return get_min(a, posa, posb);
```

LCA

```
lca(root, a, b):
  make_root(root);
  expose(a);
  return last_parent_while_expose(b);
```

Анализ асимптотики

Ребро (u, v), где v — родитель называется легким, если $s(u) < \frac{s(v)}{2}$. Очевидно, что на пути до корня не более $\log n$ легких ребер. Остальные ребра называются тяжелыми. Пусть L — множество легких ребер, а H — множество тяжелых ребер.

Во время выполнения `expose` ребра из наших путей становятся “пунктирными” и наоборот. Такие преобразования происходят парами. Пусть пар таких преобразований M. $M =$
$|\{\text{edges converted from dashed to solid}\}| =$
$|\{\text{light dashed edges converted to solid}\}| +$
$|\{\text{heavy dashed edges converted to solid}\}|$. Рассмотрим потенциал Φ как $n - 1 - |\{\text{heavy solid edges}\}|$.

Утв. $M + \Delta\Phi$ не более чем на $2\log n + 1$ за одну операцию `expose`.

□ $M + \Delta\Phi = M +$
$|\{\text{heavy solid edges converted to dashed}\}| -$
$|\{\text{heavy dashed edges converted to solid}\}| \le M +$
$\underbrace{|\{\text{light dashed edges converted to solid}\}|}_{\text{так как если убрали тяжелое, добавилось легкое!}} -$
$|\{\text{heavy solid edges converted to dashed}\}| \le 2 \cdot$
$|\{\text{light dashed edges converted to solid}\}| \le 2\log n + 1$ ■

Итак, приведенное время работы операции `expose` будет $O(\log^2 n)$, если использовать произвольное бинарное дерево поиска для хранения путей.