Link-Cut Tree + Splay

ЛКШ 2026

22.07.2026

1. Метод потенциалов

Иногда так бывает, что над структурой выполняются какие-то дорогостоящие операции (например, с асимптотикой $O(n)$), но из-за их низкой частоты асимптотика выполнения q операций остается $O(q \log n)$. Для доказательства такой асимптотики часто используется амортизационный анализ. Сегодня мы заострим внимание на одном из его методов: *метод потенциалов*.

Пусть у нас есть некая структура размера n, над которой выполняется q операций. Введем для каждого состояния структуры какую-то волшебную величину Φ под названием потенциал (для каждой задачи выбирается свой потенциал): изначально потенциал равен Φ_0, а после i-й операции он равен Φ_i.

Пусть во время операции i выполняется t_i действий.

Введем *амортизационную стоимость* операции: $a_i = t_i - (\Phi_{i-1} - \Phi_i)$.

Тогда $\dfrac{\sum\limits_{i=1}^{q} t_i}{q} = O(f(n,q))$, если выполнено два условия:

1. $a_i = O(f(n,q))$
2. $|\Phi_i| = O(q \cdot f(n,q))$

Доказательство:

$$\frac{\sum\limits_{i=1}^{q} t_i}{q} = \frac{\sum\limits_{i=1}^{q} (a_i + (\Phi_{i-1} - \Phi_i))}{q} = \frac{(\Phi_0 - \Phi_q) + \sum\limits_{i=1}^{q} a_i}{q} =$$
$$= \frac{O(q \cdot f(n,q)) + O(q \cdot f(n,q))}{q} = O(f(n,q))$$

Пример:

Рассмотрим стек с тремя операциями:

1. *add*: добавить элемент в конец.
2. *pop*: удалить последний элемент.
3. *multipop*: удалять элементы, пока стек не опустеет.

Последняя операция может работать за $O(n)$ в редких случаях, однако здравый смысл подсказывает: чтобы удалить k элементов из стека, эти k элементов сначала нужно добавить. Давайте введем потенциал, равный размеру стека. Проанализируем a_i:

1. *add*: $a_i = 1 + 1 = 2$
2. *pop*: $a_i = 1 - 1 = 0$

3. *multipop*: если в стеке k элементов, то $a_i = k - k = 0$

Таким образом, $a_i = O(1)$ и $|\Phi_i| \leq n$, а значит, каждая операция в среднем выполняется за $O(1)$.

Данная идея потребуется нам для доказательства асимптотик рассматриваемых нами структур: *Link-Cut Tree* и *Splay Tree*.

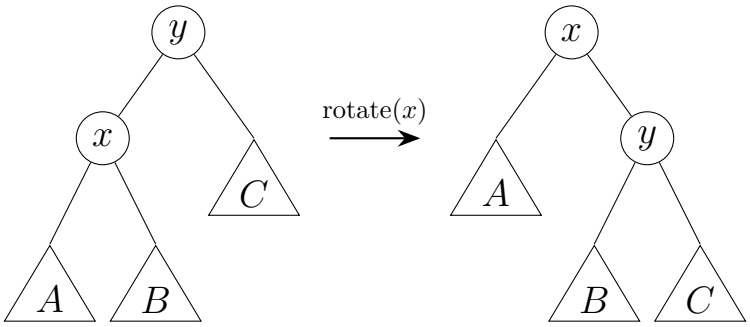
2. Splay Tree

Splay-дерево является самоперестраивающимся бинарным деревом поиска (оно не гарантирует высоты $O(\log n)$). Сам алгоритм описывается довольно просто, а вот то, почему q запросов суммарно работают быстрее, чем за $O(nq)$, требует использования метода потенциалов.

Свое название дерево получило из-за главной операции, которая в нем используется, — ***splay***. Данная операция с помощью последовательного применения *поворотов* (операция ***rotate***) делает вершину корнем дерева. За счет правильного применения поворотов достигается хорошая асимптотика.

2.1. rotate(x)

Данная операция *"поворачивает"* бинарное дерево относительно ребра:



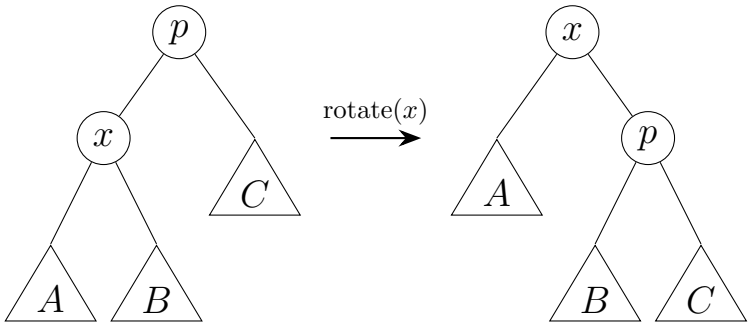
Поворот в другую сторону делается симметрично.

2.2. Паттерны поворотов

Существует три паттерна поворотов:

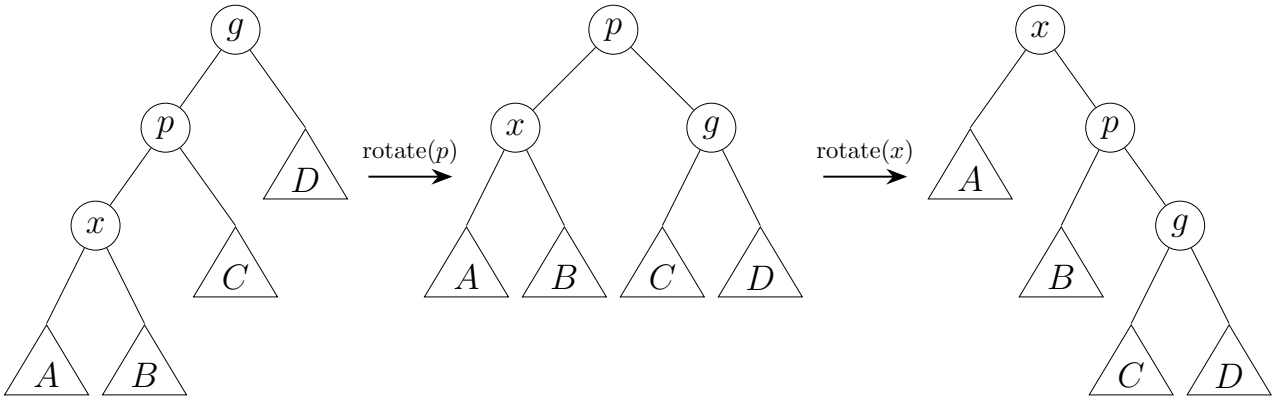
1. *zig*

   Применяется, когда родитель нашей вершины — корень дерева. К вершине просто применяется rotate(x):



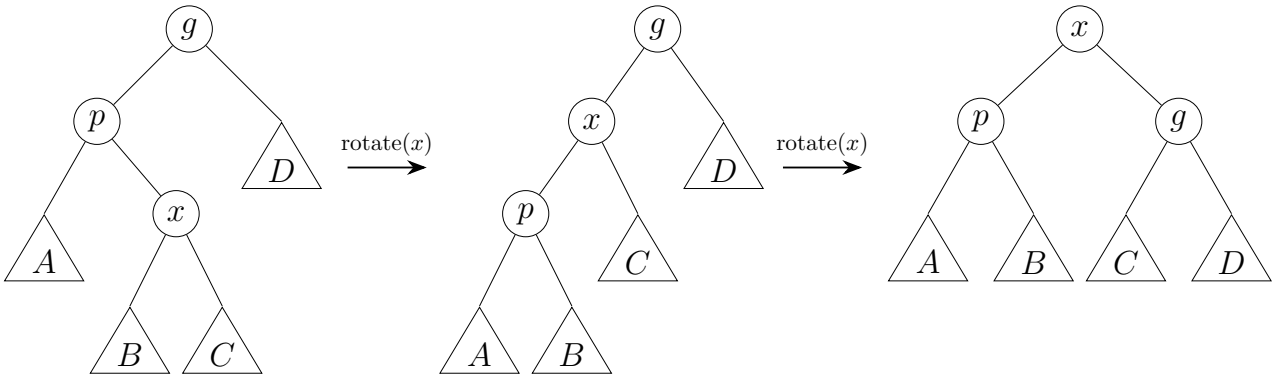
2. *zig-zig*

   Применяется, когда наша вершина и ее родитель — дети с одной стороны (оба левые или оба правые). Сначала поворот применяется к родителю, а уже затем — к самой вершине:



3. *zig-zag*

   Применяется, когда наша вершина и ее родитель — дети с разных сторон (один левый, а другой правый). Поворот дважды применяется к самой вершине:



2.3. splay(x)

Данная операция применяет нужный паттерн поворотов, пока вершина x не станет корнем. **Важно:** любая операция, спускающаяся по дереву, должна заканчиваться вызовом ***splay*** от последней посещенной вершины — даже если нужный ключ не был найден. Иначе не будет амортизационной оценки $O(\log n)$.

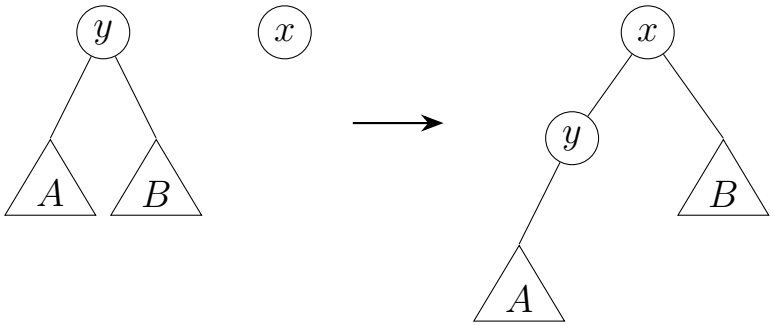
2.4. Основные операции

1. *find*

   Чтобы проверить, есть ли нужный ключ в нашем splay-дереве, нужно спуститься по нему, как в любом другом бинарном дереве поиска. Самое главное — не забыть вызвать ***splay*** от конечной вершины нашего спуска вне зависимости от того, нашли мы нужный ключ или нет. Таким образом, если ключ есть, то после выполнения операции он будет корнем.

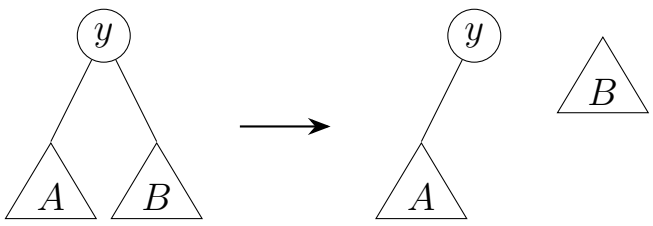
2. *insert*

   Вызовем *find*. Если ключ есть в дереве, то проигнорируем запрос. Иначе в корне дерева окажется либо самый большой ключ, меньший добавляемого, либо самый маленький ключ, больший добавляемого. Разберем первый случай (второй симметричен). Знаем, что $A < y < x < B$:



3. *split*

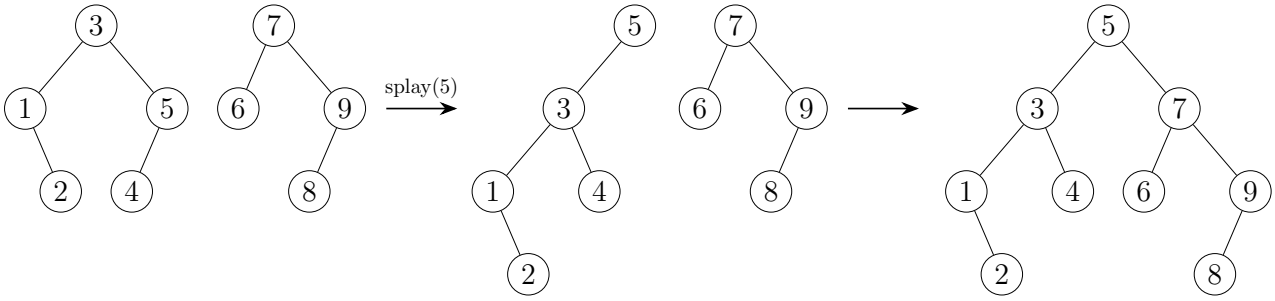
   Хотим разделить дерево на два: в первом будут все значения меньше x, а во втором — все значения $\geq x$. Заметим, что можно снова вызвать операцию *find*, после чего разобрать три случая: в корне оказался сам ключ x, наибольший ключ меньше x или наименьший ключ больше x. Разберем тот же случай, что разбирали в *insert*. Знаем, что $A < y < x < B$:



   Заметим, что для операции *insert* можно вызвать *split*, после чего подвесить два полученных дерева к добавляемой вершине (и не забыть рассмотреть случай, когда добавляемый ключ уже есть в дереве).

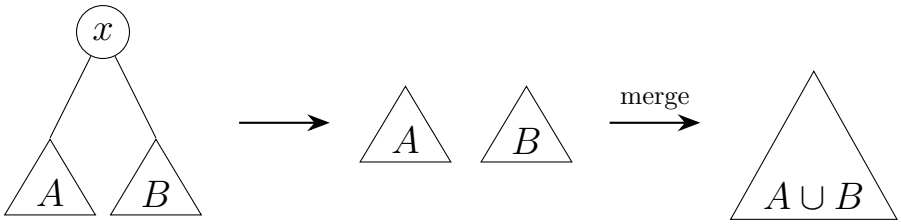
4. *merge*

   Поскольку все ключи в первом дереве меньше всех ключей во втором, было бы очень удобно, если бы мы могли просто подвесить второе дерево к первому. К сожалению, это не всегда получается сделать, ведь у корня может быть правый сын. Однако в splay-дереве эта проблема решается очень просто: давайте просто сделаем корнем первого дерева самый большой элемент в нем с помощью операции ***splay***, после чего подвесим к нему справа второе дерево:



5. *erase*

   Вызовем *find*. Если ключа нет в дереве, то проигнорируем запрос. Иначе удаляемый ключ окажется в корне. Удалим корень, после чего объединим его поддеревья с помощью операции *merge*:



2.5. Неявное splay-дерево

Реализуется аналогично неявному декартову дереву: нужно хранить размеры поддеревьев и аккуратно обновлять их при каждом rotate.

2.6. Массовые операции

Достаточно хранить пуши, как в дереве отрезков или декартовом дереве, и проталкивать их при спуске и перед поворотами.

2.7. Доказательство асимптотики

Перейдем к самой ~~бесполезной~~ скучной части — доказательству того, что splay-дерево выполняет запрос в среднем за $O(\log n)$ (действительно, совсем непонятно, почему не может получиться так, что ***splay*** всегда будет работать за $\Omega(n)$).

Введем потенциал вершины: $r(v) = \log_2 sz_v$, где sz_v — размер ее поддерева. Потенциалом дерева назовем сумму всех потенциалов вершин. Заметим, что при применении паттерна поворотов потенциал меняется только у вершин x, p и g, а поддеревья других вершин никак не затрагиваются.

Нам понадобится следующий факт:

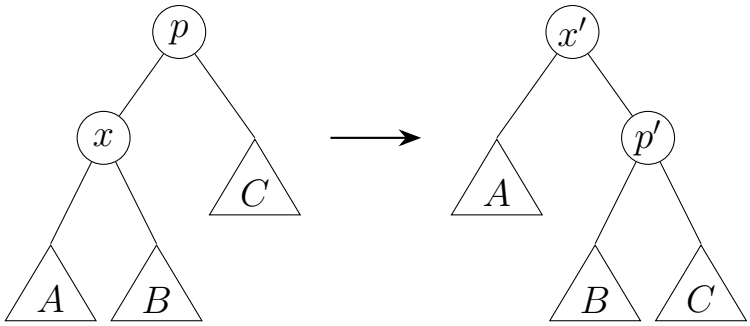
Лемма 2.1: Невероятный факт про сумму логарифмов

Если $a + b \le c$, то $\log_2 a + \log_2 b \le 2 \log_2 c - 2$.

Действительно, $ab \le \left(\frac{a+b}{2}\right)^2 \le \frac{c^2}{4}$, остается взять логарифм от обеих частей.

Аккуратно рассмотрим изменение потенциала при поворотах:

1. *zig*

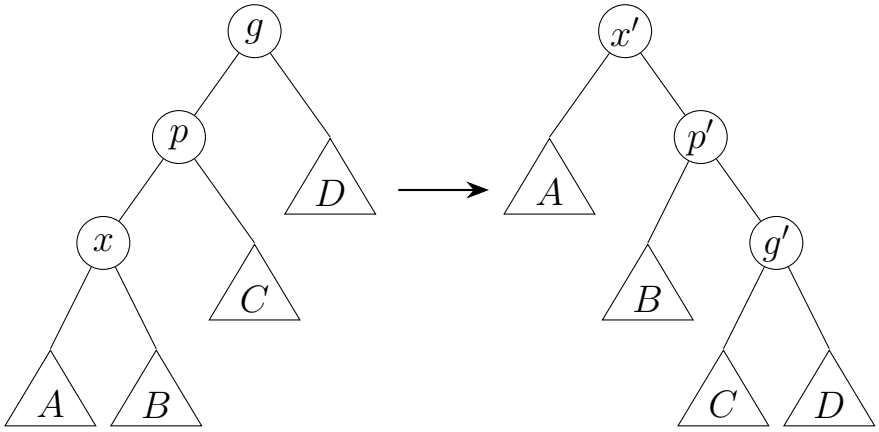


x' — корень всего изменяющегося поддерева, поэтому $r(x') = r(p)$.

$$a_{\text{zig}} = 1 + r(x') + r(p') - r(x) - r(p) = 1 + r(p') - r(x) \le$$
$$\le 1 + r(x') - r(x) \le 1 + 3\,(r(x') - r(x))$$

Воспользовались тем, что $r(p') \le r(x')$, так как p' — сын x', и $r(x') \ge r(x)$, так как поддерево x увеличилось.

2. *zig-zig*



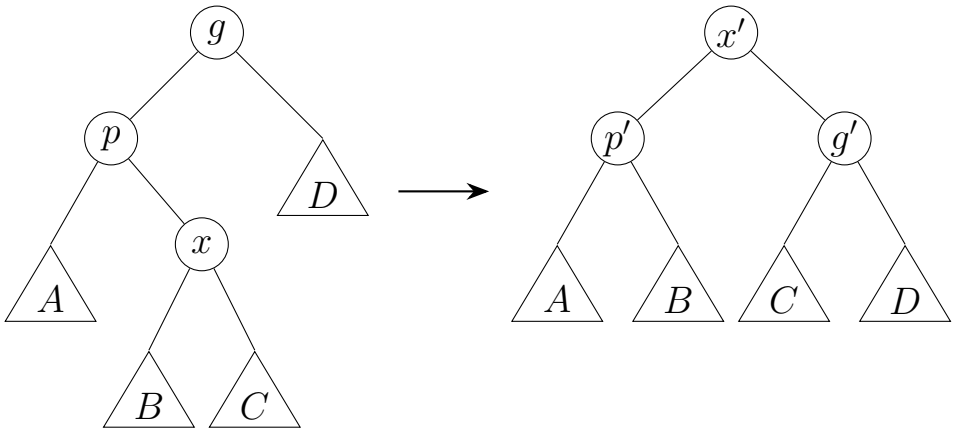
$r(x') = r(g)$, поэтому

$$a_{\text{zig-zig}} = 2 + r(x') + r(p') + r(g') - r(x) - r(p) - r(g) =$$
$$= 2 + r(p') + r(g') - r(x) - r(p) \leq 2 + r(x') + r(g') - 2r(x)$$

Использовали $r(p') \leq r(x')$ и $r(p) \geq r(x)$. Заметим, что $sz_x + sz_{g'} \leq sz_{x'}$: по лемме 2.1 $r(x) + r(g') \leq 2r(x') - 2$, то есть $r(g') \leq 2r(x') - r(x) - 2$. Подставим:

$$a_{\text{zig-zig}} \leq 2 + r(x') + (2r(x') - r(x) - 2) - 2r(x) = 3\,(r(x') - r(x))$$

3. *zig-zag*



Снова $r(x') = r(g)$ и $r(p) \geq r(x)$, поэтому

$$a_{\text{zig-zag}} = 2 + r(x') + r(p') + r(g') - r(x) - r(p) - r(g) =$$
$$= 2 + r(p') + r(g') - r(x) - r(p) \leq 2 + r(p') + r(g') - 2r(x)$$

Заметим, что $sz_{p'} + sz_{g'} \leq sz_{x'}$: по лемме 2.1 $r(p') + r(g') \leq 2r(x') - 2$. Значит,

$$a_{\text{zig-zag}} \leq 2r(x') - 2r(x) \leq 3\,(r(x') - r(x))\,,$$

где последний переход верен из-за $r(x') \geq r(x)$.

Сложим амортизационные стоимости всех паттернов одного вызова ***splay*** вершины x: слагаемые $3\,(r(x') - r(x))$ телескопируются, а *zig* применяется максимум один раз, поэтому если t — корень дерева, то

$$a_{\text{splay}} \le 3\,(r(t) - r(x)) + 1 \le 3r(t) + 1 \le 3\log_2 n + 1 = O(\log n)$$

Остается проверить, что все операции имеют хорошую амортизационную сложность после введения такого потенциала. Заметим, что после *find* мы всегда вызываем ***splay***, а значит, спуск в *find* можно "оплатить" потенциалом из ***splay*** (достаточно считать, что потенциал, например, "оплачивает" два действия, а не одно).

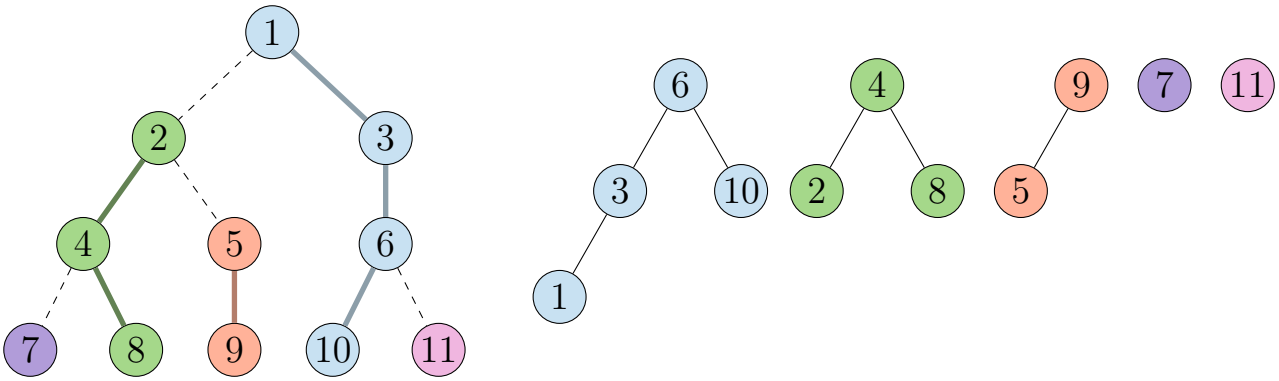
Все операции, кроме *insert* и *merge*, не добавляют новых вершин (а иногда даже удаляют) и завязаны только на *find*, а потому оценка для них тоже выполняется. А в случае *insert* мы удаляем сына у одной вершины + добавляем новую вершину в корень, а значит, потенциал изменится только на $O(\log n)$, поскольку все другие поддеревья не меняются. А значит, у *insert* тоже правильная амортизационная сложность. А во время *merge* мы увеличиваем потенциал ровно одной вершины (корня одного из деревьев, который мы хотим объединить), поэтому потенциал структуры изменится на $O(\log n)$.

Заметим, что $0 \le \Phi_i \le n \cdot r(t) \le n\log_2 n = O(n\log n)$, а потому при $q \sim n$ выполнено $|\Phi_i| = O(q\log n)$ и каждая операция выполняется в среднем за $O(\log n)$.

3. Link-Cut Tree

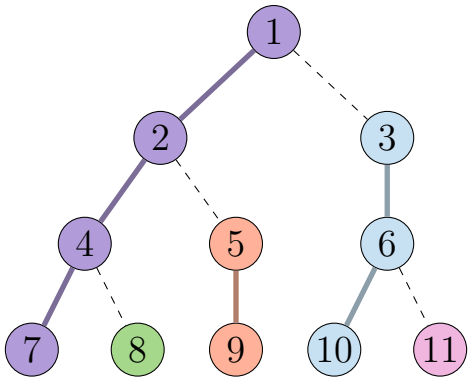
Представим себе задачу: дано дерево, нужно уметь считать сумму на пути. Мы умеем быстро такое решать с помощью Heavy-Light Decomposition. Теперь добавим туда еще два запроса: добавить ребро, удалить ребро (гарантируется, что граф в любой момент остается лесом). Возникает проблема: при изменении структуры дерева пути, на которые мы его разбили, могут поломаться, а чинить их мы никак не умеем. Link-Cut Tree решает эту проблему.

В любой момент времени будем хранить корневое дерево (или лес корневых деревьев, если их несколько), которое разбивается на какое-то количество непересекающихся (по вершинам) вертикальных путей. Каждый путь будет храниться в splay-дереве с вершинами в порядке увеличения глубины. Мы не будем накладывать никаких других условий на эти пути. Например, корректное разбиение дерева (и соответствующие путям splay-деревья):



3.1. expose(v)

Для выполнения всех запросов нам потребуется по сути одна операция — ***expose***. Она перестраивает пути так, что вершина v оказывается в одном пути с корнем дерева, причем является его нижним концом. Реализуем ее так: для начала разрежем путь вершины v на два: все, что ниже v, и остальное. Начнем подниматься из вершины v. В какой-то момент мы сменим путь, в котором находимся, и окажемся в вершине u — родителе верхнего конца нашего пути. Отрежем все, что ниже u, от ее пути и присоединим вместо этой части путь вершины v. Продолжим так подниматься, пока v и корень не окажутся в одном пути. Применим операцию ***expose*** к фиолетовой вершине из примера выше — в конце концов получится такое разбиение дерева на пути:



3.2. Основные операции

1. *make_root*

   Вторая по важности операция: делает переданную вершину v новым корнем дерева. Нам нужно не сломать ориентацию путей (ведь вершины в путях хранятся в порядке сверху вниз). Давайте сделаем ***expose*** от v. Теперь при переподвешивании ориентация в других путях не изменится (к тому же они останутся вертикальными), а вот путь вершины v придется перевернуть. Поэтому неявные splay-деревья, в которых мы будем хранить пути, должны уметь разворачивать дерево с помощью массовых операций.

2. *link*

   Хотим добавить ребро (u, v). Давайте сделаем *make_root(v)* и *expose(u)*, после чего укажем в вершине v, что предок ее пути — это вершина u.

3. *cut*

   Хотим удалить ребро (u, v). Давайте сделаем *make_root(u)* и *expose(v)*, после чего u и v окажутся в одном пути, причем u — сосед v сверху. Остается разрезать этот путь между u и v.

4. *get*

   Чтобы сделать запрос на пути из вершины u в вершину v, необходимо сделать *make_root(u)*, *expose(v)*, после чего искомый путь будет совпадать с путем вершины v.

3.3. Доказательство $O(\log^2 n)$

Мы уже знаем, что ***splay*** работает за $O(\log n)$. Поэтому можно сделать оценку для ***expose*** — $O(k \log n)$, где k — количество различных путей при подъеме от v до корня. Докажем, что k в среднем $O(\log n)$.

Назовем ребро (v, u), где u — сын v, тяжелым, если размер поддерева u составляет хотя бы половину размера поддерева v (мы рассматриваем именно реальное дерево, а не splay — аналогично HLD). Очевидно, что на пути от вершины v до корня есть не более $O(\log n)$ легких ребер (действительно, каждый раз, когда мы поднимаемся по легкому ребру, мы увеличиваем размер нашего поддерева хотя бы в два раза). Введем потенциал: $\Phi = \#$ тяжелых ребер не в путях Link-Cut Tree. Уже сейчас можно убедиться, что $\Phi_i \leq n - 1$, остается проверить *амортизационную стоимость*.

Проанализируем ***expose***. Есть несколько случаев:

1. Обрезали путь вершины v в самом начале. Это добавит не более одного тяжелого ребра, учитываемого в потенциале.

2. Мы перешли в новый путь по легкому ребру. Такое случается не чаще, чем встречаются легкие ребра, то есть не более $O(\log n)$ раз. Заметим, что из-за того, что мы обрезаем другой путь, могло добавиться не более одного тяжелого ребра, которое дает вклад в потенциал. Потенциал мог увеличиться не больше чем на $O(\log n)$.

3. Мы поднялись по тяжелому ребру. Но в таком случае потенциал уменьшается на 1, а значит, мы "оплатили" стоимость подъема (при этом отрезанное ребро не могло быть тяжелым, так как у вершины не бывает сразу двух тяжелых сыновей).

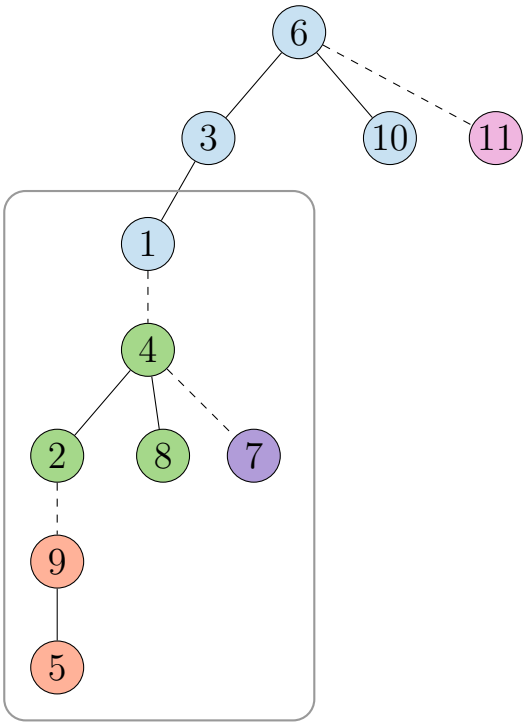
Таким образом, $a_{\text{expose}} \leq O(\log n) + O(\log n) + 1 = O(\log n)$, то есть k в среднем равно $O(\log n)$, а ***expose*** в среднем работает за $O(\log^2 n)$ (на самом деле, умножать амортизированные оценки не очень-то и правильно, поэтому правильнее будет сказать, что единица потенциала оплачивает $\Theta(\log n)$ действий, которые мы потратим на ***splay***).

Убедимся, что операции ничего не ломают. При *get* мы вызываем только ***expose***, а значит, потенциалы изменяются корректно. Во время *make_root* помимо ***expose*** мы еще и переворачиваем реальное дерево. Поэтому какие-то ребра могут стать тяжелыми, а какие-то перестать ими быть. Единственная проблема — размеры поддеревьев вершин перевернутого пути могут измениться, и какое-то "боковое" ребро не из пути в вершину пути может стать тяжелым. Пусть "боковое" ребро в вершину w стало тяжелым. Но в таком случае вершина, которая находится под w в пути, имеет размер поддерева хотя бы в два раза меньше, чем поддерево w (так как у вершины не бывает двух тяжелых сыновей), а значит, у нас может быть не более $O(\log n)$ "боковых тяжелых ребер", следовательно, потенциал увеличился не более чем на $O(\log n)$. В случае *cut* мы удаляем одно ребро из корня всего дерева, а значит, тяжелое ребро могло измениться только у корня. Значит, потенциал, возможно, поменяется на константу — ничего не сломалось. А в случае *link* у нас добавляется одно ребро + меняются размеры поддеревьев у всех вершин от u до корня. Добавленное ребро дает не больше 1 в потенциал. А с поддеревьями давайте заметим, что они лежат в одном пути, а значит, ребра между ними не учитываются в потенциале. С учетом того, что размер поддеревьев только увеличился, мы могли только уменьшить потенциал — снова все сошлось.

3.4. Доказательство $O(\log n)$

Давайте будем считать, что операция ***expose*** работает не за $O(k \log n)$, а за $O(k + T_{\text{splay}})$, где T_{splay} — суммарное число действий, потраченное всеми операциями ***splay*** во время ***expose***. Мы уже доказали, что в среднем k — это $O(\log n)$, давайте сделаем то же самое для T_{splay}. Для этого нам придется немного изменить потенциалы в splay-дереве, чтобы усилить нашу оценку.

Виртуальное поддерево: давайте мысленно подвесим к каждой вершине v в splay-дереве все корни splay-деревьев путей, которые подвешиваются к v по виртуальным ребрам (то есть ребрам, которые не входят в разбиение на пути). Например, для разбиения из примера выше (дерево до применения ***expose(7)***) получится такое виртуальное дерево (пунктиром обозначены виртуальные ребра, обведено виртуальное поддерево вершины 1):



Пусть $w(v)$ — это размер виртуального поддерева v. Сделаем потенциал $r(v) = \log_2 w(v)$. Заметим, что если делать отрезание нижней части пути u и подвешивание на ее место пути вершины v с помощью ***splay(u)***, то оно никак не меняет виртуальные поддеревья вершин, а значит, во время операции ***expose*** потенциалы может поменять только вызов ***splay***. Но, повторив доказательство для splay-дерева уже с новыми потенциалами, получим, что T_{splay} в среднем равно $O(\log n)$ (надо снова получить оценку на splay + убедиться, что значения телескопируются).

Вновь проверим, что операции ничего не ломают. *make_root* и *get* делают только вызовы ***expose***, а значит, ничего не портят. В случае с *link* u уже корень своего виртуального дерева, а потому при добавлении ребра (u, v) изменится только ее потенциал, а значит, потенциал всей структуры увеличится не больше чем на $O(\log n)$. В случае с *cut* удаление ребра может только уменьшить потенциал всей структуры, поэтому операция корректна.

Таким образом, мы доказали, что в среднем Link-Cut Tree работает за $O(\log n)$.