Содержание

1 Бинарное дерево поиска

Записал: Борис Золотов

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 $\mathcal{O}(\log n)$.

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

Хотим научиться поддерживать $\pm$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

Записал: Борис Золотов

Источники: [GR93; And89]. Мы в основном опираемся на [GR93].

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)
    $depth(x) \leq \left\lfloor \log_{\frac{1}{\alpha}} \text{size} \right\rfloor + 1$
condition WEAK α-HEIGHT(node x)
    $depth(x) \leq \left\lfloor \log_{\frac{1}{\alpha}} \text{maxSize} \right\rfloor + 1$ ▷ 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,

$$\text{maxSize} := \max\{\text{maxSize}, \text{size}\}.$$

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.

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

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

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

□

Теорема 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 $\alpha \cdot \text{size}(T)$ 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

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 и удаления работают за амортизированное время $\mathcal{O}(\log n)$.

3 Splay tree

Записал: Никита Гаевой

Оригинальная статья: [ST85]

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 delete через операцию splay, а потом будем разбираться со splay. Для delete нам понадобится операция splay_front(node). Эта операция делает splay для наименьшего ключа в поддереве.

```
1: procedure INSERT(x)
2:     standard_insert(x)
3:     splay(x)
4: procedure GET(x)
5:     splay(x)
6: procedure DELETE(x)
7:     splay(x)
8:     splay_front(root.right)
9:     standard_delete(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_delete(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,

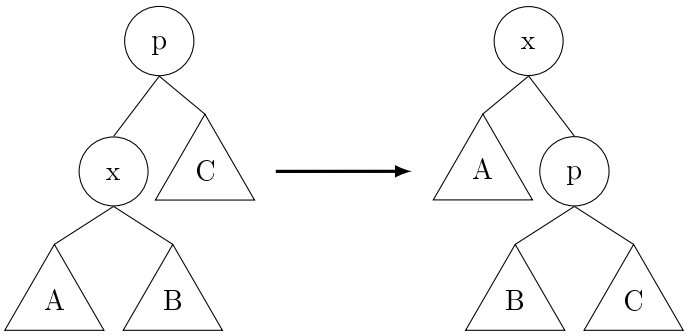


Рис. 1: Zig

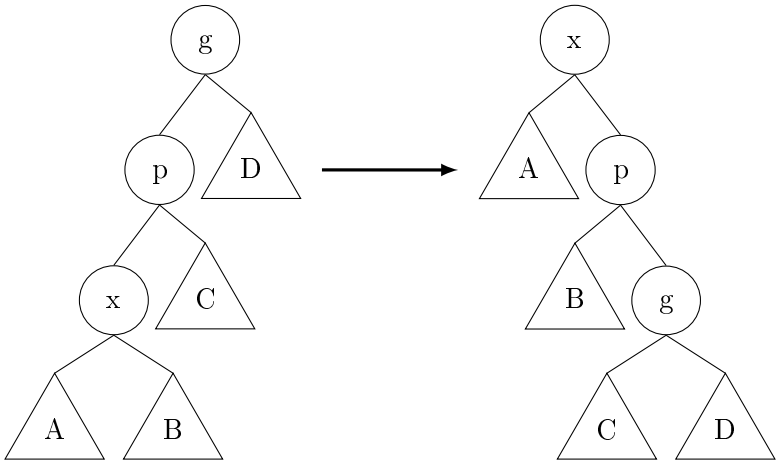


Рис. 2: Zig-zig

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 добавленная 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

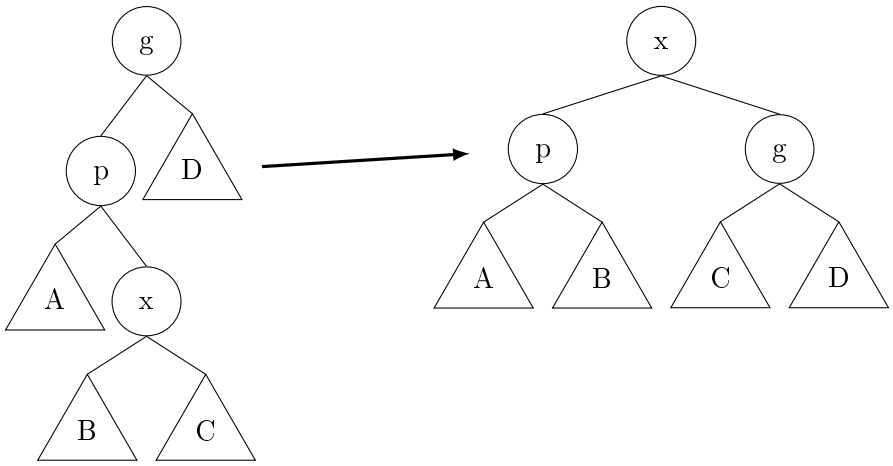


Рис. 3: Zig-zag

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: nodes $\to \mathbb{R}_+$. 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, 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.splay $= \Delta\Phi + \#\text{rotations}$ (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:

Теорема 3. *am.splay* $\le 3(r'(x) - r(x)) + \mathcal{O}(1)$

Доказательство. Надо оценить $\Delta\Phi$ для каждого из вращений. Узлы x, p, g как на рис. 1, 2 и 3, соответственно.

Zig:

$$\begin{aligned}
\Delta\Phi &= r'(p) - r(p) + r'(x) - r(x) \\
&= r'(p) - r(x) && \text{поскольку } r'(x) = r(p) \\
&\le r'(x) - r(x) && \text{поскольку } p \text{ ниже } x \text{ после вращения}
\end{aligned}$$

Дополнительно стоит отметить, что $r'(x) \ge r(x)$ 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:

$$\begin{aligned}
\Delta\Phi &= r'(g) - r(g) + r'(p) - r(p) + r'(x) - r(x) \\
&= r'(g) + r'(p) - r(p) - r(x) \\
&\le r'(g) + r'(x) - 2r(x) && \text{из структуры дерева} \\
&\le 3(r'(x) - r(x)) - 2 && \text{поскольку } r'(g) + r(x) \le 2(r'(x) - 1)
\end{aligned}$$

Осталось показать, что $r'(g) + r(x) \le 2(r'(x) - 1)$.

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

Zig-zag:

$$\begin{aligned}
\Delta\Phi &= r'(g) - r(g) + r'(p) - r(p) + r'(x) - r(x) \\
&= r'(g) + r'(p) - r(p) - r(x) \\
&\le r'(g) + r'(p) - 2r(x) && \text{из структуры дерева} \\
&\le 3(r'(x) - r(x)) - 2 && \text{поскольку } r'(g) + r'(p) \le 2(r'(x) - 1)
\end{aligned}$$

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 одиночных вращений. Таким образом, $\text{am.splay} = \Delta\Phi + \#\text{rotations} \le 3(r'(x) - r(x)) + \mathcal{O}(1)$, 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.

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

Доказательство. Берем $w(x) = 1$. □

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

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 оптимальные динамические деревья остается открытой проблемой.

Гипотеза 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 $1 \dots n$.

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

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. □

Следствие 8 (Working Set Property). *Пусть $t(x)$ — количество времени, которое прошло с последнего доступа к элементу x. Тогда время обработки последовательности запросов $\mathcal{O}(m + n \log n + \sum_x \log(t(x) + 1))$.*

Доказательство. Берем $w(x) = \frac{1}{(t+1)^2}$. В этом случае сами веса меняются со временем, но их сумма остается постоянной. Аналогично предыдущему $W = \sum_{i \leq n} = \frac{1}{i^2} = \mathcal{O}(1)$, а потенциал бывает отрицательным не более, чем на $\mathcal{O}(n \log n)$. □

Теорема 9 (Dynamic Finger Theorem). *Аналогично, но теперь f, «finger» — элемент, к которому обращались предыдущим запросом (и, следовательно, находящийся в корне). Тогда время работы $\mathcal{O}\left(m + n + \sum_{x\ -\ запрос} \log(|x - f| + 1)\right)$.*

Мы не доказывали. Доказательство трудное и объемное, можно посмотреть в этих двух статьях: [Col+00; Col00].

Теорема 10 (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 (Теорема 9). □

4 Об оффлайн деревьях поиска: введение

Записал: Владислав Макаров

Исходная статья: «The Geometry of Binary Search Trees» [Dem+09].

4.1 Оффлайн деревья поиска и геометрическое представление

Пусть у нас есть двоичное дерево поиска над ключами $1, 2, \ldots, n$ (в частности, все ключи в дереве различны) и последовательность $S = (s_1, s_2, \ldots, s_m)$ запросов поиска к нему. При этом все запросы ищут ключ, который действительно есть в дереве, то есть $1 \leqslant s_i \leqslant n$ для всех i. 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 модифицировать дерево с помощью вращений в процессе ответа на запросы.

Нам разрешено делать следующие вещи:

1) Переходить по указателю.

2) Делать одинарное вращение в этой вершине ($\mathrm{zig}(x)$, он же $\mathrm{zag}(x)$.

Обработку каждого запроса мы начинаем в корне и хотим посетить вершину с запрошенным ключом.

Определение 2. Последовательность таких действий (вместе с исходным деревом до всех запросов) для фиксированной последовательности запросов S называется *BST-алгоритмом*.

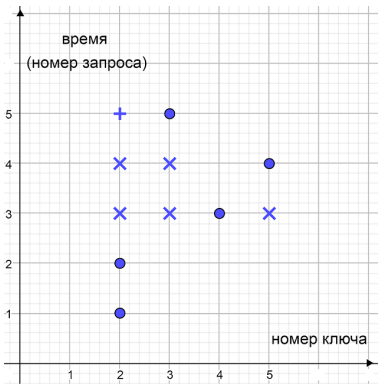
Определение 3. Цена операции поиска — количество посещённых узлов.

Замечание. Поскольку для того, чтобы сделать вращение в вершине, нам нужно её посетить, учитывать количество вращений в цене операции поиска не нужно, если мы считаем с точностью до константы.

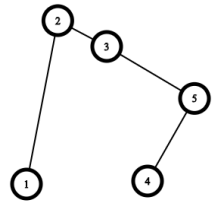
Определение 4. $\mathrm{OPT}(S)$ — минимальная суммарная цена выполнения BST-алгоритма, обрабатывающего S.

Значение $\mathrm{OPT}(S)$ мы не умеем искать (даже с точностью до мультипликативной константы) за полином от n и m. 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.

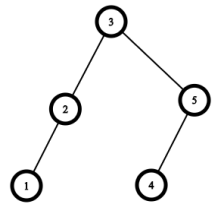
Это можно переформулировать в геометрических терминах. Рассмотрим координатную плоскость с ключами по оси x и моментами времени (то есть номерами запросов) по оси y. Для данной последовательность запросов S отметим все точки (s_i, i) на плоскости (мы обязаны посетить ключ s_i при обработке i-того запроса, 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 данного BST-алгоритма — количество отмеченных точек.



(a) Геометрическое 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 zig(3).

Рис. 4: Пример геометрического представления BST-алгоритма

Определение 5. Множество отмеченных точек — *геометрическое представление* данного BST-алгоритма.

Замечание. Понятно, что у разных BST-алгоритмов могут быть одинаковые представления.

На рисунке 4a видно геометрическое представление BST-алгоритма для последовательности запросов $S = (2, 2, 4, 5, 3)$, который не совершает никаких вращений, а в качестве исходного дерева использует дерево с рисунка 4b. От плюсика (точки $(2, 5)$) можно избавиться, если при обработке четвёртого запроса сделать операцию zig(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 $\mathcal{O}(\mathrm{OPT}(S) + n)$ 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.

Определение 6. Если a и b — точки на плоскости, то $\mathrm{rect}(a, b)$ — замкнутый (то есть содержащий границу) прямоугольник со сторонами, параллельными осям координат, (возможно, вырожденный), натянутый на точки a и b как на противоположные углы.

Определение 7. Множество E точек с целыми координатами на плоскости 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 $E \setminus \{a, b\}$.

4.2 Эквивалентность BST-алгоритмов и arborally satisfiable множеств

Теорема 11. *Если множество точек E соответствует какому-то BST-алгоритму, 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 *сразу 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 *сразу перед* обработкой j-того запроса (и тогда мы должны посетить a просто для того, чтобы дойти до b из корня), либо вершина a перестала быть предком b в какой-то момент времени на отрезке $(i + 1, j)$, а для этого мы должны были её посетить и сделать вращение в её ребёнке. Противоречие.

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 $(x(c), i)$, а она лежит в прямоугольнике $\mathrm{rect}(a, b)$. При этом возможно, что c и b совпадают как *вершины*, но в этом случае мы найдём точку $(x(c), i) = (x(b), i)$, а не $(x(b), y(b)) = (x(b), j)$, то есть найденная нами точка не совпадает с углами $\mathrm{rect}(a, b)$. Противоречие.

□

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 геометрическим представлением BST-алгоритма (позже мы поймём, что каждое arborally satisfiable множество — геометрическое представление какого-то BST-алгоритма, но не будем торопить события).

Лемма 12. *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* $\mathrm{rect}(a, b)$*, смежных с a.*

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 11: теорема 15. Для её доказательства нам понадобится понимать некоторые базовые вещи про декартовы деревья.

Определение 8. *Декартово дерево* (treap) — это двоичное дерево поиска, в котором у каждой вершины кроме ключа есть ещё и другой параметр — *приоритет*, при этом декартово дерево удовлетворяет свойству кучи на минимум по приоритетам.

Определение 9. Под *декартовым деревом на парах* $(\mathrm{key}_i, \mathrm{priority}_i)$ будем понимать любое декартово дерево с данными мультимножеством пар «ключ–приоритет».

Лемма 13. *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.

Лемма 14. *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 $\mathcal{O}(n)$ вращений.*

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

Как построить триангуляцию по дереву? Пусть в корне дерева находится ключ x. Тогда добавим в триангуляцию треугольник, построенный на корневой стороне и вершине с номером x. Тогда по одну сторону от этого треугольника находятся вершины с номерами $[1, x-1]$, а по другую — вершины с номерами $[x+1, n]$. Это как раз ключи из левого и правого поддеревьев корня соответственно. Более того, в обоих получившихся многоугольниках можно естественным образом выделить новую корневую

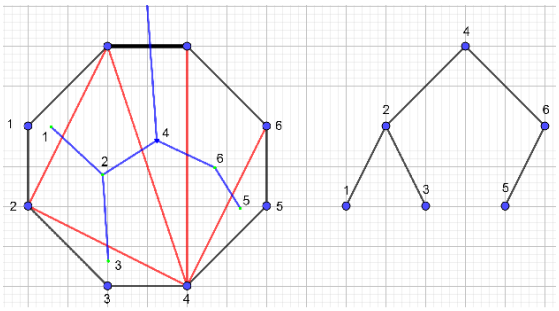


Рис. 5: Триангуляция и соответствующее ей двоичное дерево поиска.

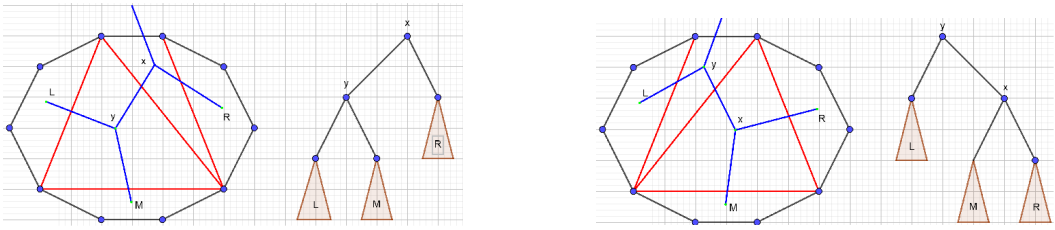


Рис. 6: Триангуляция и соответствующей ей дерево на левом изображении переходят в триангуляцию и дерево на правом изображении при операциях flip и zig соответственно.

сторону, а именно, соответствующую сторону исходного треугольника. Это позволяет продолжить построение триангуляции рекурсивно (см. рисунок 5).

Чтобы построить дерево по триангуляции, возьмём единственный треугольник, содержащий корневую сторону. Так мы нашли ключ, стоящий в корне дерева. Дальше запускаемся рекурсивно от частей, на которые треугольник разбил многоугольник — так мы получим левое и правые поддеревья корня.

Ещё более интересное свойство этой биекции состоит в том, что вращения дерева поиска соответствуют операциям flip в триангуляции: поменять проведённую диагональ в четырёхугольнике (см. рисунок 6).

Вместо того, чтобы переводить одно дерево вращениями в другое, можно привести оба к одному и тому же, а потом обратить вторую последовательность вращений. В терминах триангуляций, мы хотим привести две триангуляции операциями flip к какой-то выделенной. В качестве выделенной триангуляции возьмём такую, в которой все треугольники содержат выделенную вершину, скажем, вершину с номером 1 (эта триангуляция выглядит как веер).

Алгоритм приведения любой триангуляции к вееру: пока выделенная вершина принадлежит хотя бы одному четырёхугольнику, в котором она не конец проведёной диагонали, делаем flip этого четырёхугольника. Несложно видеть (то есть мне лень это расписывать, потому что ещё придётся картинки рисовать), что процесс завершится только когда триангуляция станет веером. □

Замечание. Есть и более прямолинейные способы доказать лемму 14, не сводящие деревья к триангуляциям. Но аналогия между деревьями и триангуляциями — вообще очень полезная штука.

Теорема 15. *Если E — arborally satisfiable, то существует BST-алгоритм, геометрическое 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 $(x, i) \in E$).

Мы хотим, чтобы сразу перед i-тым шагом наше дерево было каким-то декартовым деревом T_i 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. Чтобы добиться этого, нужно выбрать T_1 в качестве исходного дерева нашего BST-алгоритма и научиться перестраивать T_i в T_{i+1}, посещая 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 14) 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 нарушиться 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 12, 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. □

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 геометрическим 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 $\mathcal{O}(|E| + n)$ (получить в точности геометрическое 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.

Определение 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, …, 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 $\mathcal{O}(n)$ времени на построение (make_tree) и полное разрушение (n операций split_tree) дерева.

Замечание (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 $\mathcal{O}(n^2)$ времени, но затронуть вершины только $\mathcal{O}(n)$ раз в процессе полного разрушения внутреннего дерева, это бы нас более-менее устроило. Но оказывается, что мы можем потратить $\mathcal{O}(n)$ времени (и, следовательно, лишь $\mathcal{O}(n)$ 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 [Dem+09]. Более того, есть гипотеза (см. статью Лукас [Luc88]), 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.

Определение 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 вершины в них 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.

3) Отношение «отец–сын» на суперузлах определяется естественным образом. При этом у суперузла (в отличие от обычной вершины) может быть много детей.

Замечание. 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 декартова 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.

Замечание. 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 декартова 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.

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-дереву, 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 ассоциированной 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:

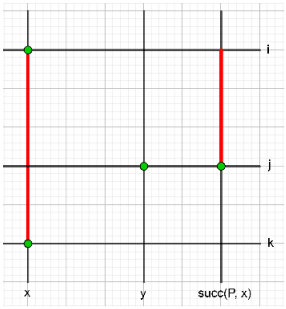


Рис. 7: Зелёные точки соответствуют точкам из E, красные отрезки — отрезкам, на которых точно нет точек из E из-за того, что мы знаем значения $\rho(\cdot,\cdot)$. Точка (y, j) появляется из леммы 12 для (x, i) и $(\mathrm{succ}(P, x), j)$.

Лемма 16. *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 $\mathrm{rect}((x, i), (\mathrm{succ}(P, x), j))$. 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 $\mathrm{succ}(P, x)$ (см. рисунок 7).

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 16, $\text{succ}(P, x)$ и $\text{pred}(P, x)$ 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 $(\text{pred}(P, x), \text{succ}(P, x))$, 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 его 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? И в формулировке, и в доказательстве леммы 16 мы неявно пользуемся тем, что у соседних суперузлов разный приоритет, так что замести это под ковёр не получится.

Чтобы разобраться с этой деталью, заметим, что для любых двух суперузлов в G_i для одного и того же приоритета есть вершина G_i, их *разделяющая*: такая вершина, что один суперузел лежит в её левом поддереве, а другой — в правом (в противном случае один суперузел — предок другого суперузла с тем же приоритетом). Аналогичным образом, когда операция split_tree разбивает один суперузел на два, корень split-дерева становится разделяющей вершиной для этих суперузлов. При этом приоритеты суперузлов в процессе преобразования G_i в G_{i+1} не меняются, а приоритеты разделяющих вершин либо не меняются (для разделяющих вершин не из τ_i), либо увеличиваются (для разделяющих вершин из τ_i). Следовательно, разделённые суперузлы остаются в поддереве разделившей их вершины и в дереве G_{i+1}. Более того, по свойству двоичного дерева поиска, один из этих суперузлов должен оказаться в её левом поддереве, а другой — в правом. Следовательно, суперузлы, которые были разделены в G_i или стали разделены из-за операций split_tree, остаются разделёнными в G_{i+1}. Наконец, разделённые суперузлы точно не могут быть соседними.

Всего мы сделали $\mathcal{O}(|E| + n)$ вращений. Действительно, на i-том шаге мы делаем $|\tau_i|$ операций split_tree (амортизированно $\mathcal{O}(1)$ времени) и одну операцию make_tree на $|\tau_i|$ вершинах ($\mathcal{O}(|\tau_i|)$ времени). Слагаемое $\mathcal{O}(n)$ появляется из-за амортизации: каждое не разрушенное полностью split-дерево могло «съесть» $\mathcal{O}(\texttt{своего размера})$ 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

Записал: Борис Золотов

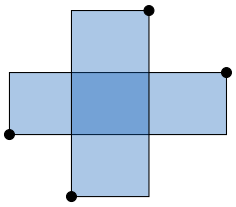
5.1 Основные определения и предваряющие результаты

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.

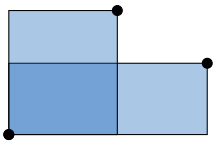
Определение 12. $E(P)$ — множество всех вершин, которые мы посетим в процессе поиска вершин с ключами из P. $E = P \cup X$, X — множество «лишних» вершин.

Определение 13. OPT — минимальный размер $E(P)$ (обозначение множества P будем опускать, и так по контексту ясно).

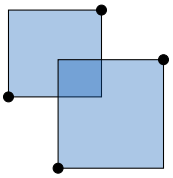
Определение 14. Пусть $a, b \in \mathbb{R}^2$. Тогда $\mathrm{rect}(a, b)$ — прямоугольник, стороны которого



(a) Эти прямоугольники независимы



(b) Эти прямоугольники независимы



(c) Эти прямоугольники **не** независимы

Рис. 8: Примеры прямоугольников, независимых и не очень

параллельны осям координат, а противоположные вершины — точки a и b. Его же будем называть *прямоугольник, определённый точками a, b.*

Определение 15. Конечное множество $G \subset \mathbb{R}^2$ называется *arborally satisfiable,* если

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

Теорема 17 (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.

Определение 16. Пусть дано множество P и его надмножество E. Два прямоугольника, определённых каждый двумя вершинами множества P, будем называть *независимыми* (смотреть Рисунок 8), 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

Определение 17. 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 9).

Рис. 9: Прямоугольники типа «+» и типа «−».

Определение 18. $\mathrm{MAX\,IND}$ — наибольшее число попарно независимых прямоугольников, определённых точками из P. Соответственно, $\mathrm{MAX\,IND}_+$, $\mathrm{MAX\,IND}_-$ — наибольшие количества попарно независимых прямоугольников фиксированного типа.

Теорема 18.

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

Прежде чем приступить к доказательству Теоремы 18, докажем следующую лемму:

Лемма 19.

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

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

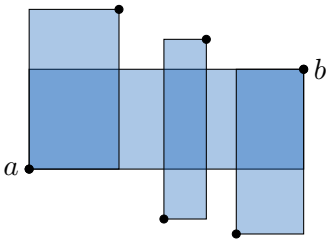
Далее мы забываем о том, что множества точек, с которыми мы работаем, — это вообще говоря выходы какой-то процедуры поиска, и рассматриваем произвольные конечные множества точек на плоскости.

Доказательство Леммы 19. 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 10a.

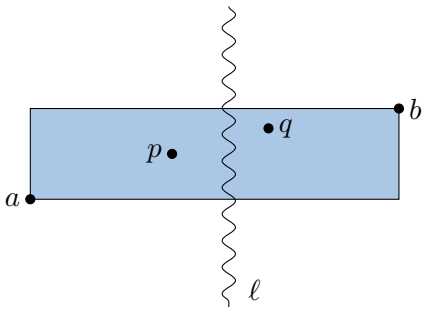
Прямоугольники, имеющие своей вершиной a, не пересекаются с прямоугольниками, имеющими своей вершиной b, потому что иначе получается случай прямо как на Рисунке 8c. Более того, оставшиеся прямоугольники тоже не могут никак налезать друг на друга, потому что опять же получится случай с Рисунка 8c. Поэтому существует вертикальная линия, пересекающая *только* выбранный нами широкий прямоугольник $\mathrm{rect}(a, b)$, обозначим её через ℓ, смотреть Рисунок 10b.

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.

Утверждение 20. *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. 10: Доказательство Леммы 19

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.

Утверждение 21. *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.

Утверждение 22. *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.

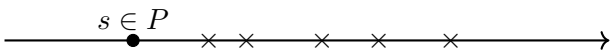


Рис. 11: Точки, добавленные в данную строку

Рассмотрим некоторую строку, в ней находится одна точка из исходного множества запросов P, смотреть Рисунок 11. 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 22. То есть на одну точку из $E(P)$ добавляется не более одного

прямоугольника. Лемма доказана. □

Доказательство теоремы 18.

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

Теперь воспользуемся тем, что максимум не меньше среднего, а также леммой 19.

$$\begin{aligned}\max(\mathrm{OPT}_+, \mathrm{OPT}_-) \;&\geq\; |P| + \frac{1}{2}(\mathrm{MAX\,IND}_+ + \mathrm{MAX\,IND}_-) \;\geq \\ &\geq\; |P| + \frac{1}{2}\cdot \mathrm{MAX\,IND}\,.\end{aligned}$$

□

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 12.

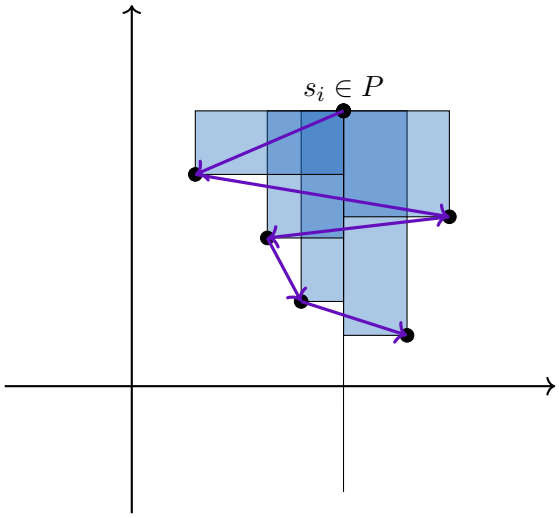


Рис. 12: Подсчёт числа пересечений с вертикальной прямой

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.

Теорема 23.

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

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 13.

Все прямоугольники в объединении, легко видеть, будут попарно независимы. Осталось лишь применить теорему 18. □

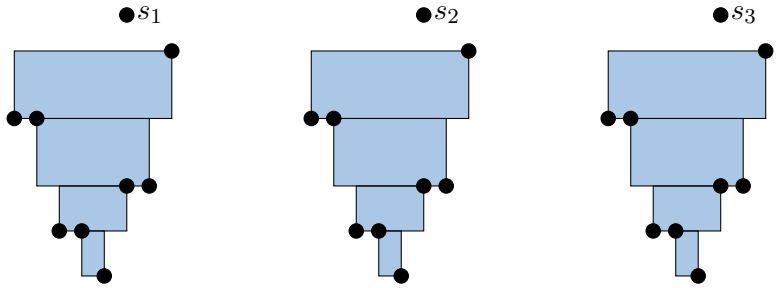


Рис. 13: Набор попарно независимых прямоугольников

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.

Теорема 24.

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

Доказательство. Следует из Теоремы 23. □

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. 14: Bit-reversal sequence делает нижнюю оценку бессмысленно большой

6 Tango деревья

Записал: Борис Золотов

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 15.

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.

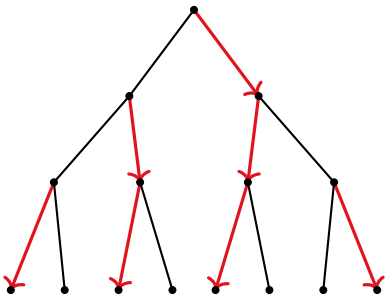


Рис. 15: Tango-дерево представлено в виде объединения путей

7 Link-Cut trees

7.1 Описание структуры, план действий

Записала: Анастасия Софронова

Наша цель — поддерживать структуру данных, которая умеет хранить лес корневых 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 [ST83] описано, как добиться асимптотики $\mathcal{O}(\log n)$ на операцию в худшем случае. Мы же изучим link-cut trees, работающие за амортизированное $\mathcal{O}(\log n)$ ([Tar84]).

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 (p — указатель на нижний конец пути).

- 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) — операция, обратная операции join: отрезать рёбра, ведущие из v в предка и в потомка в пути.
- findPathCost(p), addPathCost(p, 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, состоящие из пар $(tail(p), successor(p))$). 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).

```
1: procedure makeTree(u)
2:     makePath(u)
3: procedure findRoot(u)
4:     findTail(expose(u))
5: procedure findCost(u)
6:     expose(u)
7:     findPathCost(u)
8: procedure addCost(u, x)
9:     expose(u)
10:    addPathCost(u, x)
11: procedure link(u, w)
12:    join(ø, expose(u), expose(w))
13: procedure cut(v)
14:    expose(v)
15:    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.

```
1: procedure expose(u)
2:     p := ø                          ▷ Здесь будем накапливать наш текущий путь
3:     while u ≠ ø do
4:         w := successor(findPath(u))  ▷ Запомним следующий сверху путь в дереве
5:         (q, r) := split(u)                  ▷ Отрежем у u сплошное ребро вниз
```

```
6:        if q ≠ ∅ then                    ▷ q — часть пути, проходящего через u, ниже u
7:            successor(q) := u            ▷ Теперь ребро из q в u — пунктирное
8:        p := join(p, u, r)               ▷ А ребро из u в наш текущий путь — сплошное
9:        u := w                           ▷ Перейдём к вершине следующего пути
10:     successor(p) := ∅
```

Операцию, которая происходит в теле while, назовём splice.

Теорема 25. *Пусть выполнено m операций с лесом, 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 деревьев.

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.

Обозначим $\text{size}(v)$ количество вершин в поддереве вершины v.

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 $\mathcal{O}(\log n)$, то на каждую из m операций может произойти не более $\mathcal{O}(\log n)$ «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). Корень дерева соответствует пути. 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 $\Delta\text{min}(x) = \text{mincost}(x)$

Здесь $p(x)$ — предок x в Splay-дереве.

```
1: procedure makePath(u)
2:     makeSplayTree(u)
3: procedure findPath(v)
4:     splay(v)
5:     return(v)
6: procedure findPathCost(v)
7:     while right(v) ≠ 0 and min(right(v)) = 0 or left(v) ≠ 0 and min(left(v)) = 0 do
8:         if right(v) ≠ 0 and min(right(v)) = 0 then
9:             v := right(v)
10:        else
11:            v := left(v)
12:    splay(v)
13:    return(v, Δmin(v))
14: procedure addPathCost(v, x)
15:    Δ(min)(v) = Δ(min)(v) + x
16: procedure join(p, v, q)
17:    v.left = p
18:    v.right = q
19: procedure split(v)
20:    splay(v)
21:    cut(v, v.left)
```

22: 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

Записал: Елизаров Никита

Полное описание всех структур и доказательств можно найти в [Geo+11].

8.1 Описание структуры и чего мы хотим от этой структуры

Будем называть **упорядоченной кучей** (или просто **кучей**) дерево, где каждой вершине v сопоставлен **рациональный** ключ **l(v)**, со следующим свойством: $l(v) \geqslant l(parent(v))$.

Будем говорить, что $v > w$, если $l(v) > l(w)$.

В данной секции мы будем рассматривать **лес упорядоченных куч** (то есть не одну такую кучу, а несколько). Мы хотим уметь поддерживать следующие 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 v и w — не в одном дереве, то возвращает пустое множестве;
- $insert(v, x)$ — создание нового дерева с одной вершиной v и ключом x;
- $delete(v)$ — удалить v, если v — лист;
- $link(v, w)$ — подвесить v к w, причём v — корень какого-то дерева из леса, а w должна быть в другом дереве. Также $l(v) \geqslant l(w)$;
- $cut(v)$ — удалить ребро между v и его родителем; если v является корнем, то ничего не делаем;
- $merge(v, w)$ — если v и w находятся в разных деревьях, то мы сделаем одну операцию $link(root(v), root(w))$ (или $link(root(w), root(v))$, в зависимости от того, какой из корневых ключей больше). Полученное дерево (или изначальное дерево, если v и w были в одном дереве) обозначим за T. Тогда нас интересуют два пути:
  - P — путь от v к корню дерева T;
  - Q — путь от w к корню дерева T.

  Мы хотим совместить P и Q, чтобы v и w оказались на одном пути от корня T к какому-то листу. При этом, мы хотим, чтобы свойство кучи сохранилось.

  Примеры двух **merge** показаны ниже:

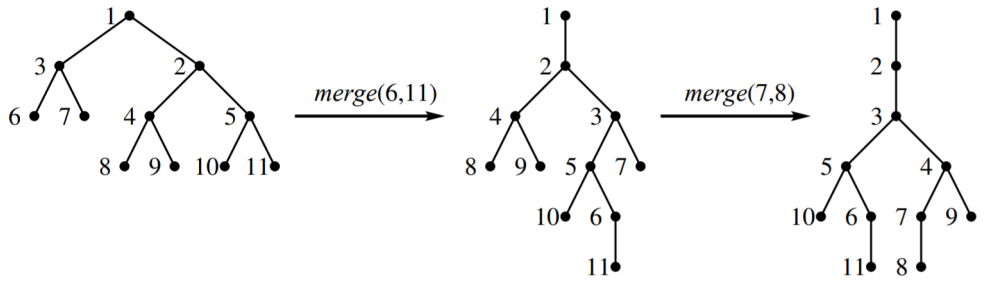


Рис. 16: merge

Теперь поговорим немного про то, как именно мы будем это реализовывать. Мы будем использовать такую же реализацию динамического дерева, как это было в **link – cut trees** (подробнее в предыдущей секции). За одним исключением: мы будем использовать небинарные **link-cut деревья.** Известно, что для них верны все те же оценки, что мы доказывали для бинарных **link-cut деревьев.** Тогда все операции, кроме **merge** и **NCA** мы делать уже умеем. Для реализации **merge** нам нужно ввести еще одну операцию — **topmost**$(\mathbf{x}, \mathbf{w})$.

- $topmost(v, w)$ — возвращает ближайшую к $root(v)$ вершину t на пути $[root(v), v]$, такую, что $l(t) > l(w)$. Предполагается, что $l(v) > l(w)$.

Покажем, что мы умеем амортизированно делать **NCA** и **topmost** за $\mathcal{O}(\log n)$:

```
1: procedure NCA(v, u)
2:     expose(v)
3:     expose(w)
4:     return successor(findTail(v))
```

```
1: procedure TOPMOST(v, w)
2:     expose(v)
3:     return Binary_Search(l(w), Splay_Tree(v))
```

Здесь **Binary_Search**($\cdot$, $Splay\_Tree(v)$) — это бинарный поиск в **splay-дереве**, которое соответствует сплошному пути, содержащему вершину **v**.

Для полного понимания, нужно вспомнить, что **splay-дерево** строилось по «неявным» ключам для пути (а именно: ключом вершины был её порядковый номер в пути). Но, мы будем строить наши **splay-деревья** по ключам из кучи. В силу того, что на любом пути ключи возрастают, порядок и структура дерева не изменятся.

8.2 Реализация merge

Для начала заметим, что **link** — это тот же самый **merge**. Поэтому, будем считать, что все операции **link** это операции **merge**.

```
1: procedure MERGE(v, u)
2:     if root(v)≠ root(w) then
3:         if l(v) ⩾ l(w) then
4:             link(v, w)
5:         else
6:             link(w, v)
7:     u := NCA(v, w)                          ▷ ищем общего предка v и w
8:     if u = v or u = w then
9:         return ∅ ▷ если общий предок совпадает с v или w, то пути уже совмещены
10:    else
11:        x := topmost(v, u)
12:        y := topmost(w, u)
13:        if x < y then          ▷ нужно, потому что будем вставлять на первом шаге
14:            swap(x, y)                      ▷ путь от x до v в путь от y до w
15:            swap(v, w)
16:        if u ≠ ∅ then
17:            cut(x)
18:        while x < u do                              ▷ начало merge step
19:            t := topmost(w, x)                  ▷ l(z) ⩽ l(x), а l(t) уже нет
20:            link(x, parent(t))                  ▷ значит x должен быть ребёнком z
21:            cut(t)
22:            y := x                  ▷ теперь мы хотим вставлять путь от t до w
23:            x := t                                          ▷ в путь от x до v
24:            swap(v, w)                                  ▷ конец merge step
```

25: link(x, w)

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

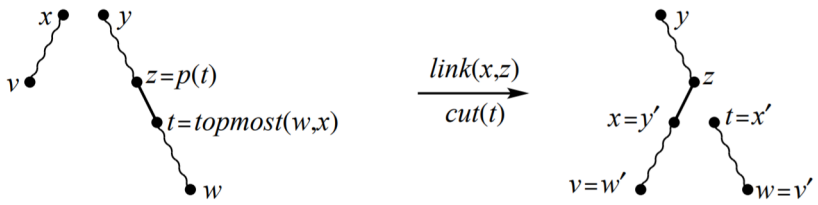


Рис. 17: merge step

Чтобы оценить количество шагов, достаточно оценить количество изменений родителя какой-либо вершины. 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 — в **link**(**x**, **w**), и еще может быть одно, если они из разных деревьев мы делаем **initial link**.

Теперь докажем следующую лемму.

Лемма 26. *Количество изменений родителей за все* ***merge*** *— это* $\mathcal{O}(m \log n)$.

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

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

$$\textbf{am. cost} = cost + \Delta\Phi$$

Потенциал следующий:

- каждому ребру e даём 1 потенциала(Φ_e);
- от каждого ребра $(v, parent(v))$ дадим $\log(v - parent(v))$ родителю и $\log(v - parent(v))$ 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 **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 $\log n$ каждой корневой вершине, которые мы соединяем, 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 рисунке 17. 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 знак нестрогий. Последнее неравенство снова видно из рисунка 17, так как $z = parent(x)$.

Математической магией этих трёх неравенств получаем, что:

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

Заметим, что гарантированно выполняется одно из этих двух условий. Более того, потенциал меняется только у вершин t и $parent(t)$. Разберём каждое из них:

1) Рассмотрим вершину t. Тогда потенциал, приходящий от родителя для неё уменьшается хотя бы на $\log(t - parent(t)) - \log(t - parent'(t)) = \log \frac{t-parent(t)}{t-parent'(t)} \geqslant 1$. Потенциал, приходящий от детей для неё не изменился. Где еще произошло изменение потенциала? Мог изменится потенциал, приходиящий от детей для вершины $parent(t)$: он был $\log(t - parent(t))$, а стал $\log(x - parent(t))$. Но $t - parent(t) > x - parent(t)$, а значит он снова мог только уменьшиться. Значит одно изменение родителя скомпенсировалось уменьшением потенциала хотя бы на 1.

2) Рассмотрим вершину $parent(t)$. Тогда потенциал, приходящий от детей для неё уменьшается хотя бы на $\log(t - parent(t)) - \log(x - parent(t)) = \log \frac{t-parent(t)}{x-parent(t)} \geqslant 1$. Потенциал, приходящий от родителей для неё не меняется. Где еще произошло изменение потенциала? Мог измениться потенциал, приходящий от родителей вершины t : он был $\log(t - parent(t))$, а стал $\log(t - parent'(t))$. Но $t - parent(t) > t - parent'(t)$, а значит он снова мог только уменьшиться. Значит одно изменение родителя скомпенсировалось уменьшением потенциала хотя бы на 1.

Таким образом, **am. cost** за один **merge** $= \mathcal{O}(\log n)$. Действительно, **initial link** даёт нам $2\log n + 1$, каждый **merge step** даёт нам не более 0 и, в конце произолшло одно дополнительное изменение родителя. □

Наконец, итоговое изменение потенциала $\Delta\Phi$ больше нуля, так как в начале был ноль, а в конце что-то положительное. Значит $total\_cost \leqslant total\_am.cost - \Delta\Phi \leqslant total\_am.cost = \mathcal{O}(m \cdot \log n)$. Теперь, вспоминая, что все операции в нашем дереве работают за $\mathcal{O}(m \cdot \log(n))$ мы получаем оценку $\mathcal{O}(m \cdot \log^2 n)$ на все операции **merge**.

8.3 Merge без операций cut

Теперь мы хотим получить амортизированный $\mathcal{O}(\log n)$ вместо $\mathcal{O}(\log^2 n)$. Также как и в прошлой секции, будем представлять дерево в виде объединения непересекающихся по вершинам сплошных путей, соединённых пунктирными рёбрами.

Определение 19. Определим ранг вершины, как $r(v) := \lfloor \log size(v) \rfloor$, где $size(v)$ — это сумма по ключам в поддереве v, включая саму v.

Ребро (v, w) будет сплошным, если $r(v) = r(w)$. Иначе ребро будет пунктирным. Пример разбиения на пути можно увидеть на рисунке 18.

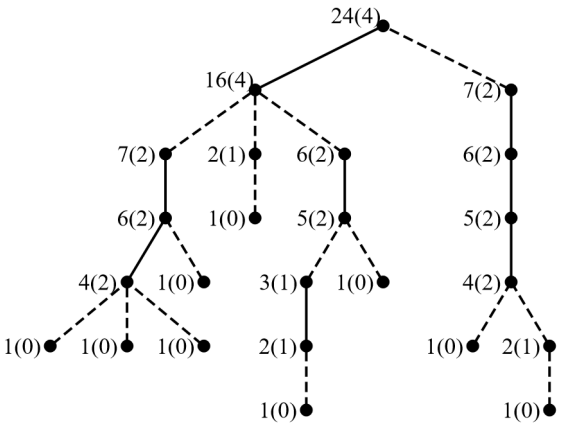


Рис. 18: разбиение дерева на пути по рангам

Как и в прошлой подсекции, мы считаем, что все ключи это числа $[1, \dots n]$. Из этого следует, что различных рангов не может быть больше, чем $\log n + 1$.

8.3.1 Новая структура дерева

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 будущем будем обозначать начало пути как **top**), в котором вершина находится;
- **dashed size(x)** $:= 1 + \sum\limits_{u \text{ — пунктирный ребёнок}} size(u)$

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 (так как в нашем случае ранги у всех вершин из сплошных путей равны — это ровно критерий, по которому составлялись сплошные пути);
- поисковая структура — структура, которая будет уметь выполнять три функции (на ближайшие несколько подподсекций будем считать, что это чёрная коробка, которая просто умеет делать представленные функции):

- удалять узлы в начале пути за $\mathcal{O}(1)$;
- добавлять одиночные узлы перед заданным за $\mathcal{O}(1)$;
- возвращать **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:

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

На нашем дереве мы хотим поддерживать следующие операции:

- $insert(x, w)$ — просто создаём дерево, состоящее из одной вершины и одного пути. Соответсвенно инициализируем поисковую структуру и *head*. В худшем случае это делается за $\mathcal{O}(1)$;
- $parent(v)$ — в нашем случае можно просто перейти по указателю за $\mathcal{O}(1)$;
- $root(v)$ — мы немного изменим выход операции $root(v)$. Теперь мы будем возвращать не только сам корень, но стек, состоящий из **top** всех сплошных путей, которые находятся на $[root(v), v]$. Так как у каждого элемента есть указатель на **top** своего пути, а также беря в рассмотрение тот факт, что сплошных путей не больше, чем различных рангов (каждому сплошному пути соответсвует ровно один ранг), получаем, что $root(v)$ работает за $\mathcal{O}(\log n)$. Будем обозначать выход $root(v)$ за S_v;
- ```
1: procedure NCA(v, w)
2:     if Top(S_w) ≠ Top(S_v) then
3:         return ∅
4:     else
5:         while Top(S_v) = Top(S_w) do
6:             pop(S_v)
7:             pop(S_w)
8:         if S_v = S_w = ∅ then
9:             return min{v, w}
10:        if S_v = ∅ and S_w ≠ ∅ then
11:            return min{v, parent(Top(S_w))}
12:        if S_v ≠ ∅ and S_w = ∅ then
13:            return min{w, parent(Top(S_v))}
14:        if S_v ≠ ∅ and S_w ≠ ∅ then
15:            return min{parent(Top(S_v)), parent(Top(S_w))}
```

Заметим, что размер каждого стека $S_v$ и $S_w$ не превосходит $\log n + 1$. Во время работы **NCA** мы проходим по стекам $S_v$ и $S_w$ не более раза. Значит **NCA** работает за $\mathcal{O}(\log n)$.
```

- $link(v, w)$ — рассмотрим отдельно;
- $merge(v, w)$ — рассмотрим отдельно.

8.3.2 Link

Для начала заметим, что при операции **link(v,w)**, у каждого предка w увеличивается $size$ на **size(v)**. Это значит, что у всех сплошных путей выше w $size$ увеличивается одинаково, что облегчает пересчёт их рангов.

Сам **link** состоит из 4х частей:

- сначала добавляем ребро (v, w) как пунктирное ребро;
- находим множество Q — вершины, ранг которых увеличился;
- делаем все сплошные рёбра, выходящие из детей вершин из Q, пунктирными;
- делаем рёбра вершин из $Q \cup \{v\}$ в их родителя сплошными, если у них совпадают ранги.

Теперь я постараюсь описать процесс чуть более детально:

- находим S_w, делаем $parent(v) = w$ и добавляем $size(v)$ к $d(w)$ и к каждому $size(x)$, где $x \in S_w$;
- повторяем следующее, до тех пор, пока S_w не станет пустым: вытаскиваем вернхий элелемент $f \in S_w$, вычисляем его новый ранг, если ранг увеличился, то добавляем его в конец Q. Если у f есть сплошной ребёнок, то вычисляем его новый $size$ как $size(f) - d(f)$ и пушим g в S_w. Так как ранг вершины не может увеличиться, без увелечения ранга её сплошного родителя, то эта операция корректна;
- идём по Q из начала в конец: смотрим на $f \in Q$, если у f есть сплошной ребёнок g, то делаем $solid(f) = null$, добавляем $size(g)$ к $d(f)$, удаляем f из поисковой структуры $head(g)$, делаем $top(head(g)) = g$, а f делаем единичным сплошным путём. Мы корректно удаляем всё ненужное из поисковых структур, потому что идём по сплошным путям сверху вниз;
- идём по Q из начала в конец: смотрим на $f \in Q$, если у предка такой же ранг, как и у f, то делаем $solid(parent(f)) = f$, изменяя при этом $d(parent(f))$. Далее, удаляем все $head$ и поисковые структуры для f (напомню, f — одноэлементный сплошной путь), делаем $head(f) = head(parent(f))$ и добавляем f прямо под $parent(f)$. Если $rank(f) = rank(v)$, то делаем $head(f) = head(v)$, $top(head(v)) = f$ и добавляем f прямо над v. Завершающий шаг такой: если v и w имеют одинаковый ранг, то мы делаем ребро между ними сплошным и пересчитываем $d(w)$.

Лемма 27. *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. □

8.3.3 Merge

Я напишу основную часть алгоритма, но часть про пересчёт структуры я предлагаю прочитать в оригинальной статье (на страницах 12-13 статьи [Geo+11]).

```
1:  procedure MERGE(v, u)
2:      if root(v)≠ root(w) then
3:          if l(v) ⩾ l(w) then
4:              link(v, w)
5:          else
6:              link(w, v)
7:      u := NCA(v, w)                          ▷ ищем общего предка v и w
8:      if u = v or u = w then
9:          return ∅ ▷ если общий предок совпадает с v или w, то пути уже совмещены
10:     else
11:         S_v = root(v)
12:         S_w = root(w)
13:         while Top(S_v) = Top(S_w) do
14:             pop(S_v)
15:             pop(S_w)
16:         if S_v ≠ ∅ and parent(Top(S_v)) = u then
17:             x := Top(S_v)
18:         else
19:             x := solid(u)
20:         if S_w ≠ ∅ and parent(Top(S_w)) = u then
21:             y := Top(S_w)
22:         else
23:             y := solid(u)
24:         if x < y then
25:             x := y
26:             swap(v, w)
27:             swap(S_v, S_w)
28:         while x < w do                  ▷ начало merge step z := Top(S_w)
29:             while z ≠ null and x > parent(z) do
```

30: replace z by the node below it on S_w
31: Make it null if we reached bottom of S_w
32: **if** z = null **then**
33: t := topnode(x, w)
34: **else**
35: t := topnode(x, parent(z))
36: parent(x):=parent(t)
37: **Update data structure**
38: x := t
39: swap(v, w)
40: swap(S_v, S_w) ▷ конец **merge step**

Новый **merge step** показан на рисунке 19:

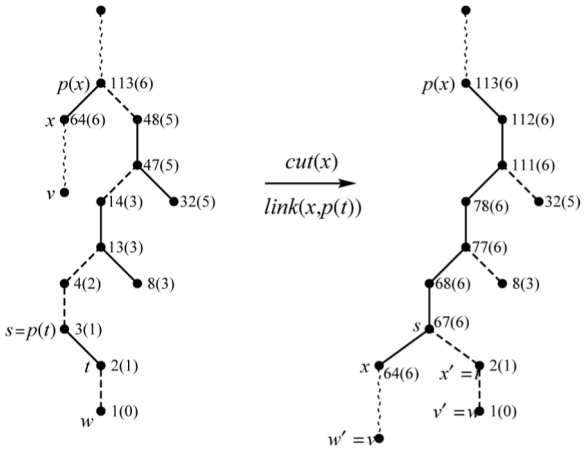


Рис. 19: new merge step

Небольшое замечание. Для того, чтобы понять следующую лемму, по-хорошему, надо понимать ту самую часть про **Update data structure**, которую я опустил. Но для леммы достаточно понимать, что мы обновляем снова используя очередь Q, в которую мы добавляем вершины только после того, как достаём их из одного из стеков S_w и S_v.

Лемма 28. *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 $\mathcal{O}(\log n)$.

Если не учитывать **topnode**, то время, затраченное на *merge* это $\mathcal{O}(1)$ за каждый **merge step** плюс $\mathcal{O}(1)$ за каждую вершину, которую мы добавляем в Q.

Из первой подсекции (лемма 26) 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.

□

8.3.4 Поисковая структура на путях

Мы будем использовать немного прокаченное **splay tree**.

Определение 20. Finger — узел, посещённый в splay-дереве последний раз.

Мы доказывали в третей секции (теорема 9), что аморитизированная стоимость операций в **splay**-дереве равна $\mathcal{O}(\log(d+1))$, где $d = |f' - f|$, а f' и f — новый и старый **fingers**.

Определение 21. $\log(d+1)$ — время Коула для данной операции.

У времени Коула есть одно полезное свойство:

Лемма 29. *Добавление посещения узла между операциями не уменьшает время Коула.*

Доказательство. Несложно придумать самим. За деталями смотри доказательство леммы 3.4 на странице 16 в статье [Geo+11]. □

Наша цель его амортизировать в нашей структуре. Но возникает одна проблема: в процессе **link** и **merge** мы можем несколько раз добавлять вершину вниз нашего дерева и удалять из корня поочередно, что даст нам логарифм высоты дерева (так как расстояние между **fingers** будет как раз вся высота). А мы хотим получить амортизированно получить $\mathcal{O}(1)$. Поэтому мы будем добавлять все вершины с задержкой. Тогда все **fingers** будут рядом при добавлениях и удалениях в структуру. Но, тогда **topnode** будет не просто бинарным поиском, а чуть сложнее.

Хранить эти дополнительные вершины будем следующим образом: на первую вершину сделаем указатель из самой правой вершины **splay-дерева**. А новые просто будем последовательно добавлять друг за другом.

Реализация **topnode(x,e)** выглядит следующим образом: мы делаем бинарный поиск x в **splay-дереве**, для пути для вершины e. 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, 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**.

Лемма 30. *Если поисковая структура это* ***splay-дерево*** *с задержками, как описано выше, то суммарное время на все операции с обновлениемя поисковых структур и на все* ***topnode*** *— это* $\mathcal{O}(m \cdot \log n)$.

Доказательство. Лемма 3.5 на странице 16 в статье [Geo+11]. **Но мы считаем, что это утверждение без доказательства!** □

9 Структуры целочисленных данных

Записал: Михаил Мрыхин

Определение 22. Модель Word RAM: единица данных - слово из w бит. Прочитать или записать слово, сравнить два, вычислить результат базовых арифметических ($+, -, \times$, div, mod) и побитовых логических операций, а также перейти по указателю можно за константу.

Хотим построить структуру, поддерживающую множество $S \subset U = \{0, \ldots, 2^w - 1\}$ с операциями insert, delete, find, pred, succ, используя внутреннюю структуру данных для ускорения. Оказывается, в данной модели можно добиться выполнения таких операций за константу (зависящую от размера модели как $\mathcal{O}(\log w)$, что, впрочем, равно $\mathcal{O}(\log \log U)$ от максимального размера структуры), причём как минимум двумя разными способами.

9.1 Деревья ван Эмде Боаса

Для удобства предположим, что $w = 2^m$.

structure TREE
    root
structure NODE
    size
    empty
    min, max
    clusters[]
    summary

При размере k дерево ван Эмде Боаса (van Emde Boas (да, это один человек) tree) с корнем в текущей вершине содержит данные об элементах из $\left\{0, \ldots, 2^{2^k} - 1\right\}$. Обозначим за high($x$) и low($x$) битовые слова половинной длины, образованные первой и второй половиной записи x соответственно; тогда вспомогательное поддерево summary хранит high(x) для всех содержащихся в дереве x, а low(x) хранится в clusters[high(x)]. Вдобавок, наименьший и наибольший элементы в дереве хранятся лишь в переменных min и max, а на summary и clusters не влияют (в частности, при $k = 0$ последние отсутствуют вовсе). По сути, мы производим серединное сечение полного двоичного дерева возможных ключей по высоте: верхняя половина отправляется в summary, а компоненты нижней половины - в clusters.

1: **procedure** INSERT(x)

```
2:     if empty then
3:         min := x
4:         max := x
5:     else if x < min then
6:         oldmin := min
7:         min := x
8:         insert(oldmin)
9:     else if x > max then
10:        oldmax := max
11:        max := x
12:        insert(oldmax)
13:    else if x > min and x < max then
14:        if clusters[high(x)].empty then
15:            summary.insert(high(x))
16:        clusters[high(x)].insert(low(x))
17: procedure DELETE(x)
18:    if empty then
19:        return
20:    if min = x then
21:        if max = x then
22:            empty := True
23:        else if summary.empty then
24:            min := max
25:        else
26:            newmin := clusters[summary.min].min
27:            delete(newmin)
28:            min := newmin
29:    else if max = x then
30:        if summary.empty then
31:            max := min
32:        else
33:            newmax := clusters[summary.max].max
34:            delete(newmax)
35:            max := newmax
36:    else
37:        clusters[high(x)].delete(low(x))
38:        if clusters[high(x)].empty then
39:            summary.delete(high(x))
40: procedure FIND(x)
41:    if empty then
42:        return false
43:    if min = x or max = x then
44:        return true
45:    return clusters[high(x)].find(low(x))
```

```
46: procedure SUCC(x)
47:     if empty or x ≥ max then
48:         return null
49:     if x < min then
50:         return min
51:     if clusters[high(x)].max > low(x) then
52:         return compose(high(x), clusters[high(x)].succ(low(x)))
53:     if summary.max > high(x) then
54:         next := summary.succ(high(x))
55:         return compose(clusters[next], clusters[next].min)
56:     return max
57: procedure PRED(x)
58:     аналогично SUCC
```

Как можно видеть, почти каждый запрос к дереву размера k состоит из константного набора операций и не более одного запроса к дереву размера $k-1$. Исключения разбиваются на две группы: запросы к тому же дереву при смене максимума/минимума, которые случаются не более раза на каждой высоте, и ветвящиеся запросы при добавлении первого/удалении последнего элемента в кластере, в случае которых запрос к кластеру происходит за $\mathcal{O}(1)$ вне зависимости от его максимального размера, а настоящая ветвь рекурсии уходит в summary. Как результат, все запросы работают за $\mathcal{O}(m) = \mathcal{O}(\log w) = \mathcal{O}(\log\log U)$ по времени, но вот по пространству структура занимает $\Theta(U)$.

Можно улучшить асимптотику памяти, если заменить массивы указателей на хэш-таблицы, но в таком случае для малых n нам придётся хранить $\mathcal{O}(nw)$ (потому что каждый элемент занимает место в хэш-таблице, summary и cluster, с рекуррентой $S(m) = \mathcal{O}(1) + 2S(m-1)$, что разрешается в $S(m) = \mathcal{O}(2^m) = \mathcal{O}(w)$).

Теперь разобьём элементы на (последовательные) группы размера в пределах $[\frac{w}{4}; 2w]$, на каждой построим двоичное дерево поиска. Из каждой группы будем хранить в дереве вЭБ её минимум с указателем. Всё работает почти так же, как и раньше, но теперь надо сначала найти нужную группу, а потом уже там искать/менять. К счастью, это всё тоже работает за $\mathcal{O}(\log w)$. Иногда надо поддерживать размер малых деревьев слиянием/делением (с соответствующим удалением/вставкой минимумов в дерево вЭБ), но это тоже не меняет асимптотику. Наконец, места это всё занимает $\mathcal{O}(n)$, как нам и хотелось.

9.2 X- и Y-быстрые деревья

Снова рассмотрим полное двоичное дерево всех возможных значений ключей. Оставим только вершины, лежащие на пути от корня к присутствующим элементам. В каждую вершину, у которой нет правого ребёнка, поместим указатель на наибольший элемент в левом поддереве; аналогично в другом направлении. Листья вместо этого снабдим ссылками на следующий и предыдущий, как в двустороннем списке. Все вершины на одном уровне будем хранить в хэш-таблице. Это и есть X-быстрое дерево (X-fast tree/trie).

Найти элемент можно за $\mathcal{O}(1)$, так как все листья находятся в одной хэш-таблице. Найти соседей наличествующего элемента — очевидно, тоже. В более общем случае — за $\mathcal{O}(\log w)$: отыщем двоичным поиском по хэш-таблицам уровней максимальный общий префикс, после чего пройдём по вспомогательному указателю в соседа (и ещё раз шагнём по списку, если сосед не с той стороны). Вставка и удаление делаются за $\mathcal{O}(w)$ проходом сверху вниз или снизу вверх соответственно. Наконец, занимает это всё $\mathcal{O}(nw)$ места.

Обратив внимание на последнюю оценку, сделаем тот же трюк, что и с деревом ван Эмде Боаса: разобьём все элементы на группы размера w, после чего построим на них двоичные деревья поиска, а в X-быстрое дерево запишем только минимум и указатель. Это и есть Y-быстрое дерево (Y-fast tree/trie).

Асимптотика такая же, как и у оптимизированного дерева вЭБ: $\mathcal{O}(\log w)$ на поиск (единственное, что малые деревья подтормаживают), $\mathcal{O}(\log w)$ на соседей (одинаково в глобальном и малых деревьях). Единственное затруднение вызывают вставка и удаление, но тут можно заметить, что они в основном затрагивают малые деревья (и работают там за $\mathcal{O}(\log w)$), а перестройка X-быстрого требуется только при их слиянии/делении. Так как после этих операций их размер попадает на отрезок $[\frac{w}{2}, \frac{9w}{8}]$, а перед следующей должен выйти из $[\frac{w}{4}; 2w]$, то перестройки могут происходить не чаще, чем раз за $\frac{w}{4}$ операций над малым деревом, что амортизирует вышеприведённое $\mathcal{O}(w)$ в константу. Наконец, занимает это все $\mathcal{O}(n)$ места.

10 Динамизация структур данных

Записал: Всеволод Евтушевский

Пусть у нас есть задача поиска:

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

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

10.1

Определение 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.

Пример 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 $Q(n)$.

Задача 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

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

Замечание. 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.

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

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

10.2

Определение 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:

Задача 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 $Q(x, M) - \blacklozenge Q(x, G)$

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 $|G| > \frac{1}{2}|M|$
  Build $M \coloneqq M \setminus G$
    $G \coloneqq \varnothing$

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.

Задача 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 $Q(x) = Q(x, M) \blacklozenge Q(x, I) - \blacklozenge Q(x, G)$

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 $Q(x) = Q(x, M) \blacklozenge Q(x, I) \blacklozenge Q(x, I') - \blacklozenge Q(x, G) - \blacklozenge Q(x, G')$

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,

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

10.3

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.

11 Структуры данных и путешествия во времени

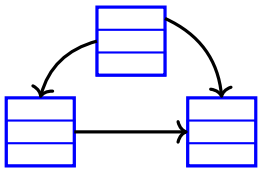
Записала: Татьяна Белова

Оригинальная статья: [DIL07].

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 ретроактивность.

11.1 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.



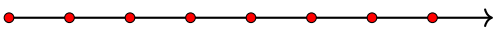
Операции, которые поддерживает структура:

- $x =$ 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.

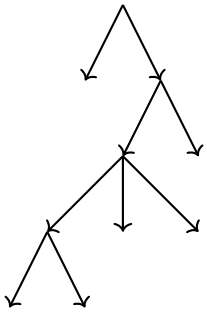
[Dri+86] сделали алгоритм с амортизированным $\mathcal{O}(1)$ overhead-ом.

[Bro96] сделали алгоритм с $\mathcal{O}(1)$ overhead-ом в худшем случае.

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

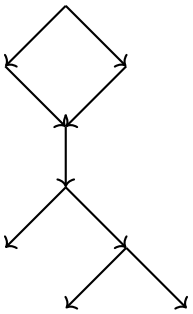
   Каждый раз, когда мы изменяем прошлое, мы создаем новую версию структуры. Получается дерево версий.

   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 ([Dri+86]), 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, [FK03] 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 ([CIL12]).

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 $\mathcal{O}(\log n)$ ([DLP08]).

Приложения частичной персистентности

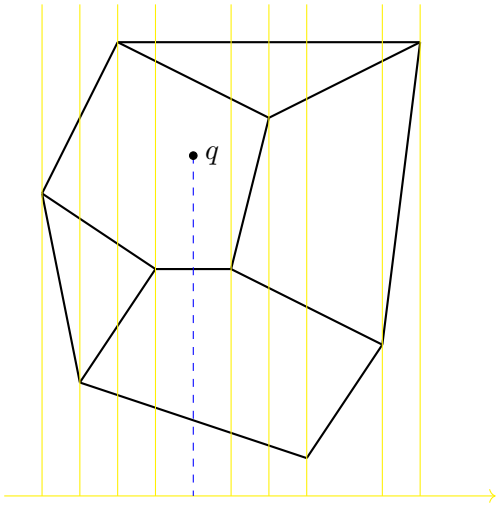
Задача Point location

Дан плоский граф G с ребрами-отрезками. Запросы: по данной точке q на плоскости определить, в какой грани G 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 сбалансиро-

ванном BST T. Таким образом, у нас появилось n версий этого BST, по версии для каждой точки.



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

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

11.2 Ретроактивность

Пусть есть структура данных, поддерживающая какие-то updates и queries.

В случае ретроактивности 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, как в случае персистентности, ветка будет всего одна, и все действия, произошедшие после нашего изменения, по-прежнему будут учитываться.

Операции:

- Insert(t, *update*) – добавить изменение *update* в момент времени t
- Delete(t) – удалить изменение, произошедшее в момент времени t
- Query(t, *query*) – сделать запрос query в момент времени 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.

Ретроактивность бывает разных видов:

1) **Частичная ретроактивность**

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

2) **Полная ретроактивность**

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

Введем параметры для измерения эффективности ретроактивной структуры.

- m – суммарное 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 операции, если мы делаем структуру ретроактивной.

Частичная ретроактивность

1) Insert(t, *update*)
   Если операции коммутативные, то Insert() делается с константным overhead-ом, так как мы всегда можем применить операцию в самом конце.
   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 ретроактивной с overhead-ом $\mathcal{O}(1)$.

Еще оказывается, что структуру данных можно сделать частично ретроактивной 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 ретроактивность с константным overhead-ом.

Полная ретроактивность

Мы снова рассмотрим некоторый подкласс структур данных, а именно структуры данных для *decomposable search problems*, и научимся наделять их полной ретроактивностью.

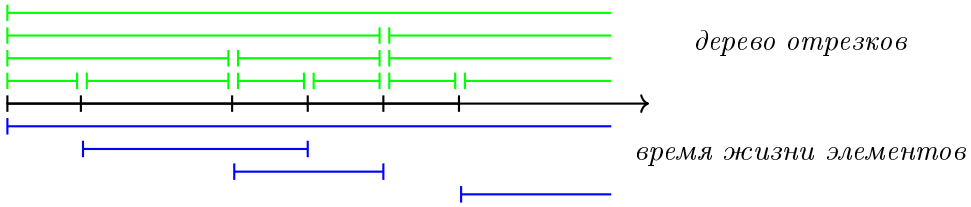
Напоминание: decomposable search problem (d.s.p.) — это задача поиска с дополнительным условием $Q(x, D \cup D') = Q(x, D) \blacklozenge Q(x, D')$, где операция $\blacklozenge$ выполняется за константное время.

Теорема 31. *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 ретроактивной, 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 структуры.

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 $\mathcal{O}(\log n)$ узлами нашего дерева, в них и запишем этот элемент. В каждом узле храним элементы в нашей структуре.



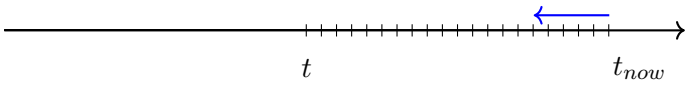
Теперь, когда нам нужно добавить или удалить элемент, происходит следующее. Если нужно, добавим новый момент времени, т.е. разделим один из отрезков на два и перебалансируем дерево. Потом добавим или удалим элемент из всех $\mathcal{O}(\log m)$ соответствующих узлов.

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

Теорема 32. *Структуру данных с операциями, работающими за $\mathcal{O}(T(n))$, можно сделать полностью ретроактивной, при этом операции для t_{now} будут работать за время $\mathcal{O}(T(n))$, а ретроактивные 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. □

Теорема 33. *Существует структура данных в модели straight-line-program с операциями update и query, работающими за $\mathcal{O}(1)$, такая, что любая частично ретроактивная 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 значение многочлена $a_n x^n + \dots + a_1 x + a_0$ в точке 0.

После чего сделаем ретроактивный 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 ([FHM01]).

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. □

11.3 Нижняя оценка на создание ретроактивной структуры: word RAM

Определение 25. *Модель вычислений* CELL-PROBE — это WORD RAM, в которой единственной операцией, имеющей цену и учитывающейся при подсчёте времени работы,

является доступ к данным. Данные по-прежнему хранятся в блоках фиксированного размера, чтение целого блока имеет единичную стоимость.

Определение 26. *Свёртка двух последовательностей слов* $\langle x_0 \dots x_{k-1} \rangle$, $\langle y_0 \dots y_{k-1} \rangle$ — это последовательность слов

$$\langle w_0 \dots w_{2k-1} \rangle, \quad \text{такая, что} \quad w_i = \sum_{a+b=i} x_a \times y_b. \tag{5}$$

Задача 5 (Динамическая свёртка)**.** Требуется поддерживать в памяти набор слов $\langle x_0 \dots x_{2k-1} \rangle$ и эффективно отвечать на следующие запросы:

1) $\texttt{change}_k(x)$: присвоить x_k значение x.

2) $\texttt{alg\_query}_k$: вернуть k-ую компоненту свёртки

$$\langle x_0 \dots x_{k-1} \rangle \otimes \langle x_k \dots x_{2k-1} \rangle .$$

Теорема 34 ([FHM98], Теорема 1)**.** *Задача 5 не решается на моделях* CELL-PROBE *и* WORD RAM *быстрее, чем* $\Omega(\sqrt{n})$ *в худшем случае.*

Теорема 35 ([DIL07], Теорема 3)**.** *Существует структура данных в модели* WORD RAM *со словами размера* $\log n$*, которая поддерживает обновление и запрос за* $O(1)$*, но любая частично ретроактивная структура данных с теми же функциями требует время*

$$\Omega\left(\sqrt{\frac{r}{\log r}}\right) \quad amortised$$

либо на обновление, либо на запрос.

Доказательство. Приведём пример такой структуры. Она будет поддерживать набор слов $\langle w_1 \dots w_m \rangle$ линейного размера. Изначально все эти слова нулевые. Будут доступны следующие операции:

$$\begin{array}{ll} \texttt{update} & w_i \coloneqq x \\ & w_i \coloneqq w_j + w_k \\ & w_i \coloneqq w_j \times w_k \\ \texttt{query} & w_i = ? \end{array}$$

Модель вычислений WORD RAM позволяет делать такие операции за константное время. Над такой структурой данных мы можем за $O(n \log n)$ времени и $O(n)$ памяти сгенерировать и проделать последовательность обновлений, которая посчитает свёртку (5) двух половинок хранимого набора слов: это называется быстрое дискретное преобразование Фурье.

Теперь пусть мы хотим делать ретроактивные Insert вида $w_i \coloneqq x$ в начальный момент времени. Понятно, что возможность такой операции равносильна решению Задачи 5 о динамической свёртке, что не может быть сделано быстрее, чем за $\Omega(\sqrt{n})$ на каждое изменение.

При этом при вычислении преобразования Фурье было сделано $r = O(n \log n)$ операций. Чтобы выразить время операции через r, заметим, что $n = \Omega(r/\log r)$. Мы получили, что время решения задачи о динамической свёртке —

$$\Omega(\sqrt{n}) = \Omega\left(\sqrt{\frac{r}{\log r}}\right).$$

□

12 От частичной ретроактивности к полной

Записал: Борис Золотов

12.1 Пессимистический результат

Теорема 36 ([DIL07], Теорема 4)**.** *Существует структура данных в модели* CELL-PROBE*, которая поддерживает частично ретроактивные обновления за константу, но при этом любая полно ретроактивная структура с тем же функционалом будет требовать $O(\log n)$ времени для запросов к прошлому (напомним, m — время жизни структуры).*

Доказательство. Приведём пример такой структуры. Она будет поддерживать набор чисел со следующими операциями:

`update`	insert(c)
`query`	вернуть сумму всех чисел

Для частично ретроактивной структуры требуются также следующие операции:

Insert $(t$, «insert(c)»$)$
Delete (t)

Понятно, что с такими запросами ретроактивные операции можно реализовать за константу: нужно всего лишь либо добавлять к текущей сумме, либо отнимать от неё соответствующее число.

Тем не менее, полно ретроактивная структура данных с таким функционалом умела бы решать задачу о динамической префиксной сумме. Действительно, мы должны уметь вставлять число в любое место массива и считать сумму от самого начала до произвольного момента. Для задачи о динамической префиксной сумме существует нижняя оценка: $\Omega(\log n)$ за операцию.

□

12.2 Позитивные результаты

Сейчас мы приведём общую схему, как из частично ретроактивной структуры данных сделать полно ретроактивную.

Теорема 37 ([DIL07], Теорема 5)**.** *Дана частично ретроактивная структура данных в модели* POINTER MACHINE *с константной входящей степенью. Пусть она требует время $T(m)$ на ретроактивное обновление и $Q(m)$ на запросы о её текущем состоянии. Тогда у этой структуры данных существует полно ретроактивная версия со следующими характеристиками:*

$$\begin{aligned} \textit{Время на ретроактивное обновление} &\quad O\left(\sqrt{m}\cdot T(m)\right), \\ \textit{Время на запрос в произвольный момент} &\quad O\left(\sqrt{m}\cdot T(m)+Q(m)\right), \\ \textit{Используемая память} &\quad O\left(m\cdot T(m)\right). \end{aligned}$$

Доказательство. Будем хранить $\sqrt{m}$ версий структуры данных — $D_1, D_2, \ldots, D_{\sqrt{m}}$ — равноудалённо раскиданных по временно́й оси, отражающей изменения в структуре. Когда приходит запрос о некотором моменте в прошлом, будем брать ближайшую к нему версию D_i и вносить изменения в неё: изменений будет не больше, чем $\sqrt{m}$. Также будем хранить последовательность вносимых изменений.

Как сделать ретро-изменение. Пусть мы хотим изменить структуру данных в момент t. Найдём ближайший $t_i < t$, для которого мы храним версию D_i. Внесём все обновления между моментами времени t_i и t, сделаем запрос про настоящее состояние структуры D_i, отменим все внесённые обновления.

Как сделать ретро-обновление. Внесём обновление, которое хочется сделать в момент времени t, во все структуры данных после этого момента.

Как быть с памятью. Мы можем сделать структуры D_i полно персистентными, чтобы вносить изменения в их прошлые состояния. Это увеличивает память и время, необходимые для работы со структурой, в константное число раз, смотреть Раздел 11.1, пункт 2.

Собственно, рассмотрим частично ретроактивную структуру данных с полно персистентными D_i. Если дана последовательность из m обновлений, мы применим их к изначальной структуре, сохраняя указатели на версии, получаемые после $i\sqrt{m}$ обновлений — это и будут D_i. Ретро-обновления создают новую ветку в дереве версий, отходящую от D_i.

Каждые $\sqrt{m}/2$ ретро-обновлений мы будем перестраивать вообще всю структуру данных, чтобы сохранять равномерную распределённость её версий, которые мы храним, и разумное количество изменений между ними. На перестроение требуется $O\left(m\cdot T(m)\right)$ времени — размазав его по $\sqrt{m}$ сделанным операциям, получим $O\left(\sqrt{m}\cdot T(m)\right)$ амортизированно, которое прибавим и ко времени запроса, и ко времени ретро-обновления.

Оценка на память $O\left(m\cdot T(m)\right)$ очевидна — собственно, бóльшему объёму памяти в этой задаче неоткуда взяться. □

Если известно, что размер структуры оказывается решительно меньше, чем количество обновлений, которые с ней проделаны, результат может быть улучшен:

Теорема 38 ([Che+18], Теорема 4)**.** *Дана частично ретроактивная структура данных в модели* WORD RAM*. Пусть её размер — n, а также*

1) *$O(n)$ запросов позволяют полностью восстановить текущее состояние структуры,*

2) *Зная текущее состояние структуры, за $O(n)$ операций можно превратить пустую структуру данных в её копию,*

3) *Эта структура частично ретроактивная со временем $T_{\text{op}}(n, m)$ на ретроактивные запросы и обновления.*

Тогда у этой структуры данных существует полно ретроактивная версия с амортизированным временем работы

$$O\left(n \log m \cdot T_{\text{op}}(n, m)\right).$$

Напомним, n — текущий размер структуры, а m — время её жизни, то есть количество проделанных изменений.

Доказательство. Воспользуемся сбалансированным деревом поиска, Scapegoat tree, на последовательности изменений, обозначим его τ. Каждый его узел соответствует отрезку в списке и хранит частично ретроактивную структуру данных, содержащую обновления из этого отрезка.

Чтобы вставить или удалить некоторое обновление в конкретный момент времени t, нужно «потрогать» $\log m$ узлов, составляющих путь к t от корня τ, на работу с каждым узлом тратя $O\left(T_{\text{op}}(n, m)\right)$ времени.

Для реализации ретро-запроса рассмотрим путь от корня τ до интересующего нас момента времени t. Есть (естественно) не более $\log m$ моментов, когда мы спускались направо, оставляя слева от себя отрезки времени, предшествующие t. Запомним всех *левых братьев* тех узлов, в которые мы спустились направо. В сумме под ними находится $O(n)$ моментов времени. Обозначим этих левых братьев через $S_1 \dots S_\ell$.

Возьмём пустую структуру данных S. Для i от 1 до ℓ проделаем следующее:

1) Присоединим $O(n)$ обновлений, имеющихся в S, к S_i,

2) Сделаем $O(n)$ запросов о текущем состоянии S_i,

3) За $O(n)$ операций приведём S к такому состоянию.

Мы потратили на эту процедуру $O\left(n \cdot \log m \cdot T_{\text{op}}(n, m)\right)$ времени. После того, как мы получили S, нужно откатить все обновления с $S_1 \dots S_\ell$, чтобы использовать эти структуры в будущем. □

Таким образом, мы умеем добиваться умножения времени работы как на $\sqrt{m}$, так и на $n \cdot \log m$. Оказывается, можно добиться того, чтобы время работы умножалось на *минимум* из этих параметров.

Следствие 39 ([Che+18]). *Частично ретроактивную структуру данных можно сделать полно ретроактивной с умножением времени работы на*

$$\min\left\{\sqrt{m},\quad n\cdot\log m\right\}.$$

Доказательство. Будем поддерживать две структуры одновременно. Когда будет приходить ретро-запрос, будем возвращать ответ той структуры, которая обработала запрос быстрее. □

13 Ретроактивность для конкретных структур данных

13.1 Двунаправленная очередь

Двунаправленная очередь — это структура данных, хранящая некоторые элементы и поддерживающая следующие операции:

`update`	pushL(x): вставить элемент слева
	popL: удалить элемент слева
	pushR(x): вставить элемент справа
	popR: удалить элемент справа
`query`	Узнать крайний правый, крайний левый элементы

Теорема 40 ([DIL07], Теорема 7). *Двунаправленную очередь можно сделать полно ретроактивной, так что ретро-обновления и ретро-запросы будут занимать время $O(\log m)$, а запросы о текущем состоянии структуры — $O(1)$.*

Доказательство.

Описание структуры данных Элементы текущей версии очереди будут храниться в массиве A. Также нам потребуются числа L и R — индексы левого и правого краёв рабочего участка массива A. Если очередь за последние несколько операций становилась короче, то за этими левыми и правыми краями будет оставаться что-то написанное, мы не будем его стирать, но и не будем мыслить его частью очереди.

В первом приближении понятно, как делать push и pop в настоящий момент времени. Для push слева уменьшим на единицу индекс левого края и запишем на левый край новый элемент (возможно, поверх чего-то старого); для pop слева просто увеличим на единицу индекс края. Справа — симметрично.

Рассмотрим два списка: U_L и U_R. Это отсортированные по времени списки операций, соответственно, с правым и левым краями очереди. У каждой операции в этих списках есть вес: push имеет вес $+1$, pop — -1. Заметим, что индекс, например, правого края равен сумме весов всех операций в списке U_R.

Сделаем из U_L, U_R сбалансированные деревья поиска: операции будут храниться в листьях, а каждый узел будет хранить сумму весов всех операций его поддерева. Значение R в момент, когда была сделана какая-то операция, можно посчитать, сложив значения, хранимые левыми братьями узлов, в которые мы спускались направо по пути к означенной операции.

Реализация ретро-операций При ретро-операции в момент t спустимся к этому моменту времени в соответствующем дереве (U_L или U_R) и запишем операцию в новом листе. При этом прибавим вес операции к суммам во всех узлах, через которые мы спускались.

Чтобы реализовать ретро-запрос, в каждом узле деревьев поиска будем также хранить наибольшее и наименьшее значения префиксных сумм, достигаемых в поддереве этого узла. Эти значения говорят нам, в каких пределах были левый и правый край рабочего участка очереди на соответствующем отрезке времени.

Дан момент времени t, нужно выяснить, где в этот момент находился конец рабочего участка массива и что там было записано. Индекс i конца рабочего участка мы можем посчитать, спустившись по дереву, как описано выше.

Пусть мы работаем с правым краем очереди. Найдём в U_R последнюю операцию перед моментом времени t. Поднимемся из её листа в корень дерева, а затем спустимся обратно, каждый раз выбирая правое поддерево из тех, для которых i находится между наибольшим и наименьшим значениями префиксных сумм, хранящимися в вершине поддерева.

Такой выбор поддерева гарантирует нам то, что мы попадём в некоторый элемент, хранящийся в нужной нам ячейке массива, при этом данный элемент появился там на нужном нам *переписывании* — то есть, он написан поверх элемента, который был на этом месте раньше, чем время t, и его ещё не успел сменить элемент, который появится там позже. Действительно, условие «находиться между максимумом и минимумом префиксных сумм» равносильно условию «оставаться в рабочем участке массива, не будучи перезаписанным другим элементом». □

13.2 Очередь с приоритетом: описание работы с элементами

Очередь с приоритетами поддерживает следующие операции:

`update`	insert(k): вставить элемент, имеющий приоритет k
	delete_min(): удалить наименьший элемент (с наибольшим приоритетом)
`query`	find_min

Будем считать, что все элементы, побывавшие в очереди за время её существования, различны, и будем обозначать их промежутки жизни отрезками на соответствующей высоте между двумя моментами времени, смотреть Рисунок 20. Удалению элемента из очереди будет соответствовать вертикальный отрезок. То есть, каждый элемент, побывавший в очереди и удалённый из неё, образует уголок. Эти уголки не могут пересекаться, так как каждый раз удаляется наименьший элемент.

Сейчас мы опишем, что происходит, когда в историю операций очереди мы ретроактивно добавляем вставку или удаление элемента (или отменяем эти операции).

Вставка вставки или отмена удаления Рассмотрим жизнь некоторой очереди с приоритетом — например, той, что на Рисунке 20. Красные отрезки — это удаления её элементов на соответствующем шаге операцией delete_min.

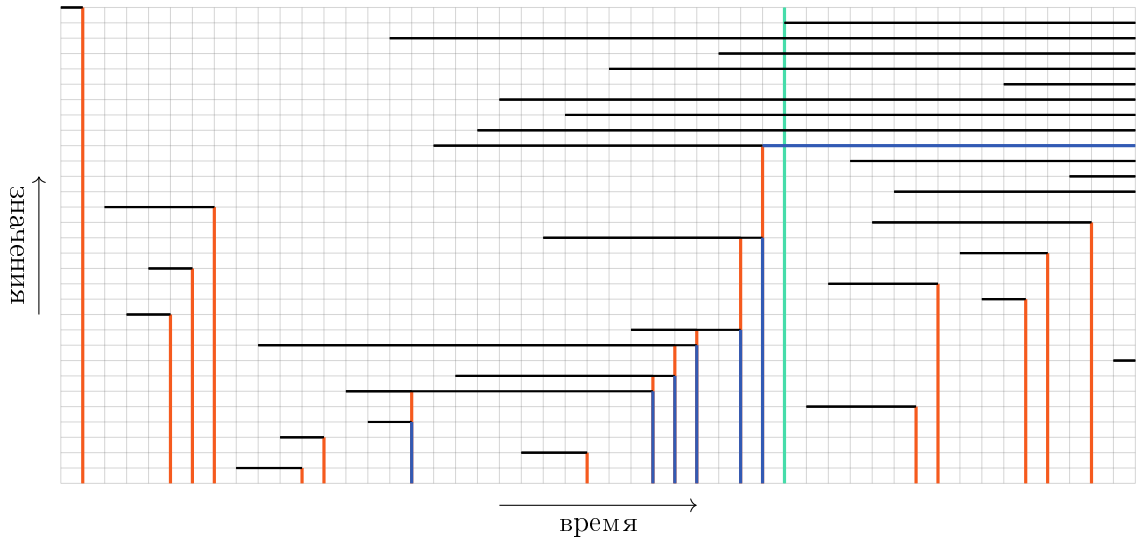


Рис. 20: Ретроактивная очередь с приоритетами

Ретроактивно вставим в очередь элемент со значением 4. Он меньше некоторых из элементов, которые были удалены из очереди — соответственно, они будут удаляться позже, а один из них так вообще останется в очереди и просуществует там до её нынешнего состояния: смотреть синие линии на рисунке, ими обозначены различия между очередью до и после ретроактивной вставки.

Какой же элемент появится в текущем состоянии очереди после ретроактивной вставки элемента со значением k в момент времени t? Либо этот элемент достаточно большой, тогда он сам. Либо этот элемент помешает удалиться какому-то другому элементу впоследствии — так вот не удалится *наибольший* элемент из тех, который раньше был удалённым: смотреть на горизонтальную синию линию. То есть элемент, который надо добавить —

$$\max\{k, k'\} \quad k' \text{ удалён после момента времени } t.$$

Ретроактивное удаление операции delete_min в момент времени t — это то же самое, что вставка в тот же момент времени только что удалённого элемента. Соответственно, после этого в текущем состоянии очереди появится наибольший элемент из удалённых после момента времени t.

То же самое на языке мостов *Мостом* будем называть состояние очереди, которое является подмножеством текущего состояния. Один из мостов на рисунке 20 отмечен зелёной вертикальной чертой. Также является мостом, например, начальное состояние, когда очередь пустая.

Утверждение 41. *Пусть мы ретроактивно вставляем элемент в момент времени t, а t' — последний мост перед t. Тогда $\max k'$ — наибольший элемент, вставленный после t' и не попавший в текущее состояние.*

Список литературы

[And89] Arne Andersson. «Improving partial rebuilding by using simple balance criteria». в: *Workshop on Algorithms and Data Structures*. Springer. 1989, с. 393—402.

[Bro96] Gerth Stolting Brodal. «Partially persistent data structures of bounded degree with constant update time». в: *Nord. J. Comput.* 3.3 (1996), с. 238—255.

[Che+18] Lijie Chen и др. «Nearly Optimal Separation Between Partially And Fully Retroactive Data Structures». в: *20th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2018)*. под ред. David Eppstein. Malmö, Sweden, июнь 2018, с. 1—12.

[CIL12] Sébastien Collette, John Iacono и Stefan Langerman. «Confluent persistence revisited». в: *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete algorithms*. SIAM. 2012, с. 593—601.

[Col+00] Richard Cole и др. «On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting log n-Block Sequences». в: *SIAM Journal on Computing* 30.1 (2000), с. 1—43. DOI: 10.1137/S0097539797326988. eprint: https://doi.org/10.1137/S0097539797326988. URL: https://doi.org/10.1137/S0097539797326988.

[Col00] Richard Cole. «On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof». в: *SIAM Journal on Computing* 30.1 (2000), с. 44—85. DOI: 10.1137/S009753979732699X. eprint: https://doi.org/10.1137/S009753979732699X. URL: https://doi.org/10.1137/S009753979732699X.

[Dem+09] Erik D. Demaine и др. «The Geometry of Binary Search Trees». в: *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '09. New York, New York: Society for Industrial и Applied Mathematics, 2009, с. 496—505. DOI: https://doi.org/10.5555/1496770.1496825.

[DIL07] Erik D Demaine, John Iacono и Stefan Langerman. «Retroactive data structures». в: *ACM Transactions on Algorithms (TALG)* 3.2 (2007), 13—es.

[DLP08] Erik D Demaine, Stefan Langerman и Eric Price. «Confluently persistent tries for efficient version control». в: *Scandinavian Workshop on Algorithm Theory*. Springer. 2008, с. 160—172.

[Dri+86] James R Driscoll и др. «Making data structures persistent». в: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 1986, с. 109—121.

[FHM01] Gudmund Skovbjerg Frandsen, Johan P Hansen и Peter Bro Miltersen. «Lower bounds for dynamic algebraic problems». в: *Information and Computation* 171.2 (2001), с. 333—349.

[FHM98] Gudmund Skovbjerg Frandsen, Johan P Hansen и Peter Bro Miltersen. «Lower Bounds for Dynamic Algebraic Problems». в: *BRICS* RS-98.11 (май 1998).

[FK03] Amos Fiat и Haim Kaplan. «Making data structures confluently persistent». в: *Journal of Algorithms* 48.1 (2003), с. 16—58.

[Geo+11] Loukas Georgiadis и др. «Data structures for mergeable trees». в: *ACM Transactions on Algorithms (TALG)* 7.2 (2011), с. 1—30.

[GR93] Igal Galperin и Ronald L Rivest. «Scapegoat Trees.» в: *SODA*. т. 93. 1993, с. 165—174.

[Luc88] Joan Marie Lucas. *Canonical forms for competitive binary search tree algorithms*. Rutgers University, Department of Computer Science, Laboratory for Computer . . ., 1988.

[ST83] Daniel Dominic Sleator и Robert Endre Tarjan. «A data structure for dynamic trees». в: *Journal of Computer and System Sciences* 26.3 (1983), с. 362—391. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(83)90006-5. URL: http://www.sciencedirect.com/science/article/pii/0022000083900065.

[ST85] Daniel Dominic Sleator и Robert Endre Tarjan. «Self-Adjusting Binary Search Trees». в: *J. ACM* 32.3 (июль 1985), с. 652—686. ISSN: 0004-5411. DOI: 10.1145/3828.3835. URL: https://doi.org/10.1145/3828.3835.

[Tar84] Robert Endre Tarjan. «Data Structures and Network Algorithms». в: *Society for Industrial and Applied Mathematics* (1984).