1 Бинарное дерево поиска

В каждом узле бинарного дерева поиска хранятся *ключ a* и два поддерева, правое и левое. Все ключи в левом поддереве не превосходят a, а в правом — не меньше a. Алгоритм поиска — начиная с корня, сравниваем искомый ключ с ключом в узле, в зависимости от сравнения спускаемся в правое или в левое поддерево.

Вставка в бинарное дерево — поиск + вставляем туда, куда пришёл поиск. Чтобы удалить элемент — ставим на его место самый левый элемент в его правом поддереве.

Проблема такой наивной структуры — может вместо дерева получиться палка (если, например, ключи приходят в порядке по убыванию), и поиск будет занимать $\mathcal{O}(n)$. Красно-чёрные деревья, например, следят за тем, чтобы дерево всегда имело высоту $\mathcal{O}(\log n)$.

Definition 1. Дерево называется идеально сбалансированным (perfectly balanced tree), если размеры детей каждой ее вершины отличаются не больше, чем на 1.

Хотим научиться поддерживать ±баланс, не храня много дополнительной информации (такой, как атрибуты red/black) — в идеале, $\mathcal{O}(1)$ дополнительных данных, какие-нибудь несколько чисел про дерево в целом. Это умеют две структуры.

2 Scapegoat tree

Источники: [**galperin1993scapegoat**; **andersson1989improving**]. *Мы в основном опираемся на* [**galperin1993scapegoat**].

2.1 Структура дерева

Зафиксируем константу $\frac{1}{2} < \alpha < 1$. Будем рассматривать структуру данных, в которой хранится дерево tree. Также будем хранить текущее количество узлов в дереве — size. У каждого узла node есть дети left, right и ключ key.

structure TREE
    root
    size
    maxSize
structure NODE
    left, right
    key

Мы хотим, чтобы глубина дерева была $\mathcal{O}(\log n)$, где n — количество узлов в дереве. Для этого заведем несколько условий

condition α-WEIGHT(node x)
    $\max\{\text{size}(x.\text{left}), \text{size}(x.\text{right})\} \leq \alpha \cdot \text{size}(x)$
condition α-HEIGHT(node x)
    $depth(x) \leq \lfloor \log \text{size} \rfloor + 1$
condition WEAK α-HEIGHT(node x)

$depth(x) \leq \lfloor \log \text{maxSize} \rfloor + 1$ $\triangleright$ maxSize will be defined later

Желаемая максимальная высота дерева (n — количество узлов с ключами) — $\mathcal{O}(\log_{\frac{1}{\alpha}} n)$.

Если $\alpha = \frac{1}{2}$, то результатом будет идеально сбалансированное дерево, то есть α — это, грубо говоря, разрешённое отклонение размера поддеревьев от состояния баланса.

Узел называется *глубоким,* если он нарушает weak α-height condition. Глубокие узлы мы не любим и каждый раз, когда они у нас будут появляться, мы будем переподвешивать часть дерева так, чтобы они переставали быть глубокими.

Заметим, что если дерево α-weight balanced, то оно и α-height balanced. Обратного следствия нет, потому что может быть «один сын справа, а слева сбалансированное поддерево».

Иногда мы будем перестраивать все дерево. Чтобы реализовать вставку и удаление, нам также потребуется хранить величину maxSize для всего дерева tree. maxSize — штука, отвечающая какой максимальный размер был у дерева с момента последней его полной перестройки. (То есть, кроме собственно дерева с ключами, мы храним дополнительно только size и maxSize — два числа.) Также, нам понадобится еще один инвариант для нашего дерева

Инвариант: $\alpha \cdot \text{maxSize} \leq \text{size} \leq \text{maxSize}$

Заметим, что из этого инварианта следует, что глубина дерева без глубоких вершин не превосходит $\mathcal{O}(\log n)$.

Удаление: просто удаляем. Проверяем, не нарушился ли инвариант. Если нарушился — просто перестроим всё дерево с нуля, сделав массив с ключами за линию и соорудив из него идеально сбалансированное дерево. size при этом уменьшается на 1, а maxSize = size.

Вставка: сначала стандартная вставка, добавляем ключ в лист. При этом size увеличивается на 1,

$$\text{maxSize} \coloneqq \max\{\text{maxSize}, \text{size}\}.$$

Может, однако, оказаться так, что новый узел x оказался глубоким. Тогда рассмотрим путь от x до корня $a_0 \dots a_H$ и найдём среди этих узлов (просто за линию, посчитав количество) самый нижний, не сбалансированный по весу (такой найдётся, докажем) и перестраиваем (глупо, за линию) дерево под ним.

Theorem 1. *Среди $a_0 \dots a_H$ всегда найдётся узел, не сбалансированный по весу (козёл отпущения).*

Доказательство. Пусть нет, тогда $\text{size}(a_i) \leq \alpha \cdot \text{size}(a_{i+1})$. Тогда $\text{size}(x) \leq \alpha^H \cdot \text{size}(T)$. Прологарифмируем это неравенство по основанию $\frac{1}{\alpha}$:

$$0 \leq -H + \log_{\frac{1}{\alpha}} n$$

□

Theorem 2. *При вставке элемента сохраняется сбалансированность по высоте.*

Доказательство. Интересен только случай, когда вставленный элемент глубокий. Достаточно показать, что при перестройке глубина перестроенного поддерева уменьшится. Заметим, что у нас в каждый момент времени бывает не более одного глубокого элемента (при вставке может появится только один, вот-вот вставленный, а при удалении maxSize меняется только если все дерево было перестроено), значит, глубина поддерева может остаться прежней тогда и только тогда, когда выбранное поддерево состояло из полного поддерева с добавленным к нему одним глубоким элементом. Но такое поддерево удовлетворяет условию сбалансированности по весу, а значит, мы его не могли выбрать. □

Корректность мы показали, но у нас остались операции перестройки, которые работают в худшем случае за линию. Покажем, что они хорошо амортизируются.

2.2 Время работы

Сначала разберемся с перестройкой дерева при удалении. Эта операция линейна и происходит не чаще, чем раз в $\alpha \cdot \mathrm{size}(T)$ операций удаления, а значит, имеет ее амортизированная сложность $\mathcal{O}(1)$.

Осталась операция перестройки нижнего несбалансированного поддерева при вставке. Пусть корень этого дерева — x. У этого поддерева есть больший ребенок (не умаляя общности будем считать, что он левый) и меньший (соответственно, правый). Рассмотрим все операции вставки в левое поддерево и удаления из правого поддерева с момента последней перестройки какого-либо родителя x. Для того, чтобы x перестал быть сбалансированный по высоте, их количество должно быть хотя бы линейно от $\mathrm{size}(x)$. Сопоставим все эти операции перестройке дерева. Заметим, что каждая вставка и удаление была сопоставлена не более чем $\mathcal{O}(\log n)$ перестройкам, значит, амортизированная сложность этих операций не увеличилась. При этом каждой перестройке мы сопоставили линейное количество вставок и удалений, значит, амортизированная сложность всех перестроек не превосходит $\mathcal{O}(1)$.

Таким образом, операции вставки у удаления работают за амортизированное время $\mathcal{O}(\log n)$.

3 Splay tree

Оригинальная статья: **[tarjan1985splay]**

3.1 Общая структура дерева

В этом дереве мы каждый раз, когда захотим что-то сделать с вершиной, будем поднимать ее до корня (операция splay). В самом дереве в этот раз мы можем не хранить ничего, кроме корня root. Но часто хочется уметь быстро считать размер дерева, для этого можно хранить отдельную переменную size для всего дерева.

```
structure TREE
    root
    size                                          ▷ optional
structure NODE
    left, right
    key
```

Выразим сначала операции insert и erase через операцию splay, а потом будем разбираться со splay. Для erase нам понадобится операция splay_front(node). Эта операция делает splay для наименьшего ключа в поддереве.

```
procedure INSERT(x)
    standard_insert(x)
    splay(x)
procedure GET(x)
    splay(x)
procedure ERASE(x)
    splay(x)
    splay_front(root.right)
    standard_erase(x)
```

Два вызова функции splay при удалении нужны для того, чтобы правый сын корневой вершины не имел левого сына (потому что он содержит наименьший ключ в своем поддереве) и операция standard_erase(x) работала за $\mathcal{O}(1)$ (потому что она просто возьмет этого правого сына и поставит на место удаленного корня). Еще стоит отметить, что даже при простом доступе к вершине мы вызываем операцию splay, это нужно потому что наше дерево может иметь довольно большую глубину во время работы, а оценка у нас будет только на амортизированную сложность операции splay.

Ниже мы будем оценивать сложность splay при фиксированном множестве ключей в дереве, покажем, что этого достаточно. Удаление вершины из дерева испортить время работы очевидно не сможет, а при добавлении мы спускаемся на полную глубину дерева и можно считать, что искомая вершина была в дереве всегда, просто мы ее не трогали до момента добавления. Тут стоит обратить внимание на то, что с таким подходим, если у нас был какой-то ключ, мы его удалили, а потом добавили обратно, то в оценке времени работы их надо рассматривать как два различных ключа.

3.2 Splay

Итак, нам надо научиться понимать вершину в корень. Это делается при помощи нескольких видов вращений дерева. Все вращения в дальнейшем будем рассматривать с точностью до симметрии. Простейшее вращение называется zig (см. рис. 1). Легко видеть, что это вращение поднимает вершину x на один уровень выше. При помощи одного этого вращения можно поднять вершину в корень, но для амортизационного анализа нам этого не хватит, поэтому мы будем делать сразу двойные вращения.

Двойные вращения бывают двух видов: zig-zig (рис. 2) и zig-zag (рис. 3). Оба эти вращения реализуются при помощи пары вращений zig, но для того, чтобы выразить

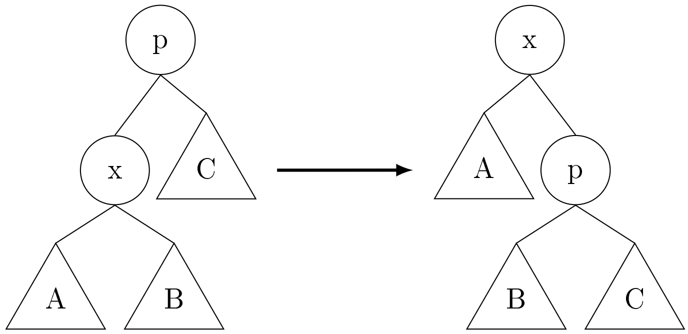


Рис. 1: Zig

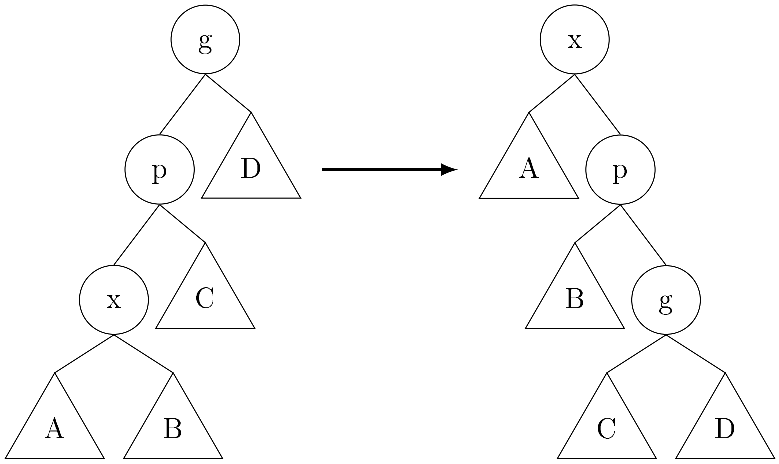


Рис. 2: Zig-zig

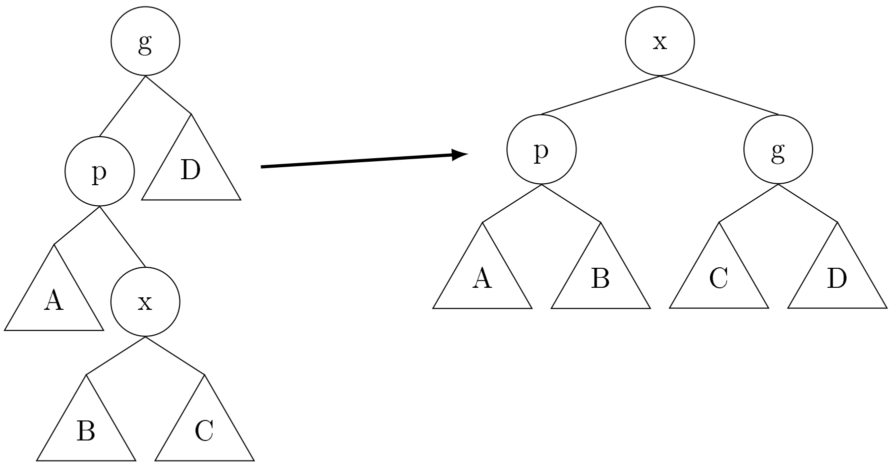


Рис. 3: Zig-zag

zig-zig, надо сначала выполнить zig от вершины p, и только потом от x. Zig-zag при этом выражается как два вызова zig от x. Стоит отметить, что при splay мы не сможем выполнить двойное вращение, если интересующая нас вершина непосредственный сын корня, тогда мы должны сделать zig и не забыть его посчитать при анализе (но он может быть только один).

Для анализа, мы воспользуемся методом потенциалов. Для начала заведем функцию $w\colon \text{keys} \to \mathbb{R}_+$. На нее тоже будут какие-то условия. Про то, какой она может быть, поймем позже, пока можно считать, что она всегда возвращает 1, реально менять ее придется только для следствий. Определим функцию «размера» поддерева $s(x) = \sum_{v \in \text{subtree of } x} w(v)$ и функцию «ранга» $r(x) = \log_2 s(x)$ (логарифм двоичный, это неожиданно важно, но дальше основание писать не будем), а функцией потенциала всего дерева T будет $\Phi(T) = \sum_{x \in T} r(x)$. Для того, чтобы метод потенциалов работал, нужно чтобы Φ всегда было неотрицательно (ну или придется оценить оценить насколько сильно оно бывает отрицательным и прибавить к асимптотике). При $w \equiv 1$ это очевидно, а вообще это надо запомнить как первое ограничение на w. Амортизированная стоимость операции splay am.cost $= \Delta\Phi + \#\text{rotations}$ (да, это просто определение). Пусть мы выполнили один splay. Теперь $r(x)$ и $s(x)$ будут обозначать значения до вызова операции, а $r'(x)$ и $s'(x)$ — после. Тогда на самом деле мы хотим доказать следующую теорему:

Theorem 3. *am.cost* $\leq 3(r'(x) - r(x)) + \mathcal{O}(1)$

Доказательство. Надо оценить $\Delta\Phi$ для каждого из вращений.

Zig:

$$\begin{aligned}
\Delta\Phi &= r'(p) - r(p) + r'(x) - r(x) \\
&= r'(p) - r(x) && \text{since } r'(x) = r(p) \\
&\leq r'(x) - r(x) && \text{since } p \text{ is lower than } x \text{ after zig}
\end{aligned}$$

Дополнительно стоит отметить, что $r'(x) \geq r(x)$ поскольку слева написана сумма по большему множеству, поэтому если мы вдруг захотим это умножить на какую-нибудь произвольно взятую константу 3, ничего не испортится.

Zig-zig:

$$\begin{aligned}
\Delta\Phi &= r'(g) - r(g) + r'(p) - r(p) + r'(x) - r(x) \\
&= r'(g) + r'(p) - r(p) - r(x) \\
&\leq r'(g) + r'(x) - 2r(x) && \text{due to the tree structure} \\
&\leq 3(r'(x) - r(x)) - 2 && \text{since } r'(g) + r(x) \leq 2(r'(x) - 1)
\end{aligned}$$

Осталось показать почему $r'(g) + r(x) \leq 2(r'(x) - 1)$.

$$
\begin{aligned}
\frac{r'(g)+r(x)}{2} &= \log s'(g) + \log s(x) \\
&\le \log\left(\frac{s'(x)-w(p)}{2}\right) && \text{Jensen's inequality} \\
&= \log\bigl(s'(x)-w(p)\bigr) - 1 \\
&\le r'(x) - 1
\end{aligned}
$$

Zig-zag:

$$
\begin{aligned}
\Delta\Phi &= r'(g) - r(g) + r'(p) - r(p) + r'(x) - r(x) \\
&= r'(g) + r'(p) - r(p) - r(x) \\
&\le r'(g) + r'(p) - 2r(x) && \text{due to the tree structure} \\
&\le 3(r'(x) - r(x)) - 2 && \text{since } r'(g) + r'(p) \le 2(r'(x) - 1)
\end{aligned}
$$

Доказательство неравенства $r'(g) + r'(p) \le 2(r'(x) - 1)$ в точности повторяет доказательство аналогичного неравенства выше.

Изменения потенциала от каждого двойного вращения мы оценили как $3(r'(x) - r(x)) - 2$. Все наши страдания были на самом деле направлены на то, чтобы получить двойку в конце. Теперь, когда мы просуммируем по всем вращениям при операции splay, мы получим оценку $\Delta\Phi \le 3(r'(x) - r(x)) - \#\text{rotations} + \mathcal{O}(1)$, поскольку все промежуточные $r(x)$ скомпенсируются, zig будет вызван не более одного раза, а в оценке двойных вращений есть слагаемое -2, которые просуммируются в количество вращений. Таким образом, $\text{am.cost} = \Delta\Phi + \#\text{rotations} \le 3(r'(x) - r(x)) + \mathcal{O}(1)$, что нам и надо. □

Ниже мы будем считать, что наше дерево работает с ключами $1 \dots n$, выполняет m операций, а $W \coloneqq \sum_i w(i)$. Теперь нам надо выбрать w. Надо вспомнить какие условия ограничения мы насобирали на w. Ограничения у нас появлялись в двух местах: из определения $w > 0$ (потому что мы потом хотим логарифмировать) и из метода потенциалов $\Phi \ge 0$. При $w \ge 1$ потенциал неотрицателен автоматически, поскольку все слагаемые неотрицательны.

Corollary 4 (Balance Theorem)**.** *Амортизированное время работы на любой последовательности из m запросов $\mathcal{O}(m \log n + n \log n)$.*

Доказательство. Берем $w(x) = 1$. □

Corollary 5 (Static Optimality Theorem)**.** *Пусть q_x — количество доступов к элементу x. Тогда амортизированное время работы $\mathcal{O}\Bigl(m + \sum_x q_x \log\Bigl(\frac{m}{q_x}\Bigr)\Bigr)$.*

Доказательство. Берем $w(x) = q_x$. □

Из этой теоремы следует, что splay деревья работают не хуже (с точностью до константного множителя, конечно), чем оптимальное статическое дерево поиска. Аналогичное утверждение про динамические деревья остается открытой проблемой.

Conjecture 6 (Dynamic Optimality Conjecture)**.** *Пусть A — произвольное двоичное дерево поиска, которое может делать некоторые вращения (Zig, рис. 1), и обрабатывать запрос на доступ к вершине за ее глубину. Обозначим $A(S)$ — время работы A на последовательности запросов S. Тогда время работы splay дерева на последовательности S не превосходит $\mathcal{O}(n + A(S))$.*

Для следующего следствия стоит вспомнить, что мы считаем, что элементы $1 \ldots n$.

Corollary 7 (Static Finger Theorem)**.** *Пусть f — некоторый фиксированный элемент, «finger». Тогда время работы $\mathcal{O}\Big(m + n \log n + \sum_{x \text{ — запрос}} \log(|x - f| + 1)\Big)$.*

Доказательство. Берем $w(x) = \frac{1}{(|x-f|+1)^2}$. Тогда $W = \mathcal{O}(1)$, а потенциал может быть отрицательным, но не больше, чем на $\mathcal{O}(n \log n)$, поскольку $w \geq \frac{1}{n^2}$, это слагаемое мы можем просто искусственно добавить к потенциалу и, следовательно, асимптотике. □

Corollary 8 (Dynamic Finger Theorem)**.** *Аналогично, но теперь f, «finger» — элемент, к которому обращались предыдущим запросом (и, следовательно, находящийся в корне). Тогда время работы $\mathcal{O}\Big(m + n + \sum_{x \text{ — запрос}} \log(|x - f| + 1)\Big)$.*

Доказательство. Надо бы как-нибудь доказать. □ todo 1

Theorem 9 (Scanning Theorem or Sequential Access Theorem or Queue theorem)**.** *Доступ к элементам в порядке возрастания работает за амортизированную единицу на запрос.*

Доказательство. Следует из Dynamic Finger Theorem (Corollary 8). □

4 Об оффлайн деревьях поиска: введение

[Я пишу здесь свою чепуху, а потом пытаюсь понять, как она вклеивается в уже написанное]

Исходная статья: “The Geometry of Binary Search Trees” [**demaine2009geometry**].

4.1 Оффлайн деревья поиска и графическое представление

Пусть у нас есть дерево над ключами 1, 2, …, n и последовательность $S = (s_1, s_2, \ldots, s_m)$ запросов поиска. Запросы нам известны заранее и мы хотим построить такое дерево, чтобы минимизировать суммарное время, потраченное на то, чтобы ответить на эти запросы. При этом мы разрешаем перестраивать дерево с помощью вращений в процессе ответа на запросы.

Нам разрешено делать следующие вещи:

1) Переходить по указателю.

2) Делать одинарное вращение с центром в этой вершине (zig, он же zag).

Начинаем при этом мы всегда в корне, а для каждого запроса хотим посетить вершину с соответствующим ключом.

Definition 2. Последовательность таких действий для фиксированной последовательности запросов S называется *BST-алгоритмом*.

Definition 3. Цена операции поиска — количество посещённых узлов. Поскольку для того, чтобы сделать вращение в вершине, нам нужно её посетить, учитывать количество вращений не нужно, если мы считаем с точностью до константы.

Definition 4. $\mathrm{OPT}(S)$ — минимальная суммарная цена выполнения S, если мы заранее знаем S и можем в связи с этим выбирать, какие именно операции вращения мы будем делать, а какие — нет (а также можем выбрать, как дерево выгляд

Значение $\mathrm{OPT}(S)$ мы не умеем искать (даже с точностью до мультипликативной константы) за полином. Впрочем, опровергать возможность вычисления $\mathrm{OPT}(S)$ за полином мы тоже не умеем, даже в предположении $\mathsf{P} \neq \mathsf{NP}$. Ясно, что задача о проверке неравенства $\mathrm{OPT}(S) \leqslant k$ лежит в NP.

Это можно переформулировать в геометрических терминах. Рассмотрим координатную плоскость с ключами по оси x и моментами времени (то есть номерами запросов) по оси y. Для данной последовательность запросов S отметим все точки (s_i, i) на плоскости (мы обязаны посетить ключ s_i при обработке i-того запроса, так как мы должны его найти). Также отметим все точки (k, i) такие, что мы посетили ключ k при обработке i-того запроса. Понятно, что стоимость данной последовательность операций — количество отмеченных точек.

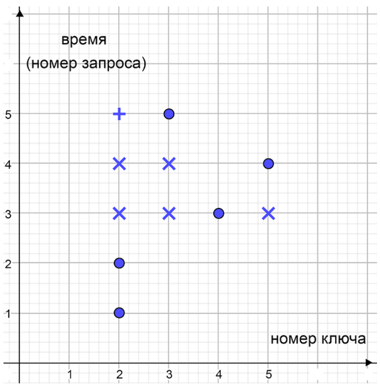
Definition 5. Множество отмеченных точек — *графическое представление* данного BST-алгоритма. У разных BST-алгоритмов могут быть одинаковые представления.

На рисунку 4a видно графическое представление, которое получится, если для дерева с рисунка 4b применить последовательность запросов $S = (2, 2, 4, 5, 3)$ и при этом не совершать никаких вращений. От плюсика можно избавиться, если при обработке четвёртого запроса сделать операцию rotate 3 и получить дерево с рисунка 4c (в таком дереве для обработки запроса найти ключ 3 не нужно посещать вершину с ключом 2, так как вершина с ключом 3 уже является корнем).

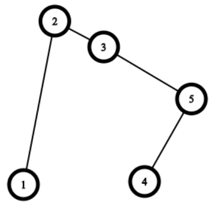
Про splay-деревья верят, что они оптимальны с точностью до мультипликативной константы, то есть что они посещают $O(\mathrm{OPT}(S) + n)$ вершин при обработке любого списка запросов S. Это достаточно круто, так как splay-деревья не знают будущего, в отличие от оптимальное алгоритма. Однако, доказывать это про splay-деревья не умеют, но есть другие деревья, про которые это умеют доказывать.

Definition 6. Если a и b — точки на плоскости, то $\mathrm{rect}(a, b)$ — прямоугольник (возможно, вырожденный), натянутый на точки a и b как на противоположные углы.

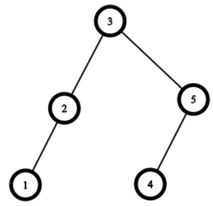
Definition 7. Множество целых точек на плоскости E называется arborally satisfiable, если для любых точек a и b из E верно хотя бы одно из следующих трёх свойств: $x(a) = x(b)$, $y(a) = y(b)$ или $\mathrm{rect}(a, b)$ содержит точку из $E \setminus \{a, b\}$.



(a) Графическое представление BST-алгоритма: жирными точками отмечены сами запросы, крестиками и плюсиком — другие ключи, которые мы посещаем в процессе их обработки. От плюсика можно избавиться, см. основной текст.



(b) Исходное дерево.



(c) Дерево после операции rotate 3.

Рис. 4: Пример графического представления

4.2 Эквивалентность BST-алгоритмов и arborally satisfiable множеств

Theorem 10. *Если множество точек E может быть отмечено каким-то BST-алгоритмом, то оно arborally satisfiable.*

Доказательство. Предположим противное. Пусть мы нашли две точки a и b, при этом $x(a) \neq x(b)$, $y(a) \neq y(b)$ и внутри $\mathrm{rect}(a, b)$ нет других точек E. Не умаляя общности, $i := y(a) < y(b) =: j$. Пусть c — наименьший общий предок a и b *после* обработки i-того запроса. Есть два случая:

1) Если $c = a$, то мы должны были посетить a в какой-то момент из отрезка $(i + 1, j]$. Действительно, раз a является предком b после обработки i-того запроса, то либо a — всё ещё предок b *перед* обработкой j-того запроса (и тогда мы должны посетить a просто для того, чтобы дойти до b), либо вершина a перестала быть предком b в какой-то из моментов на отрезке $(i + 1, j)$, а для этого мы должны были её посетить и сделать вращение в её ребёнке.

2) Если $c \neq a$, то a и b лежат в разных поддеревьях c. Следовательно, по свойству двоичного дерева поиска, $x(c) \in \langle x(a), x(b)\rangle$ (ключ вершины c должен лежать между ключами вершин a и b; здесь $\langle s, t\rangle$ это либо $[s, t]$, если $s < t$, либо $[t, s]$ в противном случае). Раз мы посетили a при обработке i-того запроса, то мы посетили и её предка c, следовательно множество E содержит точку $(x(c), y(c)) = (x(c), i)$, а она лежит в искомом прямоугольнике.

$\square$

Стоит заметить, что мы доказали более сильный факт: если $x(a) \neq x(b)$ и $y(a) \neq y(b)$, то есть точка из $E \setminus \{a\}$, которая попала на одну из сторону $\mathrm{rect}(a, b)$, смежную с

a (то, какая это сторона, зависит от того, $c = a$ или $c \neq a$). Аналогичное утверждение верно для $E \setminus \{b\}$ и b.

Дальше мы будем постоянно пользоваться следующей леммой, утверждающей, что описанное в прошлом абзаце условие верно для любого arborally satisfiable множества, а не только для тех, которые являются графическим представлением BST-алгоритма (позже мы поймём, что каждое arborally satisfiable множество — графическое представление какого-то BST-алгоритма, но не будем торопить события).

Lemma 11. *Если E — arborally satisfiable, то для любых a и b из E, таких что $x(a) \neq x(b)$ и $y(a) \neq y(b)$, существует точка из $E \setminus \{a\}$, которая попадает на одну из сторон* $\mathrm{rect}(a, b)$

Доказательство. Пусть у нас есть $\mathrm{rect}(a, b)$ для точек a и b, удовлетворяющих условиям $x(a) \neq x(b)$ и $y(a) \neq y(b)$. Так как E — arborally satisfiable множество, то есть $c \in \mathrm{rect}(a, b)$, $c \neq a$ и $c \neq b$. Есть два случая:

1) $x(c) = x(a)$ или $y(c) = y(a)$. Тогда c лежит на одной стороне $\mathrm{rect}(a, b)$ с точкой a, то есть c — искомая точка.

2) $x(c) \neq x(a)$ и $y(c) \neq y(a)$. Тогда в $\mathrm{rect}(a, c)$ тоже есть точка из $E \setminus \{a, c\}$ по обычному arborally satisfiable свойству, при этом $\mathrm{rect}(a, c)$ строго меньше $\mathrm{rect}(a, b)$. Будем повторять процесс (возьмём точку $d \neq a$, $d \neq c$ из $\mathrm{rect}(a, c)$, и так далее), пока неизбежно не выполнится случай 1.

□

Немного удивительно, но верна и теорема, обратная к теореме 10: теорема 14. Для её доказательства нам понадобится понимать некоторые базовые вещи про декартовы деревья.

Definition 8. *Декартово дерево* — это двоичное дерево поиска, в котором у каждой вершины кроме ключа есть ещё и другой параметр — *приоритет*, при этом декартово дерево образуют кучу на минимум по приоритетам.

Definition 9. Декартово дерево на парах $(\mathrm{key}_i, \mathrm{priority}_i)$ — любое декартово дерево с данными мультимножеством пар "ключ–приоритет".

Lemma 12. *Всегда есть хотя бы одно декартово дерево на данном наборе пар. Более того, если все ключи и приоритеты в наборе различны, то декартово дерево только одно.*

Доказательство. По свойству кучи корнем декартова дерева должна быть одна из вершин с минимальным приоритетом. Возьмём любую вершину с минимальным приоритетом и сделаем её корнем, пусть её *ключ* равен x. Тогда все вершины с ключами меньше x должны попасть в её левое поддерево, а с ключами больше x — в правое. Куда попадут вершины с ключом ровно x, неважно. Распределим оставшиеся вершины на левое и правое поддерево корня и построим их рекурсивно. Если все ключи и приоритеты были различны, то каждый раз мы делали единственное возможное действие, поэтому в этом случае декартово дерево уникально. □

Также нам понадобится следующая лемма.

Lemma 13. *Пусть у нас есть два двоичных дерева поиска на одном и том же наборе из n ключей. Тогда одно из них можно перестроить в другое, сделав $O(n)$ вращений.*

Доказательство. Самое интуитивное доказательство этого факта использует соответствие между двоичными деревьями поиска на n вершинах и триангуляциями $(n+2)$-угольника.

Соответствие выглядит так: зафиксируем любую сторону многоугольника и назовём её *корневой стороной.* Оставшиеся n вершин, не попавшие на корневую сторону, пронумеруем по циклу числами от 1 до n. Пусть теперь у нас есть

дописать □ todo 2

Theorem 14. *Если E — arborally satisfiable, то существует BST-алгоритм, графическое представление которого в точности равно E. Строго говоря, нужно ещё не забыть наложить условие, что множество y-координат точек из E — в точности отрезок целых чисел $[1, n]$ для какого-то n. Это соответствует тому, что при каждом запросе мы должны обязательно посетить корень, то есть хотя бы одну вершину.*

Доказательство. Обозначим за τ_i множество из ключей, которые нам разрешено посещать на i-том шаге (то есть такие ключи x, что $(x, i) \in E$).

В момент времени i наше дерево будет *каким-то* (не любым, а именно каким-то; то есть "существует последовательность декартовых деревьев", а не "для любой последовательности декартовых деревьев") декартовым деревом на парах $(x, N(x, i))$, где $N(x, i)$ — минимальное такое $j \geqslant i$, что $(x, j) \in E$ или $+\infty$, если таких j нет. Интуитивно, $N(x, i)$ должно быть первым моментом времени, начиная с i, когда мы посетим ключ x. T_1 — какое-то декартово дерево, хотим перестроить T_i в T_{i+1}, посетив только вершины из τ_i.

Вершины, которые мы можем посещать в момент времени i — какой-то связный кусок T_i, содержащий корень T_i. Почему? Потому что у всех вершин T_i приоритет равен $N(x, i)$, то есть хотя бы i, а у вершин из τ_i приоритет равен ровно i. При этом только у вершин из τ_i приоритет поменяется на что-то новое (так как для других ключей $N(x, i) = N(x, i+1)$). Назовём вершины из τ_i *верхними*, а все остальные — *нижними.*

Дерево T_i устроено следующим образом: это какое-то дерево поиска (назовём его *верхней компонентой*) на вершинах из τ_i, только вместо некоторых "пустых детей" (то есть вместо nullptr'ов) подклеены *нижние поддеревья* — поддеревья, целиком состоящие из нижних вершин. Чтобы получить T_{i+1} из T_i, перестроим верхнюю компоненту в соответствии с новыми приоритетами вершин из τ_i (это можно сделать по лемме 13) с помощью вращений. Строго говоря, лемма даёт нам последовательность вращений верхней компоненты, но каждое вращение верхней компоненты естественным образом является вращением и всего дерева тоже. Поскольку мы использовали только вращения, свойство двоичного дерева поиска не начало нарушаться.

Теоретически, могло нарушится условие кучи. Поскольку все верхние вершины остались наверху, а нижние — внизу, то в T_{i+1} может быть лишь три типа пар "родитель-сын", где нарушилось условие кучи: "верх-верх", "низ-низ" и "верх-низ". Нарушений типа "верх-верх" нет, поскольку T_{i+1} строилось так, чтобы для верхних вершин в нём выполнялось условие кучи с новыми приоритетами. Нарушений типа "низ-низ" нет, так как при вращениях вершин из τ_i нижние поддеревья могли как-то переставляться, но при этом их внутренняя структура не менялась, так как мы не трогали нижние вершины. Осталось понять, почему не могло быть нарушений вида "верх-низ". Тут-то нам и пригодится то, что E — arborally satisfiable.

Пусть в T_{i+1} есть нижняя вершина с парой "ключ–приоритет" (y, j) и её родитель — верхняя вершина с парой (x, k). Раз эти вершины нарушают свойство кучи, то $j < k$. Не умаляя общности, $x < y$. Посмотрим на точки (x, i) и (y, j) из E и натянутый на них прямоугольник $\mathrm{rect}((x, i), (y, j))$. На вертикальной стороне от (x, i) до (x, j) нет ничего из $E \setminus \{(x, i)\}$ по определению, так как $N(x, i+1) = k > j$. Следовательно, по лемме 11, есть точка (c, i) на стороне от (x, i) до (y, i). Это значит, что $c \in \tau_i$, то есть c — верхняя вершина.

Все наши операции при перестройке T_i в T_{i+1} были вращениями: они могли сломать свойство кучи, но не свойство двоичного дерева поиска. Поэтому, ключ c всё ещё лежит между ключами x и y. Но вершина с ключом y — родитель вершины с ключом x в T_{i+1}. Следовательно, вершина с ключом c находится где-то в правом поддереве вершины y дерева T_{i+1}. Это невозможно, так как c — верхняя вершина и должна была остаться наверху (но не осталась, так как она отделена нижней вершиной y от верхней вершины x). Противоречие. $\square$

4.3 "Онлайн-эквивалентность" BST-алгоритмов и arborally satisfiable множеств

Только что мы получили оффлайн-алгоритм, который, зная arborally satisfiable множество E, строит BST-алгоритм с графическим представлением E. Утверждается, что есть *онлайн*-алгоритм который, получая не всё E сразу, а по строкам (получил τ_1, сделал нужные операции, получил τ_2, сделал нужные операции, и так далее)), строит BST-алгоритм со стоимостью $O(|E|+n)$ (получить в точности графическое представление E не получится, ухудшения на мультипликативную константу не избежать). Здесь, как и в прошлом разделе, под τ_i понимается множество таких ключей x, что $(x, i) \in E$.

Нам понадобится немного необычная структура данных.

Definition 10. *split-дерево* (split-tree) — это абстрактная структура данных, состоящая из *внутреннего двоичного дерева поиска* на имеющихся ключах и какой-то *дополнительной информации*, которая может иметь любую природу. При этом она должна уметь поддерживать две операции:

1) make_tree(x_1, x_2, $\ldots$, x_n) — по отсортированному массиву ключей построить структуру данных, при этом внутреннее дерево поиска должно быть двоичным деревом поиска на данных ключах;

2) split_tree(x) — найти ключ x во внутреннем двоичном дереве поиска (гарантируется, что он там есть), с помощью вращений поднять его в корень, удалить его и вернуть два новых split-дерева: левое и правое поддеревья корня (в левом все ключи меньше x, а в правом все ключи больше x).

При этом разрешается тратить суммарно только $O(n)$ времени на построение (make_tree) и полное разрушение (n операций split_tree) дерева.

Remark (Небольшое отступление о природе split-дерева.)**.** Операции “постройте структуру по списку чисел” и “найдите данное число в структуре, удалите его и разбейтесь на «до» и «после»” можно легко реализовать с помощью односвязного списка и хэш-таблицы или кучи других подобных методов.

Но суть split-дерева не в этом. Суть split-дерева в том, что оно реализует операцию split_tree *физически* на внутреннем двоичном дереве поиска с помощью вращений в точности так, как описано. Вся дополнительная информация, которую мы храним, существует не для того, чтобы отвечать на какие-то запросы об элементах структуры, а только для того, чтобы лучше понимать, как и когда совершать дополнительные вращения, кроме тех, которые нам нужны, чтобы пригнать ключ x в корень.

Нас интересует не столько время, которые мы потратили, сколько число вершин во внутреннем двоичном дереве, которые мы затронули. Если бы мы могли потратить $O(n^2)$ времени, но затронуть вершины только $O(n)$ раз в процессе полного разрушения внутреннего дерева, это бы нас более-менее устроило. Но оказывается, что мы можем потратить $O(n)$ времени (и, следовательно, лишь $O(n)$ раз затронуть вершины внутреннего дерева). Раз можем, то почему бы и не воспользоваться чуть лучшей версией алгоритма?

Я не буду воспроизводить принцип работы split-дерева, так как он не очень важен. Узнать его можно в исходной статье [**demaine2009geometry**]. Более того, есть гипотеза (см. статью Лукас [**lucas1988canonical**]), что в качестве split-дерева можно использовать обыкновенное splay-дерево без дополнительной информации (и, соответственно, не делать никаких вращений, кроме тех, которые нужны, чтобы пригнать ключ в корень), но доказывать это не умеют.

Теперь мы будем на каждом шаге строить не обычное декартово дерево, а обобщённое декартово дерево (определение в следующем абзаце). G_i — обобщённое декартово дерево (с отличием, что теперь мы просим, чтобы декартово дерево было кучей на *максимум* по приоритетам), построенное на парах $(x, \rho(x, i))$, где $\rho(x, i)$ — максимальное такое $j < i$, что $(x, j) \in E$ или $-\infty$, если таких нет. То есть $\rho(x, i)$ — последний момент строго перед i, когда ключ x был задет. Фактически, мы повернули вспять течение времени и сделали так, что теперь вершины с большими $\rho(x, i)$ находятся выше в дереве (раньше — вершины с меньшими $N(x, i)$). Однако, не всё так просто, так как теперь вершины, у которых мы меняем приоритет расположены внутри дерева, на первый взгляд, как-то случайно.

Definition 11. *Обобщённое декартово дерево* (general treap) — это на самом деле обычное декартово дерево, вершины которого объединены в *суперузлы*:

1) Каждый суперузел — связное подмножество вершин дерева с одинаковым приоритетом. Для одного приоритета может быть несколько суперузлов, но они должны быть не связаны между собой (то есть соседние вершины с одинаковым приоритетом обязаны попасть в один и тот же суперузел).

2) Каждый суперузел — split-дерево (точнее, внутреннее двоичное дерево для split-дерева). Гарантируется, что суперузлы создаются с помощью операции make_tree, а потом постепенно разрушаются с помощью операций split_tree. Так как внутри суперузла все приоритеты одинаковые, то любое двоичное дерево поиска будет удовлетворять условию кучи.

Definition 12. Отношение “отец–сын” на суперузлах определяется естественным образом. При этом у суперузла (в отличие от обычной вершины) может быть много детей.

Remark. Важно понимать, что суперузлов в некотором смысле “не существует” — обобщённое декартово дерево не является двумерной структурой данных в любом привычном смысле слова. Как уже было сказано, обобщённое декартово дерево по своей внутренней структуре — обычное декартово дерево. Суперузлы мы выделили сами, чтобы внутри них строить split-деревья и пользоваться амортизированной оценкой из определения split-деревьев.

На это можно смотреть так: все наши ключи различны, но среди приоритетов есть одинаковые, и их достаточно много. Поэтому в структуре декартова дерева есть некоторая свобода, связанная с тем, как именно мы располагаем соседние вершины с одинаковым приоритетом. Мы воспользовались этой свободой, чтобы построить в каждом суперузле split-дерево, которое будет нам говорить, как именно расположены вершины этого суперузла.

Изначально, G_1 состоит из одного суперузла с приоритетом $-\infty$, который мы строим с помощью make_tree. Как получить G_{i+1} из G_i? Для этого нужно взять все вершины из τ_i, так как только для них $\rho(x, i+1) \neq \rho(x, i)$ (а именно, $\rho(x, i+1) = i$ для $x \in \tau_i$; для других вершин $\rho(x, i+1) = \rho(x, i) < i$), вырезать их из своих суперузлов с помощью split_tree и собрать из них новый суперузел с приоритетом i с помощью make_tree. Строгое понимание этих слов (в частности, то, как мы поддерживаем при всех этих операциях свойства обобщённого декартово дерева и даже то, почему G_{i+1} вообще окажется хотя бы обычным декартовым деревом) отложим на потом. А пока сделаем ещё одно полезное замечание о split-деревьях.

Remark. Split-дерево в процессе своей работы производит какие-то операции вращения внутреннего двоичного дерева поиска. Но эти операции вращения (выполнить вращения в какой-то вершине) можно совершать и со связными подмножествами большего дерева поиска, в нашем случае обобщённого декартова дерева.

То есть внутренние деревья поиска наших split-деревьев — какие-то куски нашего обобщённого декартового дерева, а именно, суперузлы. В частности, эти внутренние деревья не нужно хранить отдельно, так как они сохранены в нашем декартовом дереве. Когда split-дерево говорит, что нам нужно сделать вращение во внутреннем дереве, мы делаем вращение в соответствующей вершине нашего большого дерева.

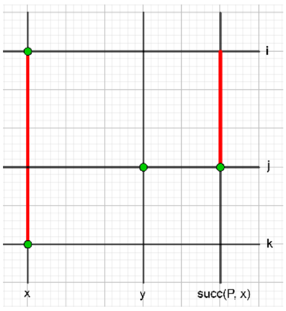


Рис. 5: Зелёные точки соответствуют точкам из E, красные отрезки — отрезкам, на которых точно нет точек из E из-за того, что мы знаем значения $\rho(\cdot,\cdot)$. Точка (y, j) появляется из леммы 11 для (x, i) и $(\mathrm{succ}(P, x), j)$.

При таком понимании у удаления вершины (операции split_tree(x)) появляется следующая интерпретация: мы не столько *удаляем* вершину с ключом x, сколько пригоняем её в корень куска большого дерева, соответствующего нашему split-дерева, прекращаем ассоциировать её с нашим split-деревом и чисто формально (дополнительную информацию, если она есть, нужно будет пересчитать, но менять в структуре большого дерева ничего не надо) разбиваем наше split-дерево на два.

После такой операции наша вершина перестаёт быть ассоциированой с *каким-либо* split-деревом, поэтому мы больше не будем делать вращений с центром в ней до того, как перейдём к $(i + 1)$-ой итерации процесса (построению G_{i+2} по G_{i+1}).

Однако, как мы поймём позже, это не помешает ей дойти до корня

Нам понадобится следующая лемма:

Lemma 15. *Пусть мы посетили (то есть $x \in \tau_i$) вершину с парой "ключ–приоритет" (x, k) в G_i. Пусть отец её суперузла (в G_i, всё пока в G_i) — суперузел P с приоритетом $j > k$. Пусть $\mathrm{succ}(P, x)$ — наименьший ключ в P, больший x, $\mathrm{pred}(P, x)$ — наибольший ключ в P, меньший x. Утверждается, что мы их посетили (если они существуют), то есть $\mathrm{succ}(P, x) = +\infty$ или $\mathrm{succ}(P, x) \in \tau_i$, и, аналогично, $\mathrm{pred}(P, x) \in \tau_i \cup \{-\infty\}$.*

Доказательство. Действительно, пусть $\mathrm{succ}(P, x) < +\infty$ и $\mathrm{succ}(P, x) \notin \tau_i$, откуда $(\mathrm{succ}(P, x), j + 1), (\mathrm{succ}(P, x), j + 2), \ldots, (\mathrm{succ}(P, x), i) \notin E$. Тогда, так как $(x, i) \in E$ и $(\mathrm{succ}(P, x), j) \in E$, то в $E \setminus \{\mathrm{succ}(P, x)\}$ есть точка на стороне $(\mathrm{succ}(P, x), j)$–(x, j) прямоугольника $\mathrm{rect}((x, i), \mathrm{succ}(P, x))$. Следовательно, есть такое $y \in [x, \mathrm{succ}(P, x))$, что $(y, j) \in E$. Так как $\rho(x, i) = k < j$, то $(x, j) \notin E$, откуда y лежит строго между x и $\mathrm{succ}(P, x)$.

Получается, что $\rho(y, i) \geqslant j$. Почему это странно? Мы уже знаем, что ключ y каким-то образом лежит между ключом $\mathrm{succ}(P, x)$ и ключом x. Это означает, что он лежит в поддереве наименьшего общего предка $\mathrm{succ}(P, x)$ и y. Этот наименьший общий предок — какая-то вершина из P (потому что $\mathrm{succ}(P, x)$ лежит в P по определению, а P — отец суперузла, в котором лежит x), следовательно его приоритет *не больше* приоритета P, то есть j. С другой стороны, мы уже доказали, что $\rho(y, i) \geqslant j$. Отсюда, приоритет y в точности равен j и y лежит в P. Но это противоречит определению $\mathrm{succ}(P, x)$, так как $x < y < \mathrm{succ}(P, x)$. Аналогично с $\mathrm{pred}(P, x)$.

□

Что мы получили? Как минимум то, что множество посещённых *суперузлов* (то есть таких суперузлов, в которых есть вершина из τ_i), образуют связное множество, содержащее корень. Аналогичное утверждение про *вершины* неверно, но оно нам и не понадобится. Сейчас нам понадобится понять ещё одну интересную особенность split-дерева.

Наша цель состоит в том, чтобы небольшим количеством вращений пригнать все ключи из τ_i в какое-то связное множество вершин, содержащее корень, не сломав при это условие кучи на других вершинах. После этого мы вызываем make_tree от этих вершин и делаем их приоритеты равными i. Внутреннее двоичное дерево, которое нам вернёт make_tree может отличаться от структуры двоичного дерева поиска на этих вершинах, которая получилась после того, как мы пригнали их всех наверх, но, как мы знаем, мы можем переделать одно в другое за линейное количество вращений.

Как мы пригоняем все вершины наверх? На удивление просто: пройдёмся по суперузлам в порядке от более глубоких к менее глубоким. Внутри каждого суперузла пройдёмся (скажем, в порядке возрастания, но это должно быть неважно) по всем ключам x из этого суперузла, попавшим в τ_i и для каждого из них сделаем операцию split_tree(x).

Почему это работает? Внутри каждого суперузла первый рассмотренный ключ приедет в корень суперузла, второй — в один из корней двух внутренних деревьев, полученных из исходного, то есть в одного из детей корня суперузла, и так далее. То есть все рассмотренные ключи в итоге приедут в какое-то связное множество, содержащее корень суперузла. Таким образом, каждый ключ "доезжает" до корня своего суперузла "своим ходом".

Однако, как мы уже отметили ранее, мы перестаём делать вращения в вершине после того, как она приехала наверх своего суперузла. Раз сама она дальше проехать не может, то её должны дальше довезти друзья (звучит позитивно)!

Чтобы понять главную идею, рассмотрим случай, когда мы затрагиваем всего два суперузла: суперузел-корень (назовём его P) и одного из его суперузлов-сыновей. При этом в сыне мы затронули только один ключ x. Сперва мы пригоняем ключ x в корень суперузла сына. После этого ключи из P, вместе с ключом x образуют правильное двоичное дерево поиска. По лемме 15, $\mathrm{succ}(P, x)$ и $\mathrm{pred}(P, x)$ лежат в τ_i. Когда мы пригоняем вершины из $\tau_i \cap P$ мы на самом деле разбиваем все оставшиеся вершины из P на поддеревья в зависимости от того, как они сравниваются с вершинами из $\tau_i \cap P$. Но теперь в одном из этих поддеревьев появляется гостья, которой раньше не было: вершина с ключом x. Это поддерево раньше было пустым, так как соответствовало вершинам из P с ключами из интервала $(\mathrm{pred}(P, x), \mathrm{succ}(P, x))$, а таких нет по определению pred и succ. А теперь в этом поддереве будет одна вершина с ключом x. Значит, её отец лежит в верхнем связном куске, состоящем из вершин с ключами из $\tau_i \cap P$. Значит, мы можем подклеить вершину с ключом x к этому куску с сохранением связности.

В общем случае, происходит следующее: внутри каждого суперузла затронутые (то есть из τ_i) вершины этого суперузла собираются в одну большую группу наверху "своим ходом". Более того, все группы, пришедшие из суперузлов-детей тоже подклеятся к этой большой группе. В итоге все эти группы постепенно едут наверх и постепенно склеиваются, в итоге склеиваясь в один большой снежный ком в самом верху большого дерева. Мы это, собственно и хотели доказать.

G_{i+1} — действительно обычное декартово дерево, так как все вращения внутри split-деревьев сохраняют свойство кучи. Поэтому условие кучи могло нарушиться только для вершин, которые мы удаляли из split-деревьев, а они все приехали наверх и получили наибольший приоритет.

Перед тем, как мы объявим доказательство законченным, есть ещё одна тонкость: почему в процессе сделанных нами вращений два разных суперузла для одного и того же приоритета не стали соседними и из-за этого склеились? В формулировке леммы 15 мы неявно пользуемся тем, что у соседних суперузлов разный приоритет, так что замести это под ковёр не получится.

Всего мы сделали $O(|E| + n)$ вращений. Действительно, на i-том шаге мы делаем $|\tau_i|$ операций split_tree (амортизированно $O(1)$ времени) и одну операцию make_tree на $|\tau_i|$ вершинах ($O(|\tau_i|)$ времени). Слагаемое $O(n)$ появляется из-за амортизации: каждое не разрушенное полностью split-дерево могло "съесть" $O(\texttt{своего размера})$ операций. Проще всего это понять, когда $|E| = 1$: хоть первая операция make_tree и "бесплатна" (так как мы вольны выбирать, как дерево выглядит до всех запросов), она могла теоретически вернуть нам бамбук и заставить нас сделать $(n-1)$ вращение уже на первой итерации алгоритма).

5 Об оффлайн деревьях поиска: нижняя граница времени работы, геометрическое представление

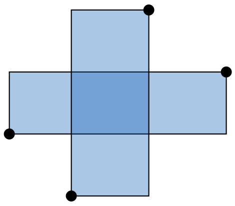
5.1 Основные определения и предваряющие результаты

Пусть дано бинарное дерево поиска с n ключами. Мы знаем последовательность запросов, которые зададим этому дереву: $P = \{s_1, s_2, \ldots, s_m\}$. В поисках ключей s_i мы будем бегать по дереву туда-сюда и в процессе спуска/подъёма пройдём через некоторые вершины, которые нам не нужны.

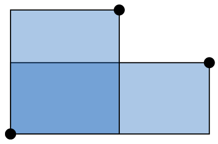
Definition 13. $E(P)$ — множество всех вершин, которые мы посетим в процессе поиска вершин с ключами из P. $E = P \cup X$, X — множество «лишних» вершин.

Definition 14. OPT — минимальный размер $E(P)$ (обозначение множества P будем опускать, и так по контексту ясно).

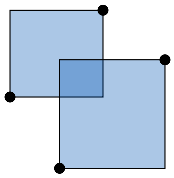
Definition 15. Пусть $a, b \in \mathbb{R}^2$. Тогда $\mathrm{rect}(a, b)$ — прямоугольник, стороны которого параллельны осям координат, а противоположные вершины — точки a и b. Его же будем называть *прямоугольник, определённый точками a, b.*



(a) Эти прямоугольники независимы



(b) Эти прямоугольники независимы



(c) Эти прямоугольники **не** независимы

Рис. 6: Примеры прямоугольников, независимых и не очень

Definition 16. Конечное множество $G \subset \mathbb{R}^2$ называется *arborally satisfiable,* если

$$\begin{aligned}&\forall\, a, b \in G \quad x(a) = x(b), \ \text{ либо } \ y(a) = y(b), \ \text{ либо}\\ &\exists\, c \in \operatorname{rect}(a, b) \ \ \text{(внутри или на границе)}.\end{aligned}$$

Theorem 16 (Доказана ранее)**.** *Рассмотрим последовательность запросов*

$$\{(s_1, 1), (s_2, 2), \ldots, (s_m, m)\} \subset \mathbb{Z}^2.$$

Надмножество этой последовательности может представлять из себя последовательность узлов, которые были посещены при поиске $s_1, \ldots, s_m$, в том и только том случае, если оно arborally satisfiable.

Далее мы будем рассматривать изображение последовательности запросов на плоскости, соответственно под множеством P будем понимать $\{(s_1, 1), (s_2, 2), \ldots, (s_m, m)\}$, аналогично вторую координату приделаем к ключам вершин из множества E.

Definition 17. Пусть дано множество P и его надмножество E. Два прямоугольника, определённых каждый двумя вершинами множества P, будем называть *независимыми* (смотреть Рисунок 6), если

1) они оба не arborally satisfiable, то есть им не принадлежит ни одна точка из E,

2) ни одна из вершин одного из этих прямоугольников не лежит во внутренности другого.

5.2 Оценка снизу числа OPT

Definition 18. Будем говорить, что прямоугольник, определённый точками (x_1, y_1), (x_2, y_2), *имеет тип «+»,* если $(x_1 - x_2) \cdot (y_1 - y_2) \geq 0$, иначе прямоугольник *имеет тип «−»* (смотреть Рисунок 7).

Рис. 7: Прямоугольники типа «+» и типа «−».

Definition 19. MAX IND — наибольшее число попарно независимых прямоугольников, определённых точками из P. Соответственно, $\mathrm{MAX\,IND}_+$, $\mathrm{MAX\,IND}_-$ — наибольшие количества попарно независимых прямоугольников фиксированного типа.

Theorem 17.
$$\mathrm{OPT} \;\geq\; |P| + \frac{1}{2}\,\mathrm{MAX\,IND}\,. \tag{1}$$

Прежде чем приступить к доказательству Теоремы 17, докажем следующую лемму:

Lemma 18.
$$\mathrm{OPT}_+(P) \;\geq\; |P| + \frac{1}{2}\,\mathrm{MAX\,IND}_+(P). \tag{2}$$

Здесь OPT_+ — количество точек в множестве $E(P)$, нужное для того, чтобы множество всех прямоугольников типа «+» было arborally satisfiable. Это более слабое условие.

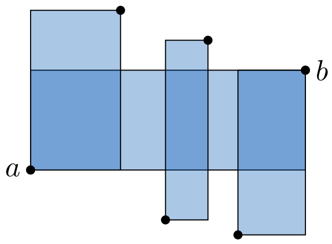
Далее мы забываем о том, что множества точек, с которыми мы работаем, — это вообще говоря выходы какой-то процедуры поиска, и рассматриваем произвольные конечные множества точек на плоскости.

Доказательство Леммы 18. Пусть все координаты точек из P различны (точки можно чуть-чуть пошевелить, чтобы это стало так и ничего больше не нарушилось). Рассмотрим максимальный набор попарно независимых «+»-прямоугольников и самый широкий из них — пусть он определён точками a, b. Некоторые прямоугольники будут пересекать наш самый широкий прямоугольник, одной из их вершин может быть a или b, либо их определяющие вершины будут лежать за границами самого широкого прямоугольника, одна выше, одна ниже, смотреть Рисунок 8a.

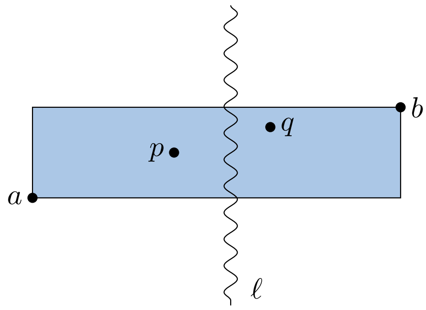
Прямоугольники, имеющие своей вершиной a, не пересекаются с прямоугольниками, имеющими своей вершиной b, потому что иначе получается случай прямо как на Рисунке 6c. Более того, оставшиеся прямоугольники тоже не могут никак налезать друг на друга, потому что опять же получится случай с Рисунка 6c. Поэтому существует вертикальная линия, пересекающая *только* выбранный нами широкий прямоугольник $\mathrm{rect}(a, b)$, обозначим её через ℓ, смотреть Рисунок 8b.

Рассмотрим самую верхнюю, самую правую точку из $E(P)$, которая левее ℓ и принадлежит $\mathrm{rect}(a, b)$, обозначим её через p. Такая точка точно существует, потому что как минимум a подойдёт, мы выбираем из непустого множества. Рассмотрим самую нижнюю, самую левую точку из $E(P)$, которая правее ℓ, принадлежит $\mathrm{rect}(a, b)$ и не ниже p, обозначим её через q. Опять же такая найдётся, потому что есть b.

Claim 19. *Точки p и q лежат на одной горизонтали.*



(a) Прямоугольники, независимые с $\mathrm{rect}(a, b)$



(b) Вертикальная линия, не пересекающая ни один из прямоугольников набора. Две точки, соответствующие прямоугольнику $\mathrm{rect}(a, b)$

Рис. 8: Доказательство Леммы 18

Иначе образованный ими прямоугольник должен быть arborally satisfiable, и это бы значило, что мы неправильно выбрали p, q (найдётся точка из $E(P)$, принадлежащая прямоугольнику $\mathrm{rect}(p, q)$ и лежащая ближе к ℓ). Сопоставим прямоугольнику $\mathrm{rect}(a, b)$ горизонтальный отрезок pq, удалим этот прямоугольник из набора и продолжим сопоставление.

Claim 20. *Каждый отрезок pq сопоставлен не более чем одному $\mathrm{rect}(a, b)$ из набора независимых прямоугольников.*

Потому что pq лежит внутри $\mathrm{rect}(a, b)$ и пересекает линию, которую не пересекает больше никто из прямоугольников набора, имеющих общие точки с $\mathrm{rect}(a, b)$. Остальные прямоугольники из выбранных нами независимых просто не пересекаются с $\mathrm{rect}(a, b)$.

Рассмотрим точки $p_1 \dots p_t$, $q_1 \dots q_t$, отрезки с концами в которых были сопоставлены некоторым прямоугольникам и которые все оказались на одной горизонтальной прямой.

Claim 21. *Точки p_i, q_i (соответствующие одному прямоугольнику) — соседние из отмеченных точек на этой горизонтальной прямой.*

В противном случае отрезок $p_i q_i$ будет пересекать какой-то другой отрезок $p_j q_j$. И в процессе сопоставления точек соответствующим прямоугольникам мы бы взяли какие-то другие точки.

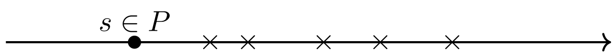


Рис. 9: Точки, добавленные в данную строку

Рассмотрим некоторую строку, в ней находится одна точка из исходного множества запросов P, смотреть Рисунок 9. Пусть мы добавили в эту строку ещё n точек, сопоставленных различным независимым прямоугольникам. Тогда точек стало $n + 1$, и наибольшее число прямоугольников, которое может им соответствовать, — n, потому что Утверждение 21. То есть на одну точку из $E(P)$ добавляется не более одного прямоугольника. Лемма доказана. □

Доказательство теоремы 17.

$$\mathrm{OPT} \ \geq \ \max(\mathrm{OPT}_+, \mathrm{OPT}_-).$$

Теперь воспользуемся тем, что максимум не меньше среднего, а также леммой 18.

$$\max(\mathrm{OPT}_+, \mathrm{OPT}_-) \ \geq \ |P| + \frac{1}{2}(\mathrm{MAX\,IND}_+ + \mathrm{MAX\,IND}_-) \ \geq$$
$$\geq \ |P| + \frac{1}{2} \cdot \mathrm{MAX\,IND}.$$
□

5.3 Более практичная оценка снизу

Рассмотрим пару (s_i, i) из набора поисковых запросов. Упорядочим все остальные точки (s_j, j), $j < i$ по второй координате и соединим их y-монотонной ломаной сверху вниз, смотреть Рисунок 10.

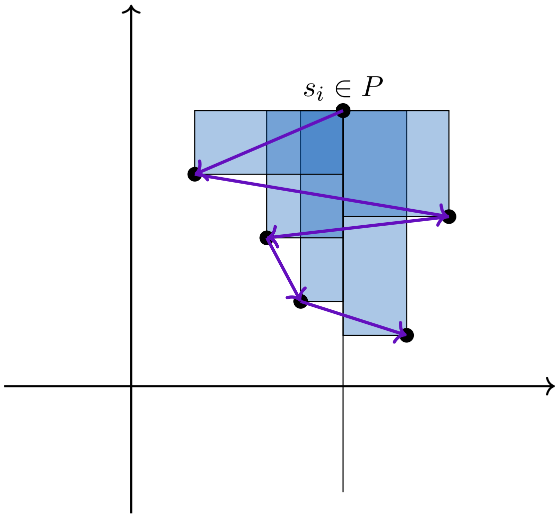


Рис. 10: Подсчёт числа пересечений с вертикальной прямой

Обозначим через $J(s_i)$ количество пересечений этой ломаной с вертикальным лучом, идущим из s_i вниз. Понятно, что такое число можно посчитать для любого элемента последовательности запросов.

Theorem 22.

$$\mathrm{OPT}(P) \ \geq \ |P| + \sum_{s_i} \frac{J(s_i)}{2} \qquad (3)$$

Доказательство. На каждом ребре ломаной, пересекающем вертикальный луч, построим как на диагонали прямоугольник, стороны которого параллельны осям координат. Так у каждого пересечения появится свой прямоугольник. Объединим получившиеся наборы прямоугольников, смотреть Рисунок 11.

Все прямоугольники в объединении, легко видеть, будут попарно независимы. Осталось лишь применить теорему 17. □

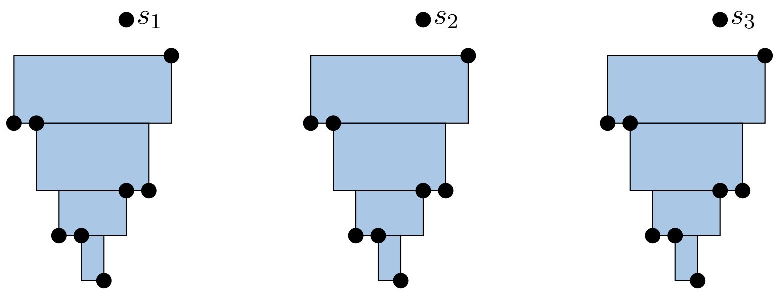


Рис. 11: Набор попарно независимых прямоугольников

5.4 Оценка снизу через число перебежек

Рассмотрим вершину q бинарного дерева поиска T. Обозначим через $R(q)$ количество чередований между спусками в левое поддерево q и правое поддерево q. Спуски в сам узел q и всё, что происходит вне поддерева q, при этом игнорируется.

Theorem 23.

$$\mathrm{OPT}(P) \ \geq \ \sum_{q \in T} R(q). \tag{4}$$

Доказательство. Следует из Теоремы 22. □

0	0 0 0	0 0 0	0
1	0 0 1	1 0 0	4
2	0 1 0	0 1 0	2
3	0 1 1	1 1 0	6
4	1 0 0	0 0 1	1
5	1 0 1	1 0 1	5
6	1 1 0	0 1 1	3
7	1 1 1	1 1 1	7

Рис. 12: Bit-reversal sequence делает нижнюю оценку бессмысленно большой

6 Tango деревья

Дерево, где у каждой вершины есть «любимый потомок» — тот, в которого происходил спуск при предыдущем запросе. Отметим у каждой вершины её любимого потомка — дерево окажется представленным виде объединения путей, смотреть Рисунок 13.

Каждому такому пути сопоставим дерево поиска (чтобы за $\log\log$ отправляться в нужное место пути). При смене любимого потомка у вершины нам придётся перестраивать такие деревья. Это мы умеем.

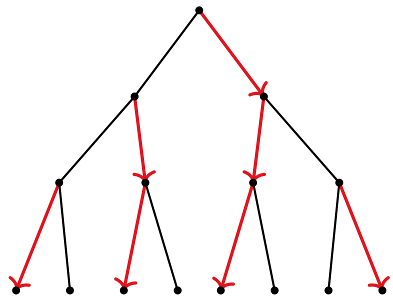


Рис. 13: Tango-дерево представлено в виде объединения путей

7 Link-Cut trees

7.1 Описание структуры, план действий

Наша цель — поддерживать структуру данных, которая умеет хранить лес подвешенных бинарных деревьев и производить с ними следующие операции:

- makeTree(v) — создать дерево из одной вершины v.
- link(v, w) — подвесить u к w (при этом u является корнем одного из деревьев леса, а у w не более одного ребёнка).
- cut(v) — удалить ребро между v и её родителем.
- findRoot(u) — найти корень дерева вершины u.
- findCost(u) — возвращает ближайшее к корню ребро минимального веса на пути от u до корня.
- addCost(u, x) — добавить x к весам всех рёбер на пути от u до корня.

При этом findCost можно адаптировать, чтобы искать не минимум на пути, а, например, сумму и т.д.

В [**sleator1983linkcut**] описано, как добиться асимптотики $\mathcal{O}(\log n)$ на операцию в худшем случае. Мы же изучим link-cut trees, работающие за амортизированное $\mathcal{O}(\log n)$ ([**tarjan1984linkcut**]).

В описании структуры данных и доказательстве времени работы будет две смысловых части.

1) Научиться реализовывать структуру для частного случая дерева — пути. А именно, нам потребуются следующие операции:
   - makePath(v) — создать путь из одной вершины.
   - findPath(v) — вернуть путь, в котором лежит вершина v.
   - findTail(p) — найти верхний конец пути p.
   - join(p, v, q) — объединить пути p и q в один через вершину v, т.е., верхний конец пути p и нижний конец пути q соединить с v.

   - split(v) — отрезать ребро, ведущее из v в предка в пути.
   - findPathCost(u), addPathCost(u, x).

2) Выразить операции на лесе через операции на путях. Т.е., разобьём вершины дерева на пути. После этого некоторые рёбра лежат на путях (сплошные рёбра), а некоторые соединяют разные пути (пунктирные рёбра). Для операций на дереве нам понадобится также дополнительная функция expose(v), которая превращает путь от v до корня дерева в один из путей разбиения (при этом рёбра, идущие из v вниз, не входят в этот путь).

7.2 Выражение операций на дереве через операции на путях

Мы начнём с того, что выразим операции на дереве (разбитом на пути) через операции на путях и expose(v).

```
procedure makeTree(u)
    makePath(u)
procedure findRoot(u)
    findTail(expose(u))
procedure findCost(u)
    expose(u)
    findPathCost(u)
procedure addCost(u, x)
    expose(u)
    addPathCost(u, x)
procedure link(u, w)
    join(ø, expose(u), expose(w))
procedure cut(v)
    expose(v)
    split(v)
```

Таким образом, expose помогает нам свести задачу на дереве к задаче на пути. Мы считаем, что функция expose возвращает указатель на путь, получившийся в результате её исполнения. Некоторых пояснений требует функция link: здесь мы отождествляем вершину и путь, состоящий только из этой вершины.

Итак, теперь нужно научиться делать expose.

```
procedure expose(u)
    p := ø                               ▷ Здесь будем накапливать наш текущий путь
    while u ≠ ø do
        w := successor(findPath(u))      ▷ Запомним следующий сверху путь в дереве
        (q, r) := split(u)               ▷ Отрежем у u сплошное ребро вниз
        if q ≠ ø then                    ▷ q — часть пути, проходящего через u, ниже u
            successor(q) := u            ▷ Теперь ребро из q в u — пунктирное
        p := join(p, u, r)               ▷ А ребро из u в наш текущий путь — сплошное
        u := w                           ▷ Перейдём к вершине следующего пути
```

```
    successor(p) := ∅
```

Операцию, которая происходит в теле while, назовём splice.

Theorem 24. *Пусть выполнено m операций с деревом, из них n операций makeTree (т.е., в дереве не более n вершин). Тогда верно следующее:*

1) *Мы произвели $\mathcal{O}(m)$ операций с путями дерева.*

2) *expose был вызван $\mathcal{O}(m)$ раз.*

3) *За все вызовы expose было выполнено $\mathcal{O}(m \log n)$ операций splice.*

Доказательство. Первые два пункта очевидно следуют из того факта, что во всех операциях на дереве expose вызывается константное количество раз. Докажем оценку на количество splice.

Назовём ребро (v, w) тяжёлым, если $2 \cdot \text{size}(v) > \text{size}(w)$, и лёгким, если это неравенство не выполняется. Таким образом, на пути от любой вершины до корня дерева не более логарифма лёгких рёбер.

Мы будем рассматривать следующие величины:

- HS — количество тяжёлых сплошных рёбер в текущий момент времени;
- HSC — сколько раз мы создавали тяжёлые сплошные рёбра к текущему моменту времени.

Каждый splice превращает некоторое пунктирное ребро в сплошное. Будем рассматривать отдельно лёгкие и тяжёлые рёбра. Так как на пути от u до корня не более логарифма лёгких рёбер, то и превратить лёгкое пунктирное в лёгкое сплошное мы могли не более логарифма раз.

Тогда $\#\text{splice} \leq m(\log n + 1) + \text{HSC}$.

В конце $\text{HS} \leq n - 1$. Значит, почти все создания тяжёлых сплошных рёбер были «отменены», т.е., если мы создавали HSC тяжёлых сплошных рёбер, то по крайней мере $\text{HSC} - n + 1$ раз мы превратили тяжёлое сплошное в тяжёлое пунктирное.

Это могло произойти во время splice, тогда одновременно с этим мы превратили лёгкое пунктирное в лёгкое сплошное. Из этого следует, что $\text{HSC} \leq n - 1 + \frac{m}{2}(\log n + 1)$

Итак, мы получили нужную оценку на количество splice. По модулю одной маленькой детали: операции link и cut тоже влияют на наш потенциал HSC.

Во время этих операций лёгкое сплошное ребро могло превратиться в тяжёлое сплошное — такие тяжёлые рёбра можно просто не учитывать в значении HSC.

Также тяжёлое сплошное ребро могло превратиться в лёгкое сплошное. Это соответствует уменьшению потенциала, которое при этом не «уравновешивает» создание этого тяжёлого ребра в какой-то предыдущий момент времени. Однако, так как на любом пути лёгких рёбер не больше логарифма, то на каждую из m операций может произойти не более $\log n$ «незарегистрированных» изменений потенциала.

Суммарно это внесёт в HSC (и нашу итоговую оценку) ещё $\mathcal{O}(m \log n)$ операций.

□

7.3 Операции на путях

Для реализации операций на путях мы будем использовать Splay-дерево. Будем хранить путь в дереве таким образом, чтобы при обходе дерева dfs-ом мы выписывали путь слева направо, заканчивая вершиной tail (таким образом, findTail будет просто возвращать самую правую вершину дерева). В узле дерева будем хранить также следующие величины:

- $\Delta\text{cost}(x) = \text{cost}(x) - \text{mincost}(x)$, где $\text{mincost}(x)$ — это минимальная стоимость вершины в поддереве x.
- $\Delta\text{min}(x) = \text{mincost}(x) - \text{mincost}(p(x))$, а если x — корень дерева, то $\Delta\text{min}(x) = \text{mincost}(x)$

```
procedure makePath(u)
    makeSplayTree(u)
procedure findPath(v)
    splay(v)
    return(v)
procedure findPathCost(v)
    while right(v) ≠ 0 and min(right(v)) = 0 or left(v) ≠ 0 and min(left(v)) = 0 do
        if right(v) ≠ 0 and min(right(v)) = 0 then
            v := right(v)
        else
            v := left(v)
    splay(v)
    return(v, Δmin(v))
procedure addPathCost(v, x)
    Δ(min)(v) = Δ(min)(v) + x
procedure join(p, v, q)
    v.left = p
    v.right = q
procedure split(v)
    splay(v)
    cut(v, v.left)
    cut(v, v.right)
```

Для анализа мы воспользуемся уже доказанной асимптотикой splay-дерева. Мы рассмотрим «виртуальное» splay-дерево, которое будет состоять из всех splay-деревьев путей, а также проведённых между путями пунктирными рёбрами.

Потенциалы будут такими же:

$$iw(v) = \begin{cases} \text{size}(v), & \text{если у v два пунктирных ребра} \\ \text{size}(v) - \text{size}(u), & \text{если (u, v) — сплошное ребро} \end{cases}$$

$$tw(v) = \sum_{\text{u — из поддерева v в виртуальном дереве}} iw(u)$$

$$r(v) = \log tw(v)$$

$$\Phi = \sum_v r(v)$$

Тогда за одну операцию splay на одном splay-дереве мы платим $3(r(u) - r(v)) + 1$, что даёт амортизированный логарифм, как в анализе асимптотики splay-дерева. Но нам нужно сказать, что на все операции splay во время выполнения одного expose мы суммарно заплатим не более логарифма. Легко видеть, что операция splay не меняет структуры виртуального дерева, а значит, не меняет потенциалы. Таким образом, во время переходов от одного пути к другому во время операции expose все слагаемые $r(v)$, кроме двух, взаимно уничтожатся. Тогда:

$$\text{expose}(v) = 3(r(root) - r(v)) + 2\#\text{splice},$$

что есть $\mathcal{O}(m \log n)$.

8 Mergeable trees

Полное описание всех структур и доказательств можно найти в [**georgiadis2011data**].

8.1 Описание структуры и чего мы хотим от этой структуры

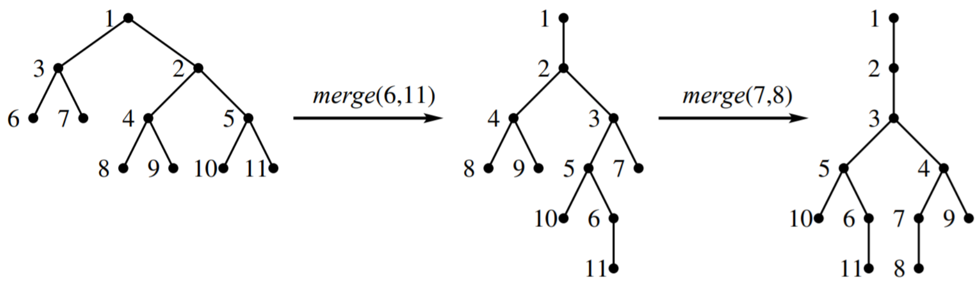
- Храним:
  - храним лес бинарных деревьев с корнем;
  - на узлах — рациональные ключи;
  - выпонено свойство кучи: $l(v) \geqslant l(p(v))$.
- Поддерживаем операции:
  - $parent(v)$ — узнать, кто является предком v;
  - $root(v)$ — узнать, кто является корнем дерева, в котором находится v;
  - $nca(v, w)$ — nearest common ancestor: если в одном дереве, то возвращает первого общего предка, если не в одном дереве, то возвращает пустое множестве;
  - $insert(v, x)$ — создание нового дерева с вершиной v и ключом x;
  - $link(v, w)$ — подвесить v к w, причём v — корень какого-то дерева из леса, а у w не более одного ребёнка. Также $l(v) \geqslant l(w)$;
  - $cut(v)$ — удалить ребро между v и его родителем;

– *delete*(v) — удалить v, если v — лист;

– *merge*(v, w) — для начала надо сделать *link* корней вершин, если вершины в разных деревьях. Затем мы рассматриваем два пути:

  * P — путь от v к корню дерева;
  * Q — путь от w к корню дерева.

  Далее мы совмещаем эти два пути с сохранением свойства кучи. Примеры двух **merge** показаны ниже:



- Реализация. Реализация будет устроена так же, как и в **link-cut trees** за исключением того, что мы добавим еще одну команду **topmost**, про которую поговорим чуть позже.

  Заметим, что **merge** есть обобщение **link-cut**, поэтому считаем, что **link-cut** это **merge**.

  Введём два обозначения:

  – m — количество **merge** операций к данному моменту, включая **link-cut**;

  – n — количество **insert** операций к данному моменту (начинаем с пустого дерева).

  Таким образом, всё будет занимать $\mathcal{O}(n)$ памяти.

- План.

  Сначала заметим, что в идеале хотим всё уметь делать за амортизированное $\mathcal{O}(\log n)$. Но научимся только вот так:

  – если в последовательности операций над деревом есть **cut**, то умеем делать **merge** за амортизированное $\mathcal{O}(\log^2 n)$;

  – если **cut** в последовательности нет, то научимся делать за амортизированное $\mathcal{O}(\log n)$;

  – ну а еще попутно докажем некоторые нижние оценки.

8.2 Реализация merge

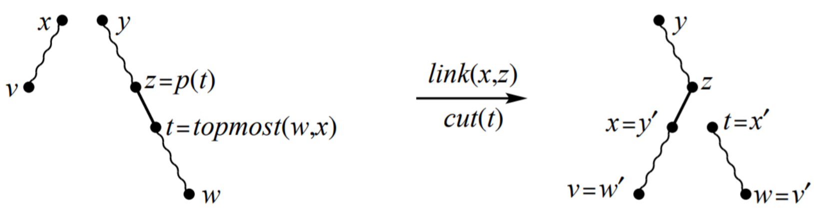
Как и оговаривалось ранее, введём операцию $\mathbf{topmost}(\mathbf{v}, \mathbf{w})$.

$topmost(v, w)$ — мы идём по пути от v к $root(v)$, начиная с v, и берём первую вершину u такую, что $l(u) > l(w)$. Это можно сделать за $\mathcal{O}(\log n)$ стандартным бинарным поиском.

```
procedure MERGE(v, u)
    u := nca(v, w)                          ▷ ищем общего предка v и w
    if u = v or u = w then
        return   ▷ если общий предок совпадает с v или w, то пути уже совмещены
    else
        x = topmost(v, u)
        y = topmost(w, u)
        if x < y then
            swap(x, y)
            swap(v, w)
        if u ≠ ø then
            cut(x)
        while x < u do
            t := topmost(w, x)
            link(x, parent(t))
            cut(t)
            y := x
            x := t
            swap(v, w)
        link(x, w)
```

Если вершины находятся в разных деревьях, то нужно сделать еще один **link** в самом начале.

Операцию, которая находится в теле while назовём **Merge step**. Картинка к ней показана ниже:



Чтобы оценить количество шагов, достаточно оценить количество изменений родителя. Поймём, что количество изменений родителя это количество merge steps плюс не более чем 2: в каждом merge step ровно одно изменение родителя, еще одно берётся в самом конце (**link(x, w)**) и еще может быть одно, если они из разных деревьев.

Теперь докажем следующую лемму.

Lemma 25. *Количество изменений родителя это* $\mathcal{O}(m \log n)$.

Доказательство. Будем считать, что все наши ключи это не просто рациональные числа, а числа $\{1, \ldots n\}$. Для этого упорядочим все наши ключи в порядке возрастания и заменим каждый ключ на соответствующий номер.

Определим **cost** операции как количество изменений родителя.

$$\mathbf{am.\,cost} = cost + \Delta\Phi$$

Потенциал следующий:

- каждому ребру e даём 1 потенциала(Φ_e);
- от каждого ребра $(parent(v), v)$ дадим $\log(v - w)$ родителю и $\log(v - w)$ ребёнку;
- Φ_v от вершины определяется как сумма потенциалов, которая приходит ей от рёбер (из предыдущего пункта);
- $\Phi = \sum\limits_{v \in V} \Phi_v + \sum\limits_{e \in E} \Phi_e$.

Заметим, что **cut** и **delete** уменьшают Φ хотя бы на 1 и дают не более одного изменения родителя. Значит их $\mathbf{am.\,cost} \leqslant 0$. Остаётся только **merge**.

В начале **merge** возможен **initial link** (так как вершины могут быть в разных деревьях). Заметим, что он увеличивает потенциал не более, чем на $2\log n + 1$: по логарифму в каждой вершине и 1 от ребра.

Как влияет **merge step**: Посмотрим на родителей t за два **merge step**: пусть $parent'(t)$ это родитель, который будет у t на следующем **merge step** после того, который показан на картинке. Тогда:

$$t > parent'(t) \geqslant x > parent(t)$$

Первое неравенство понятно — это просто свойство кучи, второе чуть хитрее: новый родитель t будет в ветке y', то есть в ветке x, но вполне может так оказаться, что это будет сам x, поэтому больше либо равно. Последнее неравенство снова видно из картинки, так как $z = parent(x)$.

Математической магией этих трёх неравенств получаем, что:

- либо $t - parent'(t) \leqslant \frac{t - parent(t)}{2}$;
- либо $x - parent(t) \leqslant \frac{t - parent(t)}{2}$.

В первом случае родительский потенциал t уменьшается не менее чем на 1. Во втором случае детский потенциал x уменьшается не менее чем на 1.

Одно из этих условий точно выполнится, а значит потенциал уменьшится хотя бы на один, таким образом (так как у нас ровно одно изменение родителя)

$$am.\,cost(merge\ step) \leqslant 0.$$

Стоит подметить (иначе неверна строка выше), что после **initial link** все остальные изменения потенциала неположительны. Это **(!)** вроде как следует из того факта, что для t его и родительский и десткий потенциал только уменьшаются (родительский - видно из неравенства, а детский, потому что на самом деле t переходит в x, а его детский потенциал уменьшился опять же из-за этого неравенства). В итоге t пробегает по всем вершинам пути (так как переходит в x, а x в свою очередь переходит в y).

Таким образом амортизационная стоимость **merge** это $\mathcal{O}(\log n)$. Если вспомнить, что для всех остальных операций требуется $\mathcal{O}(\log n)$ времени, то получается, что амортизационное время работы **merge** это $\mathcal{O}(\log^2 n)$. □

8.3 Merge без операций cut

8.3.1 Новая структура дерева

Разбиваем деревья на сплошные пути: некоторые рёбра считаем сплошными, некоторые рёбра считаем пунктирными; компонента связности по сплошным рёбрам и есть сплошной путь.

Definition 20. Определим ранг вершины, как $r(v) := \lfloor \log size(v) \rfloor$, где $size(v)$ — это сумма по ключам в поддереве v, включая саму v.

Разбивать будем следующим образом: говорим, что ребро (v, w) сплошное, если $r(v) = r(w)$. **Note!** Заметим, что все таким образом определённые сплошные рёбра были бы сплошными в **link-cut** trees.

Для каждого узла будем хранить следующее:

- указатель на **parent(x)**;
- указатель на **solid child(x)** (то есть указатель на ребёнка, соединённого сплошным ребром, если такой есть);
- указатель на **head(x)** — отдельная вершина, в которой хранится указатель начало сплошного пути, в котором вершина находится;
- **dashed size(x)** $:= 1 + \sum\limits_{u \text{ — пунктирный ребёнок}} size(u)$

Такая структура двухсвязного списка позволяет ускорить операции вставки и удаления.

Также, для каждого сплошного пути мы будем хранить следующее:

- **head(x)** — отдельная вершина, в которой хранится указатель на **top** пути, а также размер **top** и ранг всего пути;
- поисковая структура — пока структура, которая будет уметь выполнять три функции:
  - удалять узлы в начале пути за $\mathcal{O}(1)$;
  - добавлять одиночные узлы перед заданным за $\mathcal{O}(1)$;
  - делать **topnode(x, s)** — наименьший узел на сплошном пути, содержащим s, ключ которого больше чем x.

- более того, каждый **head** хранит указатель на поисковую структуру соответствующего сплошного пути.

Тот факт, что мы храним только размер top пути, позволяет нам пересчитывать все остальные размеры, пользуясь формулой:

$$size(x) = size(p(x)) - d(p(x)).$$

8.3.2 Стэк S_v

Теперь root(v) возвращает стек начал сплошных путей, которые находятся на пути $[v, root(v)]$. Эти стеки полезны в нахождении **nca(v, w)**. Для начала, найдём два стека для v, w соответственно (обозначим их за S_v и за S_w). А далее:

procedure NCA(v, w)
 if $top(S_w) \neq top(S_v)$ **then**
  return 0
 else
  while $top(S_v) = top(S_w)$ **do**
   $pop(S_v)$
   $pop(S_w)$
  if $S_v = S_w = \varnothing$ **then**
   return $\min\{v, w\}$
  if $S_v = \varnothing$ and $S_w \neq \varnothing$ **then**
   return $\min\{v, p(top(S_w))\}$
  if $S_v \neq \varnothing$ and $S_w = \varnothing$ **then**
   return $\min\{w, p(top(S_v))\}$
  if $S_v \neq \varnothing$ and $S_w \neq \varnothing$ **then**
   return $\min\{p(top(S_v)), p(top(S_w))\}$

Так как рангов всего не более $\log n$, то видно, что **nca** работает за $\mathcal{O}(\log n)$.

8.3.3 Link

Теперь поговорим про **link(v, w)**. Проделывать мы будем его в несколько этапов:

- добавляем ребро (v, w), сначала как пунктирное;
- ищем множество узлов, где ранг меняется. Обозначим его за Q. Утверждается, что Q представляет собой объединение начал сплошных путей. Поэтому искать мы Q будем поднимаясь в **head** пути и спускаясь, пока ранг отличается от старого.
- все сплошные выходящие рёбра из вершин из Q заменить на пунктирные, пересчитать ранг;
- меняем обратно те пунктирные, которые должны быть сплошными; (Здесь возникает проблема, так как нам надо следить за добавлением вершин в поисковую структуру пути)

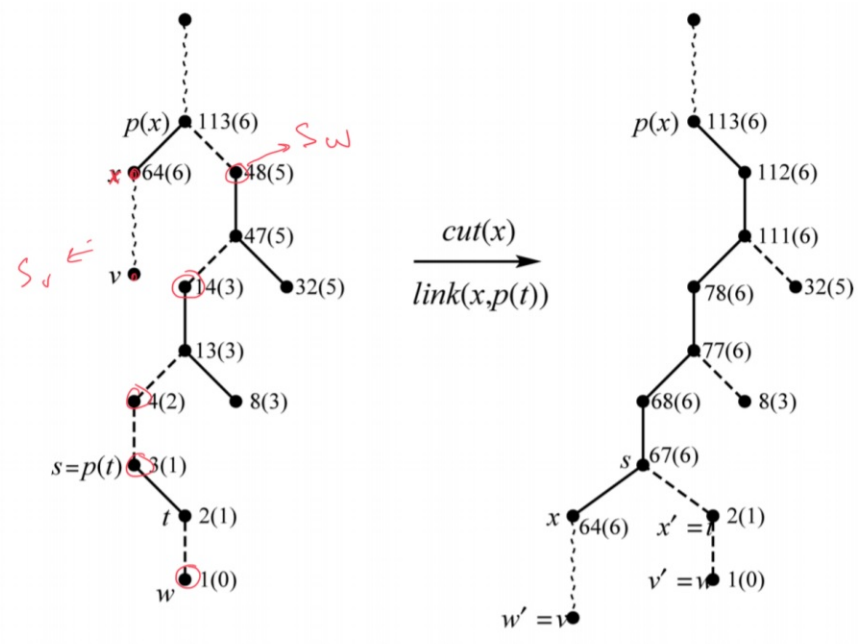
Lemma 26. *Суммарное время на все операции **link** это $\mathcal{O}(m \log n)$.*

Доказательство. Можем оценить время работы как константа умножить на количество вставок в Q или в S_w. Заметим, что если мы вставляем вершину в Q, то в итоге вставляем и в S_w (когда мы меняем сплошные рёбра на пунктирные; здесь мы можем считать, что мы просто добавляем их в S_w).

Также в самом начале мы вызывали **root(w)** и поэтому мы добыли S_w размера $\mathcal{O}(\log n)$. Заметим, что при добавлении вершины в S_w её ранг увелиличился. Так как ранг для каждой вершины увеличился не более чем на $\mathcal{O}(\log n)$ раз и $m > n$, то общее время работы не более $\mathcal{O}(m \cdot \log n)$. **Важно!** Так как у нас нет cut, то ранги только растут. ☐

8.3.4 Merge

- сначала мы снова заботимся о том, чтобы обе вершины оказались в одном дереве, делая **link**;
- в остальном делаем merge также как и в прошлой подсекции (что мы делаем с обновлением структур поймём позже)



- **(?)** утверждается, что понимать, куда именно вставлять вершины мы будем уметь, используя структуру на путях (которую мы еще не ввели) и **topnode**.

Lemma 27. *Не считая вызовов **topnode**, затраченное время на все **merge** не превосходит $\mathcal{O}(m \cdot \log n)$.*

Доказательство. Во время одного merge мы возможно соединяем два дерева с помощью **link**. Это мы делаем за амортизированный $\mathcal{O}(\log n)$.

Из первой подсекции мы знаем, что количество **merge steps** это $\mathcal{O}(m \cdot \log n.)$ Во время **merge step** мы достаем из S_v и S_w и добавляем в Q. Как и в прошлой лемме, в начале мы добавили в S_v и S_w по $\mathcal{O}(\log n)$ вершин, а последующие добавления снова лишь увеличивают ранг, а так как у каждой вершины не более $\mathcal{O}(\log n)$ увелечений ранга, то мы снова получаем $\mathcal{O}(m \cdot \log n)$. **Важно!** Так как у нас нет **cut**, то ранги только растут.

□

8.3.5 Поисковая структура на путях

Также как и в **link-cut trees** мы будем представлять сплошной путь в виде **splay-дерева**.

Будем доказывать, что в **splay-дереве topnode** запрос можно сделать за $\mathcal{O}$(**parent change**), где **parent change** следует за запросом **topnode**.

Definition 21. Finger — узел, посещённый в splay-дереве последний раз.

Результат без доказательства: **am. cost** (операций со **splay-деревом**) это $\mathcal{O}(\log d + 1)$, где $d = |f' - f|$ **(?)** Здесь, f' и f — новый и старый **finger** соответственно.

Definition 22. $\log d + 1$ — время Коула для данной операции.

Lemma 28. *Добавление посещения узла между операциями не уменьшает время Коула.*

Доказательство. Самостоятельно (утверждается, что несложно). □

Предпологается, что можно доказать, что **topnode** работает за $\mathcal{O}$(**parent change**) без модификаций, но это открытый вопрос, поэтому мы модифицируем: когда хотим что-то добавить в сплошные пути во время работы **merge**, мы добавляем не сразу а при случае (позволяется иметь узы в сплошном пути не в **splay-дереве**).

Тогда реализация **topnode(x,e)** выглядит следующим образом: мы делаем бинарный поиск e в **splay-дереве**. Тогда ответом будет являться:

- либо последний посещённый узел в бинарном поиске;
- либо сплошной ребёнок последнего посещённого узла;
- либо вершина, которая находится не в дереве.

Если это либо первая, либо вторая ситуация, то **(?)** всё хорошо, мы нашли и делаем **splay**. Если же это третья ситуация, то мы остановились в самой правой нижней вершине, обозначим её за y, делаем **splay(y)**, а далее просто идём по указателям по вершинам и последовательно находим **topnode**. Пока мы идём до нашего результата и находим что-то новое —добавляем это в **splay-дерево**.

Далее будет описана идея доказательства, что **topnode** работает за $\mathcal{O}($**parent change**$)$ (автор конспекта не очень понял на момент написания, для полного понимания нужно прочитать почти всю статью):

На **topnode** требуется $\mathcal{O}(1)$ + количество операций со **splay-деревом** (ну и время на эти операции).

Определим потенциал и **am. cost**:

Пусть d_f — количество предков f на сплошном пути. Тогда потенциал дерева $\Phi_T = \log d_f$, где f — это **finger** дерева. Потенциал всей системы это $\sum\limits_{T \in M} \Phi_T$.

am.cost операций со **splay-деревом** = время Коула + увеличение потенциала.

Также перед любым **merge** мы "просто потрогаем **top** пути."**Важное замечание.** Если мы хотим что-то делать с **top** пути, то это будет за $\mathcal{O}(1)$, так как время Коула уравновешивается изменением потенциала. Если мы хотим что-то делать на расстоянии $\mathcal{O}(1)$ от предыдущего узла, то тоже делается за $\mathcal{O}(1)$. Из чего следует, что **delete** и **insert** из **top**, в **top**, либо рядом с предыдущим **finger** тоже делается за $\mathcal{O}(1)$.

Из всего этого мы можем заключить, что суммарная стоимость на **merge** $\mathcal{O}(m \cdot \log n)$ за исключением **access**.

Access: разбор случаев, но в каждом случае **am.cost** $\leqslant 2 \cdot$ **cost(query)** $+ \mathcal{O}(1)$, где **cost(query)** — изменение родителей.

9 Динамизация структур данных

Пусть у нас есть задача поиска:

$$Q\colon X \times 2^D \to A,$$

где A — ответы, D — объекты, X — запросы.

9.1

Definition 23. Говорим, что задача поиска является decomposable, если функция Q обладает следующим свойством:

$$Q(x, D \cup D') = Q(x, D) \blacklozenge Q(x, D'),$$

где $\blacklozenge$ означает, операцию, которая быстро считается.

Example 1. *Пусть мы хотим в каком-то множестве точек узнавать ближайшего соседа от точки запроса. Ясно, что мы можем узнать ближайшего соседа в множестве D, ближайшего соседа в множестве D' и взять из них того, что ближе.*

Предположим, что у нас есть такая функция Q. Давайте сформулируем задачу:

Итак, пусть существует структура данных, которая умеет только хранить и выдавать ответ на запрос, со следующими свойствами:

- В ней n объектов;
- Она занимает $S(n)$ памяти;
- Её можно построить за $P(n)$;
- Она отвечает на вопрос за $Q(n)$.

Problem 1. Мы хотим построить структуру данных, которая обладает следующими свойствами:

- Она занимает $S'(n) = \mathcal{O}(S(n))$ памяти;
- Она строится за $P'(n) = \mathcal{O}(P(n))$;
- Она отвечает на вопрос за $Q'(n) = \mathcal{O}(\log n \cdot Q(n))$;
- Вставка занимает $I'(n) = \mathcal{O}\left(\log n \cdot \frac{P(n)}{n}\right)$.

В данной задаче имеется ввиду амортизированное время для запроса и вставки.

Итак, давайте строить. Разбиваем наши элементы на на логарифмическое количество уровней. Теперь у нас имеется $\log n$ уровней, которые называются $L_0, \ldots, L_l$:

- L_0: $\varnothing$ или структура с 1 элементом
- L_1: $\varnothing$ или структура с 2 элементами

  $\vdots$
- L_i: $\varnothing$ или структура с 2^{i-1} элементами

  $\vdots$
- L_l: $\varnothing$ или структура с 2^{l-1} элементами

Запрос делаем следующим образом:

```
Query(x):
  a := E (ответ на ∅)
  for i = 0 to l
    if L_i ≠ ∅
      a := a♦Q(x, L_i)
  return a
```

Ясно, что на запрос мы потратим времени

$$\sum_{i=0}^{l-1} Q(2^i) < l \cdot Q(n) = \mathcal{O}(\log n) Q(n).$$

Remark. Если $Q(n)$ — большое, то есть $Q(n) > n^{\varepsilon}$ при каком-то $\varepsilon > 0$, то время на запросе — это $\mathcal{O}(Q(n))$.

Вставку делаем следующим образом:

```
Insert(x):
  Find min k : L_k = ∅
  build L_k := {x} ∪ ⋃_{i<k} L_i
  for i = 0 to k − 1
    destroy L_i
```

Ясно, что build происходит за $P\left(2^k\right)$.

Мы хотим, чтоб цена за одну вставку была

$$I'(n) = \mathcal{O}(\log n)\frac{P(n)}{n}.$$

$$I'(n) = \mathcal{O}(\log n)\frac{P(n)}{n} = \sum_{i=0}^{\log n} \frac{P(2^i)}{2^i}.$$

Ясно, что за n вставок у нас получится

$$\sum_{i=0}^{\log n} P\left(2^i\right)\frac{n}{2^i} = n\sum_{i=0}^{\log n} \frac{P(2^i)}{2^i} = \mathcal{O}(\log n)P(n).$$

Что и требовалось.

Remark. Если $P(n) > n^{1+\varepsilon}$, где $\varepsilon > 0$, то $I'(n) = \mathcal{O}\left(\frac{P(n)}{n}\right)$.

Problem 2. Мы хотим получить тот же результат, что и в Задаче 1, только время для запроса и вставку теперь не амортизированное, а в худшем случае.

Теперь на каждом уровне у нас будут старые структуры и новые. То есть, на уровне i будут находиться структуры O_i^1, O_i^2, O_i^3, N_i, снова в каждой из них $\varnothing$ или 2^i объектов. Также ещё мы хотим, чтоб выполнялось:

- если $O_i^1 = \varnothing$, то $O_i^2 = \varnothing$
- если $O_i^2 = \varnothing$, то $O_i^3 = \varnothing$

От N_i хотим, чтоб $N_i = \varnothing$ или N_i было частичной (то есть той, которая находится в процессе построения) структура для 2^i объектов. Также нам надо, чтоб каждый объект был ровно в одной структуре O и, может быть, в структуре N.

Запрос делаем следующим образом:

```
Query(x):
   a_i = E
   for i = 0 to l
      if Q_i^1 ≠ ∅
         a_i = a♦Query(x, Q_i^1)
      if Q_i^2 ≠ ∅
         a_i = a♦Query(x, Q_i^2)
      if Q_i^3 ≠ ∅
         a_i = a♦Query(x, Q_i^3)
   return a
```

Очевидно, что время запроса у нас такое же.

Вставку делаем следующим образом:

```
Insert(x):
   for i = l...1
      if O_{i-1}^1 ≠ ∅ and O_{i-1}^2 ≠ ∅
         do P(2^i)/2^i шагов построения N_i(:= O_{i-1}^1 ∪ O_{i-1}^2)
         if N_i is complete
            destroy O_{i-1}^1, O_{i-1}^2
            O_{i-1}^1 := O_{i-1}^3
            destroy O_{i-1}^3
            Update(i)
         N_0 := x
         Update(0)
```

Что такое Update(i)?

```
Update(i):
   if O_i^1 = ∅
      O_i^1 := N_i
   else if O_i^2 = ∅
      O_i^2 := N_i
   else O_i^3 := N_i
   destroy N_i
```

Все ошки не могут быть заняты, это доказывается по индукции, но это было оставлено в качестве упражнения.

Идея в том, что мы не делаем построение сразу, а делаем столько его шагов, сколько можем себе позволить. А значит, время вставки даже в худшем случае будет $\mathcal{O}\left(\frac{P(n)}{n}\log n\right)$.

9.2

Definition 24. Говорим, что задача поиска является invertible, если функция Q обладает следующим свойством: Если $D = D_1 \cup D_2$, то

$$Q(x, D_1) = Q(x, D) - \blacklozenge Q(x, D_2),$$

где $-\blacklozenge$ быстро считается.

Предположим, что у нас есть такая функция Q. Давайте сформулируем задачу:

Problem 3. Пусть у нас есть структура данных M, которая умеет вставлять и ещё другая структура данных G, в которую мы будем вставлять элемент, который хотим удалить из M. Хотим, чтоб амортизированное время удаления из M равнялось $\mathcal{O}\left(P(n)\frac{\log n}{n}\right)$.

```
Query(x):
  Return Q(x, M) −♦Q(x, G)
```

Подвох заключается в том, что у нас может быть много удалённых элементов, поэтому весь наш анализ будет от количества элементов, которые когда-либо вставлялись. А мы хотим не этого.

Мы будем ждать момента, когда $|G| > \frac{1}{2}|M|$, и в это время перестраивать структуру полностью. Идея в том, что между двумя такими моментами пройдёт минимум $\frac{n}{2}$ удалений, а значит, когда мы считаем амортизированную стоимость удаления, мы можем заключить, что цена каждого удаления — это $\mathcal{O}\left(\frac{P(n)}{n}\right) + \mathcal{O}\left(P(n)\frac{\log n}{n}\right)$ (второе слагаемое — это вставки в структуру G).

```
When |G| > 1/2|M|
  Build M := M \ G
    G := ∅
```

Также есть проблема, что глобальные перестроения могут помешать локальным, то есть для работы вставок мы тоже перестраиваем систему, и надо, чтоб нам хватило ресурсов для вставки, несмотря на то, что мы потратили что-то на удаление. Для борьбы с этим достаточно просто амортизированную стоимость удаления умножить на достаточно большую константу.

Problem 4. Мы хотим получить тот же результат, что и в задаче 3, только время теперь не амортизированное, а в худшем случае.

Для решения Задачи 4 мы поддерживаем три структуры: M, I, G:

```
Query(x):
  Return Q(x) = Q(x, M)♦Q(x, I) − ♦Q(x, G)
```

В какой-то момент снова перестраиваем структуру, а именно в случае, если $|G| > \frac{1}{2}(|M| + |I|)$. Чтоб удовлетворять запросам, придётся заморозить наши структуры. Поэтому также поддерживаем ещё три структуры: M', I', G'.

Во время работы по перестроению структуры, мы будем временно объекты, которые мы хотим удалить, вставлять в G', которые хотим вставить, вставляем в I'. M' — это, собственно, структура, которую мы строим. Пока мы строим:

```
Query(x):
  Return Q(x) = Q(x, M)♦Q(x, I)♦Q(x, I') − ♦Q(x, G) −
♦Q(x, G')
```

Когда мы делаем каждое удаления, мы будем делать $c\frac{P(n)}{n}$ (для какой-то константы c) шагов построения структуры M'.

Через $\frac{n}{c}$ шагов $M := M', I := I', G :=$ G'

Ясно, что время в худшем случае будет каким надо, а именно,

$$\mathcal{O}\left(P(n)\frac{\log n}{n}\right).$$

9.3

Наши запросы не всегда бывают invertible. Далее будет идея того, что нужно делать с теми запросами, которые не invertible.

Нам всё равно потребуется какое-нибудь условие, а именно, weak deletion in time $D(n)$. Это какая-то операция, которая даст структуре данных понять, что этого элемента не должно в ней быть, и при этом не увеличит время на запрос.

Делаем то же самое для вставок — поддерживаем уровни.

Когда нам нужно удалить x: сделаем weak deletion x в каждом уровне, содержащем x (+ чтоб найти эти уровни, нам нужен какой-то словарик, который по элементу называет уровни, где он содержится).

Делаем те же глобальные перестройки, то есть когда не удаленных объектов $> \frac{1}{2}$ удалённых — делаем global rebuild.

Из прошлого рассуждения знаем, что

- амортизированное время вставки — это $\mathcal{O}\left(P(n)\frac{\log n}{n} + D(n)\right)$;
- амортизированное время удаления — это $\mathcal{O}\left(\frac{P(n)}{n}\right)$.

Это также можно несколькими структурами сделать в худшем случае. А именно, структурами структурами: M, S, M', S', u. M — главная, S её дублирует. Когда у нас удалений становится больше, чем половина тех элементов, которые лежат в M, снова начинаем глобальное перестроение. Для этого мы заморозим структуру S, для новых запросов заведём u — очередь запросов, которую будем применять к M'. M' — это рабочая копия M на время перестроения. Когда мы закончили, у нас M' будет на правах M и S' на правах S. После этого надо сделать все обновления в очереди u. Нужно просто подобрать константы, сколько шагов построения S' и M' мы будем выполнять каждый раз, когда делаем удаление.

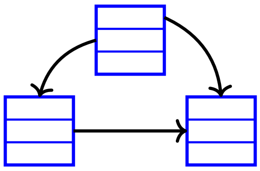
10 Структуры данных и путешествия во времени

Оригинальная статья: [**demaine2007retroactive**].

В этом разделе нам будут интересны структуры данных, позволяющие осуществлять эти самые "путешествия во времени". Определить это понятие можно по-разному, мы разберем два варианта: персистентность и ретрактивность.

10.1 Персистентность

Мы будем рассматривать структуры данных в модели pointer machine. Структура данных рассматривается как некоторое множество узлов, в каждом узле хранится $\mathcal{O}(1)$ полей, в поле может быть записано число или указатель на другой узел. Иногда вводят дополнительное константное ограничение на входящую степень узла, то есть, для каждого узла, количество указателей, указывающих на него, должно быть не больше какой-то константы.



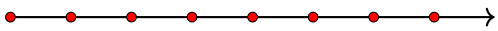
Операции, которые поддерживает структура:

- $x =$ new node — создать новый узел
- $x = y$.field — взять значение поля
- $x = y + z$ — объединить два узла
- destroy(x) — удалить узел, если на него нет указателя

Персистентность бывает разных видов:

1) **Частичная персистентность**

   - изменяем только последнюю версию
   - спрашиваем о любых версиях

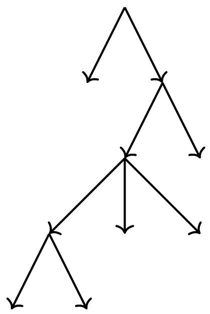
   Известные результаты. Если у структуры данных константная входящая степень для всех узлов, то можно сделать ее частично персистентной с константным мультипликативным overhead-ом.

   [**driscoll1986making**] сделали алгоритм с амортизированным $\mathcal{O}(1)$ overhead-ом.

   [**brodal1996partially**] сделали алгоритм с $\mathcal{O}(1)$ overhead-ом в худшем случае.

2) **Полная персистентность**: частичная + можем изменять прошлое.

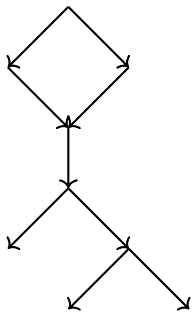
   Каждый раз, когда мы изменяем прошлое, мы создаем новую версию структуры. Получается дерево версий.



Известные результаты. Если у структуры данных константная входящая степень для всех узлов, то ее можно наделить полной персистентностью также с константным мультипликативным overhead-ом. Правда в случае полной персистентности такого overhead-а умеют добиваться только амортизированно ([**driscoll1986making**]), и можно ли добиться его в худшем случае — открытый вопрос.

3) **Конфлюентная персистентность**: полная + можем комбинировать версии.

   В этом случае, помимо обычных изменений структуры, можно также сливать две версии в одну. Получается ациклический граф версий.



   Известные результаты. С конфлюентной персистентностью все сложнее, [**fiat2003making**] умеют с overhead-ом $\log(\#upd) + \max\limits_v e(v)$, где $\#upd$ – это количество всех сделанных к этому моменту изменений, v – вершина в графе версий, а $e(v) = 1 + log\#(\text{путей из корня в } v)$.

   Открытый вопрос: можно ли сделать overhead $\mathcal{O}(\log n)$? При этом $\mathcal{O}(\log n)$ умеют получать для частного случая задачи, когда мы хотим сливать только версии, в которых нет общих полей ([**collette2012confluent**]).

4) **Функциональная персистентность**: комбинированная + нельзя модифицировать узлы

   Известные результаты. В общем виде задачу решать не умеют. Умеют только для частных случаев. Например, Balanced BST и Link-cut-tree умеют делать функционально персистентными с overhead-ом $\mathcal{O}(\log n)$ ([**demaine2008confluently**]).

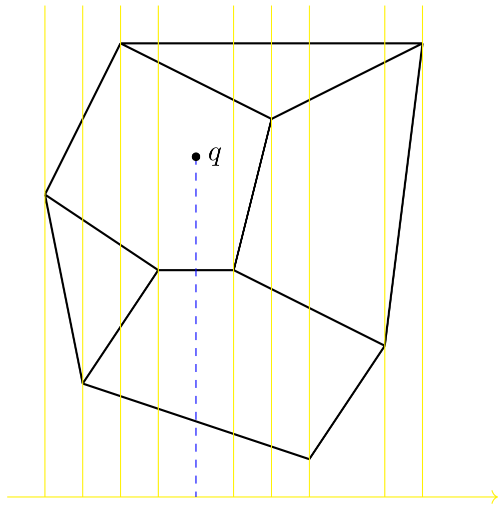
Приложения частичной персистентности

Задача Point location

Дан плоский граф G с ребрами-отрезками. Запросы: по данной точке определить, в какой грани она находится. Можно сделать предобработку.

Решение с помощью частичной персистентности.

Отсортируем вершины графа по координате x. Будем в этом порядке добавлять вершины и поддерживать множество ребер, которые пересекают вертикальную полоску от этой вершины до следующей. Ребра будем хранить в каком-нибудь BST. Таким образом, у нас появилось n версий этого BST, по версии для каждой точки.



Когда приходит запрос, мы сначала должны понять, между какими вершинами v и u по оси x лежит наша точка. Это делается за $\mathcal{O}(\log n)$. Затем обращаемся к BST для v и смотрим, между какими ребрами лежит точка.

Изменяем мы только последнюю версию, запросы делаем к любой, получается, нам достаточно частичной персистентности.

10.2 Ретрактивность

Есть структура данных, поддерживающая какие-то updates и queries.

В случае ретрактивности мы хотим уметь возвращаться в прошлое и исправлять какие-то действия, а именно отменить или добавить какой-то update. При этом мы не будем создавать новую ветку версий, ветка будет всего одна, и все действия, произошедшие после нашего изменения, по-прежнему будут учитываться.

Операции:

- Insert(t, $update$) – добавить изменение $update$ во время t
- Delete(t) – удалить изменение, произошедшее во время t
- Query(t, $query$) – сделать запрос query во время t

Операции мы будем писать с большой буквы, чтобы не путать их с операциями insert, delete и query для структуры.

Также небольшая деталь: мы считаем, что между любыми моментами времени мы всегда можем вставить операцию, причем без дополнительных затрат на кодирование этих моментов времени. Для этого поддерживаем order-maintenance structure, которая будет об этом заботиться, и которую мы не будем обсуждать.

Ретрактивность бывает разных видов:

1) **Частичная ретрактивность**

   - Insert, Delete в любой момент времени t
   - Query только в последний момент времени t_{now}

2) **Полная ретрактивность**

   - Insert, Delete в любой момент времени t
   - Query в любой момент времени t

Введем параметры для измерения эффективности ретрактивной структуры.

- m – количество изменений за все время
- r – при обращении к моменту времени t, количество операций, примененных к структуре после t
- n – максимальное количество одновременно находящихся в структуре элементов за все время

Теперь давайте оценим overhead на выполнение запроса, если мы делаем структуру ретрактивной.

Частичная ректрактивность

1) Insert(t, $update$)
   Если операции коммутативные, то Insert() делается с константным overhead-ом, так как мы всегда можем добавлять операцию в конец.
   Insert(t, $update$) $\Leftrightarrow$ Insert(t_{now}, $update$)

2) Delete(t)
   Если операция a, которую мы хотим отменить, обратима, а также все операции коммутируют, то Delete(t) $\Leftrightarrow$ Insert(t_{now}, a^{-1}).

Таким образом, если операции коммутативные и обратимые, то мы можем сделать структуру частично ретрактивной с overhead-ом $\mathcal{O}(1)$.

Еще оказывается, что структуру данных можно сделать частично ректрактивной с константным overhead-ом, если это структура данных для задачи поиска, т.е. структура с операциями insert, delete и query. В этом случае мы также можем вставлять и удалять элементы только в последний момент времени и благодаря этому получаем ретрактивность с константным overhead-ом.

Полная ретрактивность

Мы снова рассмотрим частный случай структур данных, а именно *decomposable search problems*, и научимся наделять их полной ретрактивностью.

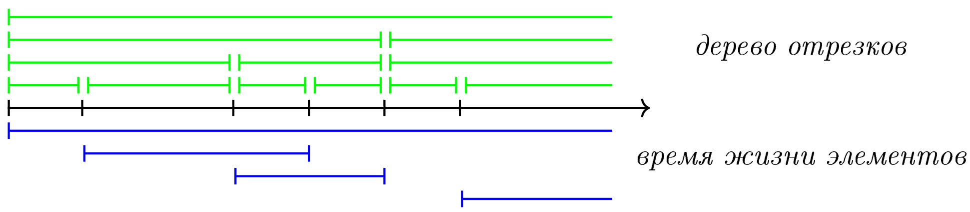
Напоминание: decomposable search problem (d.s.p.) — это задача поиска с дополнительным условием $Q(x, D \cup D') = Q(x, D) \blacklozenge Q(x, D')$, где операция $\blacklozenge$ выполняется за константное время.

Theorem 29. *Структуру данных для d.s.p. с операциями insert, delete и query, работающими за время $T(n)$ и использующими $S(n)$ памяти, можно сделать полностью ретрактивной, при этом все операции будут работать за время*

- $\mathcal{O}(T(m))$, если $T(m) = \Omega(n^{\varepsilon})$, $\varepsilon > 0$
- $\mathcal{O}(T(m) \log m)$, иначе

и использовать $\mathcal{O}(S(m) \log m)$ памяти, где m – это общее количество изменений.

Доказательство. Отметим все моменты времени, когда произошла какая-то операция. Таким образом, весь интервал от начального момента времени до последнего разбился на отрезки. Будем хранить их в в дереве отрезков, представляющем из себя сбалансированное бинарное дерево, то есть сами наши отрезки будут в листьях дерева. Каждому элементу сопоставим интервал времени, в который он присутствует в структуре. Этот интервал покрывается логарифмическим количеством узлов нашего дерева, в них и запишем этот элемент. В каждом узле храним элементы в нашей структуре.



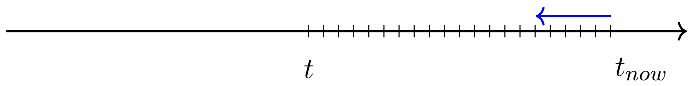
Теперь, когда нам нужно добавить или удалить элемент, происходит следующее. Если нужно, добавим новый момент времени, т.е. разделим один из отрезков на два и перебалансируем дерево. Потом добавим или удалим элемент из всех $\mathcal{O}(\log m)$ соответствующих узлов.

Если мы хотим сделать Query для времени t, то мы должны рассмотреть $\mathcal{O}(\log m)$ узлов, соответствующих t, и объединить с помощью $\blacklozenge$ результаты query в них. □

Общая техника

Theorem 30. *Структуру данных с операциями, работающими за $\mathcal{O}(T(n))$, можно сделать полностью ретрактивной, при этом операции для t_{now} будут работать за время $\mathcal{O}(T(n))$, а ретрактивные операции будут работать за $\mathcal{O}(rT(n))$. Памяти потребуется $\mathcal{O}(S(m))$.*

Доказательство. Будем запоминать все изменения, а также как изменилась структура, чтобы можно было откатить изменения.



Каждый раз, когда нам нужно сделать Insert(t, *update*) или Delete(t, *update*), будем откатывать изменения от t_{now} до t, затем применять новое изменение и применяем все r изменений снова. □

Theorem 31. *Существует структура данных в модели straight-line-program с операциями update и query, работающими за $\mathcal{O}(1)$, такая, что любая частично ретрактивная структура в модели history-dependent algebraic-computation-tree, integer-RAM или real-RAM будет тратить $\Omega(r)$ времени на update или query, причем как в худшем случае, так и амортизированно.*

Доказательство. Рассмотрим структуру, которая хранит два числа x и y, изначально равные нулю, и поддерживает следующие операции:

- $x \mathrel{+}= c$
- $y \mathrel{+}= c$
- $y = x \cdot y$

Покажем, что эта структура нам подходит. Для этого рассмотрим последовательность операций:

$y \mathrel{+}= a_n$
$y = x \cdot y$
$y \mathrel{+}= a_{n-1}$
$y = x \cdot y$
$\dots$
$y \mathrel{+}= a_0$
Query(t_{now}, y)

Так мы получим значеним многочлена $a_n x^n + \dots + a_1 x + a_0$ в точке 0.

После чего сделаем ретрактивный Insert($t = -1$, "$x \mathrel{+}= c$"). Теперь Query(t_{now}, y) уже дает нам значение этого многочлена в точке c.

И так мы можем считать значение многочлена в любой точке.

Но для вычисления многочлена в точке (при известном заранее многочлене, но неизвестной точке) есть нижняя оценка $\Omega(n)$ в любой модели из условия ([**frandsen2001lower**]).

Таким образом, если мы сделаем достаточное количество вызовов Insert($t = -1$, "$x \mathrel{+}= c$") и Query(t_{now}, y), то мы поймем, что нам потребуется $\Omega(r)$ времени на операцию даже амортизированно. □